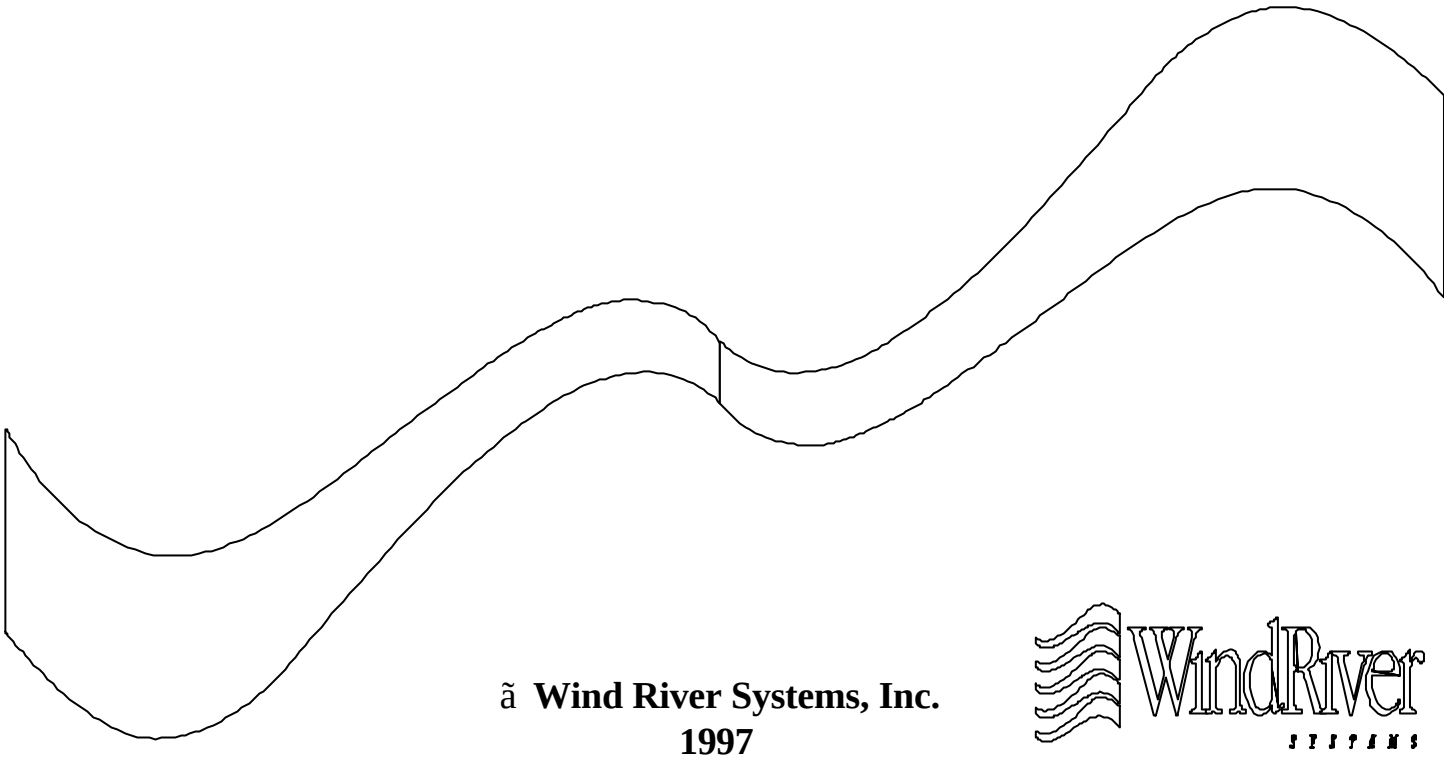
# Tornado™ BSP

## Training Workshop

Wind River Systems, Inc.
1010 Atlantic Avenue
Alameda, CA 94501

510-749-2148
FAX: 510-749-2378
training@wrs.com
http://www.wrs.com/training



Version 1.0.2, April 1998


This document is designed to support the Tornado BSP Training Workshop class. It is not designed as a stand-alone document, nor can it substitute for Wind River Systems BSP documentation. For information about the Wind River Systems training program, contact:

Training Department
Wind River Systems,
1010 Atlantic Avenue
Alameda, CA 94501

510-749-2148 (phone)
510-749-2378 (fax)
EMAIL: training@wrs.com

Wind River Systems S.A.R.L.
Inc. 27, Avenue de la Baltique
Bâtiment B4, LP739
91962 Les Ulis Cedex
France

33-1-69-07-78-78 (phone)
33-1-69-07-08-26 (fax)

Wind River Systems Japan/Asia-Pacific
Pola Ebisu Bldg. 11F
3-9-19 Higashi
Shibuya-ku
Tokyo 150
Japan

+81-03-5467-5900 (phone)
+81-03-5467-5877 (fax)



# Course Prerequisites

- General prerequisites:
  - Solid knowledge of C programming, and familiarity with general assembly level programming principles.
  - Experience writing device drivers using the C programming language.
  - Basic understanding of standard embedded systems hardware.
  - Basic understanding of VxWorks and debugging techniques.
  - Basic understanding of makefiles and building executable images.
- Functional knowledge of host platform and Tornado tools:
  - UNIX: user-level knowledge of make, csh, man, vi or emacs, etc.
  - Windows NT: user-level knowledge of Windows NT graphical and command-line user interfaces, file systems, and standard Windows editor.
  - Tornado tools: configuration of a target server to support various back end connection strategies, practical experience using CrossWind, and basic user-level knowledge of other Tornado tools.

# Course Objectives

- Overview of BSP responsibilities and integration issues.
- Choose a BSP development strategy.
- Manage a BSP development environment.
- Choose BSP development tools.
- Use WDB agent for BSP development.
- Perform pre-kernel initialization.
- Perform post-kernel initialization.
- WRS guidelines for device driver design.
- Manage interrupts in a BSP.
- Integrate timer drivers.
- Manage memory.
- Integrate serial communication controller for debugging.
- Build and support VxWorks images such as:
  - Loadable images
  - ROM-based images (compressed/uncompressed)
  - ROM-resident images
- Writing and testing WRS a compliant BSP.

# What Course Does Not Cover

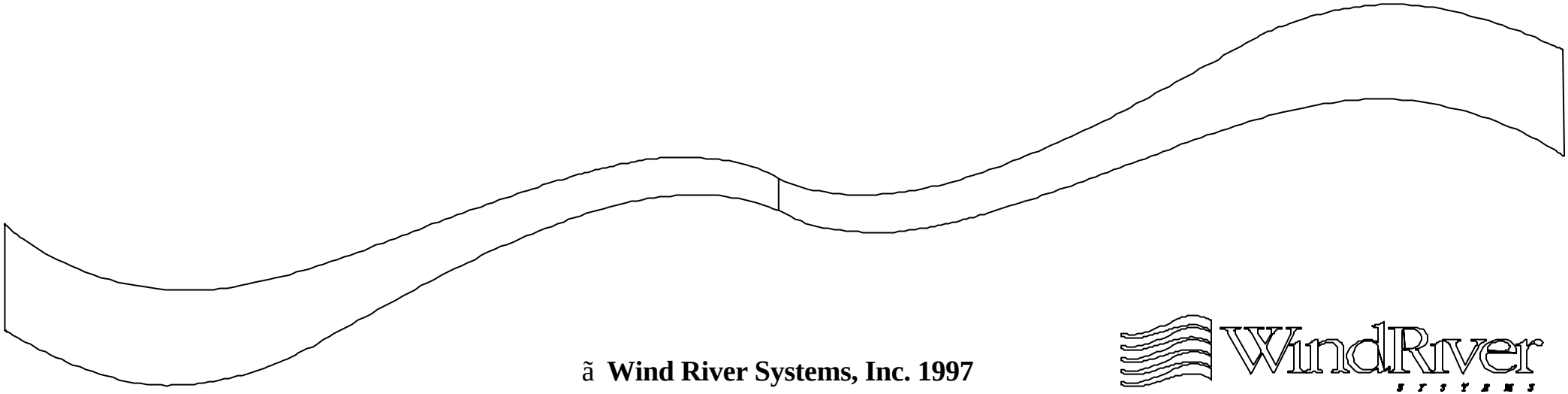
- Writing generic device drivers (network, SCC, SCSI, etc.). Material covered in Tornado Device Driver Workshop.
- Using Tornado tools and non-BSP VxWorks facilities. Material covered in Tornado Training Workshop.
- Architecture port issues.
- Specific vendor hardware:
  - Target devices.
  - Non-WRS development tools.

# Table of Contents

**Chapter**

# Chapter - 1

## Overview



# Overview

1.1 Integration Issues

VxWorks Boot Sequence

Tornado Directory Structure

Conventions and Validation

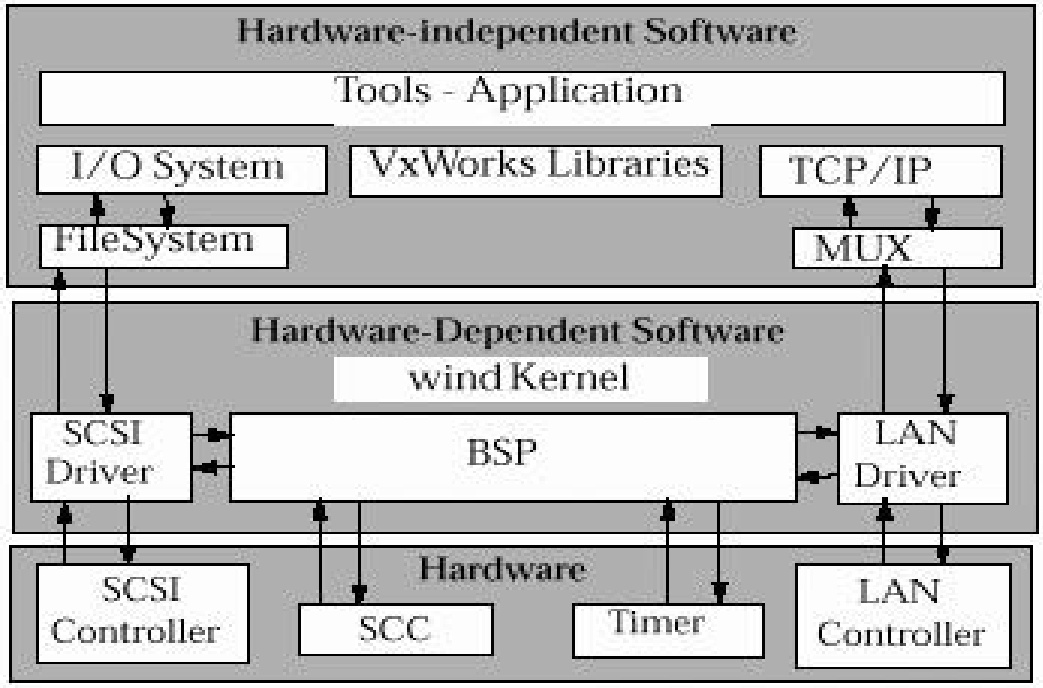
# What is a BSP?

- Provides VxWorks with primary interface to hardware environment.
- BSP Responsibilities:
  - Hardware initialization on power-up.
  - Support for VxWorks access to hardware drivers.
  - Integration of hardware-dependent and hardware-independent software in VxWorks.
- Components consist of:
  - Source, include, and make files.
  - Derived files.
  - Binary driver modules.
- May be validated to be WRS compliant.

# What a BSP is Not

- A BSP is not a hardware driver:
  - A hardware driver accesses hardware.
- Hardware drivers are classified as generic or BSP specific:
  - Generic drivers manage devices which can be moved from one target environment to another (e.g. LAN chip).
  - BSP drivers manage devices which are specific to the target environment (e.g. interrupt controller).
- BSP developer responsible for:
  - Complete support for BSP specific drivers.
  - Integration of generic device drivers.

# BSPs and VxWorks



# BSP Responsibility: Hardware Initialization

- VxWorks boot sequence specifics will vary with processors and hardware environments.
- Common initialization requirements:
  - Provide code at specific location in memory which processor will jump to on reset or power-up.
  - Set processor in a specific state.
  - Initialize memory and memory addressing.
  - Disable interrupts.
  - Pass control to additional bootstrapping code.
  - Load required VxWorks segment(s) into RAM.
  - Place hardware in quiescent state before initializing VxWorks kernel.

# BSP Responsibility: VxWorks Access To Hardware Drivers

- Some driver support is provided by BSP. Examples:
  - Driver defines ISR(s), but BSP connects ISR(s) to interrupt vector table.
  - BSP creates structures (objects) which are passed to driver for initialization.
  - Offset constants and access macros for hardware registers provided by BSP and used by driver.
- Provides portability for hardware driver code.
- Device configuration management:
  - Access to full range of device features (possibly at a later time).
  - Separate development/production configurations.

# BSP Responsibility: Integration of Hardware Dependent Software

- Provides code flexibility and portability:
  - Compile-time flexibility.
  - Run-time portability.
- Compile-time flexibility:
  - Uses preprocessor macros to customize system.
  - Provides ability to produce optimized modules without changing source code.
- Run-time portability:
  - Uses pointers to access routines.
  - Provides portability for compiled object modules.

# BSP Components: Primary Files

- Primary BSP files:
  - Source files.
  - Include files.
  - Make files.
- Source files:
  - Generic code is written in C. Architecture specific and performance optimized code is assembly.
- Include files:
  - All includes and definitions specific to a CPU board are localized in two files.
- Make file:
  - Controls building of all images.

# BSP Components: Derived Files

- Derived BSP files are created using:
  - Primary BSP files.
  - Driver source files.
  - Modules in VxWorks archive libraries.
- Derived BSP files are classified as:
  - Hardware initialization object modules.
  - VxWorks boot object modules.
  - VxWorks images.
  - VxWorks binary symbol table.
- A complete BSP port will generate all of these files.
- End users will recreate some of these files when configuring the system.

# BSP Development

- Development should occur in incremental steps:
  - First set up development environment (down-load path(s), debug strategies, etc.).
  - Write pre-kernel initialization code.
  - Optionally activate WDB agent and Tornado tools using polled serial or ethernet interface.
  - Start minimal VxWorks kernel adding support for a system clock, and install interrupts.
  - Complete BSP providing all necessary support for hardware environment (full network support etc.).
  - Clean-up, testing and documentation.
- Course material will be presented following this sequence as closely as possible.

# BSP Development - cont.

- Development time may be reduced by purchasing:
  - The BSP Developer's Kit.
  - Appropriate reference BSP.
- BSP Developers Kit provides:
  - A Validation Test Suite (VTS).
  - Template BSP (all architectures).
  - Template device drivers.
- Purchasing a reference BSP which most closely matches target environment:
  - Specific device drivers which are not part of reference BSP can also be purchased from WRS.
- Reference BSP obtained when Tornado is purchased.

# BSP Validation

- BSP validation:
  - WRS validated.
  - Non WRS validated.
- A WRS validated BSP:
  - Classified as Tornado Certified, and may be distributed displaying this information.
  - Contact WRS to obtain validation requirements.
- BSP validation uses a Validation Test Suite (VTS):
  - Automated test suite which runs on host and target to exercise BSP and report defects.
  - Included in BSP Developer's Kit.
  - VTS distribution includes source to allow extension.

# Overview

Integration Issues

1.2 VxWorks Boot Sequence

Tornado Directory Structure

Conventions and Validation

# VxWorks Image Types

- There are three classes of VxWorks images:
  - Loadable images.
  - ROM-based images - compressed/uncompressed.
  - ROM-Resident images.
- Loadable images are loaded into RAM by boot code.
  - Boot code is “burned” into ROM or Flash.
  - Boot code is a stand-alone VxWorks application.
- ROM-based images load themselves into RAM from ROM or Flash.
- ROM-resident images execute out of ROM or Flash.
  - Only the data segment of the VxWorks image is loaded into RAM.

# Some Terminology

- VxWorks boot image - A VxWorks image designed to load another VxWorks image containing application code (often referred to as “boot code”).
  - “Burned” into ROM or loaded into Flash.
  - May execute in ROM/Flash (ROM-resident).
  - May execute out of RAM.
- VxWorks image - A VxWorks image containing “end-user” code. Sub-types:
  - Loadable VxWorks image - VxWorks images loaded by VxWorks boot image.
  - VxWorks ROM image - VxWorks image “burned” into ROM or loaded into Flash. May execute in ROM/Flash (ROM-resident) or RAM.

# VxWorks Startup Sequence

- The sequence of events which occur at power-up are a function of the type of VxWorks image which will run.
- The initial phase of the start-up sequence is the same across all VxWorks image types.
- Processor is “jumped” to the entry point of boot-strap code in ROM or Flash. This code:
  - Disables interrupts (via the processor).
  - Initializes target memory.
  - Loads appropriate VxWorks image segments.
  - Jumps to code to place target in a quiet state.
- Various startup sequences are discussed next.

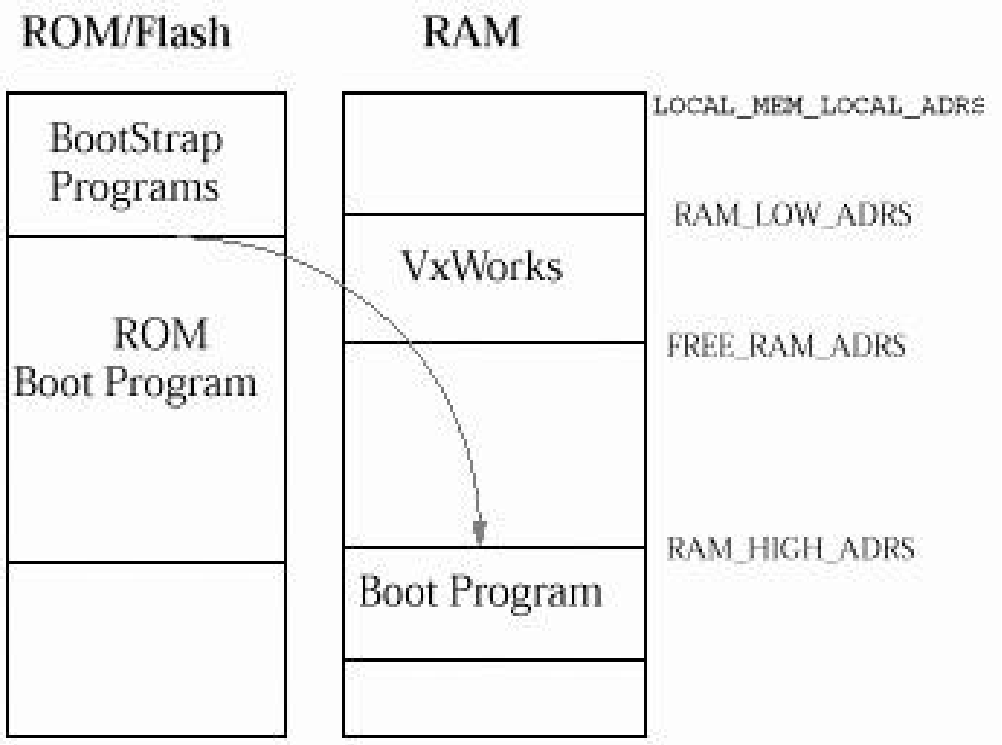
# Boot Sequence - Loadable VxWorks Image

- Bootstrap code executes and loads text and data segments of boot code (from ROM or Flash) into RAM.

  Scenarios are:

  - Boot code compressed - Decompression during copy
  - Boot code uncompressed - Copy
  - Boot code is ROM-resident - Copy data segment only
- Boot program executes and loads VxWorks image into RAM. Jumps to VxWorks load point.
- System initialization code statically linked into loaded VxWorks image executes and completes initialization.

# Loadable VxWorks Image



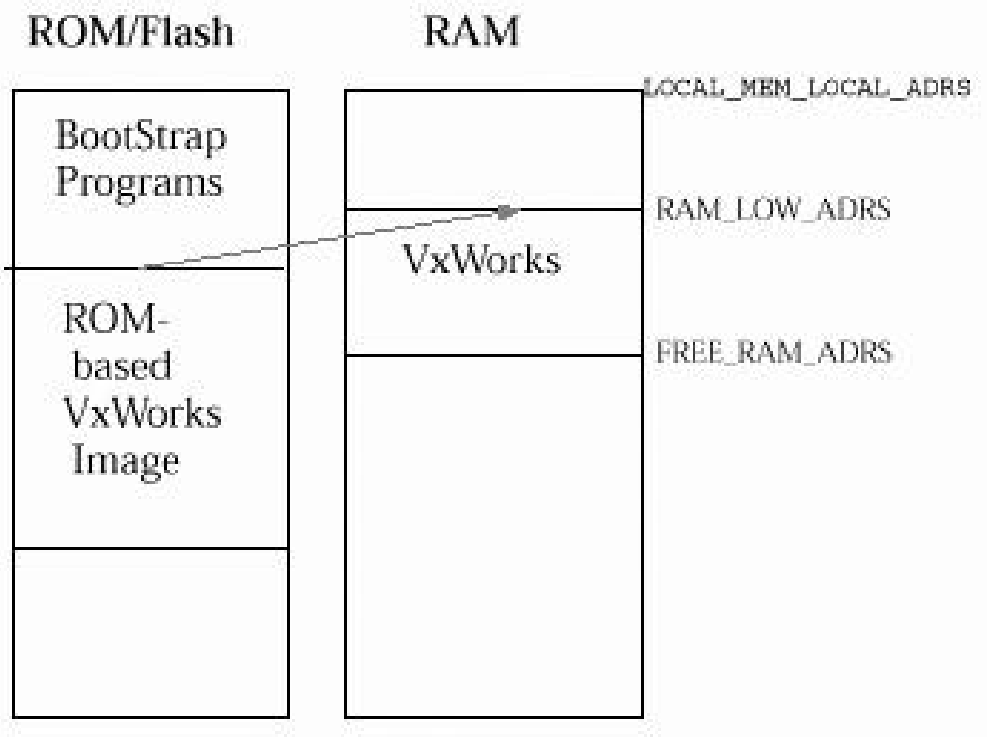
# Startup Sequence - ROM-based VxWorks Image

- Bootstrap code executes and loads text and data segments of VxWorks (from ROM or Flash) into RAM.

  Scenarios are:

  - VxWorks compressed - Decompression during copy
  - VxWorks uncompressed - Copy
- Control transfers to VxWorks initialization code in RAM.
- System initialization code statically linked into VxWorks image executes (in RAM) and completes initialization.

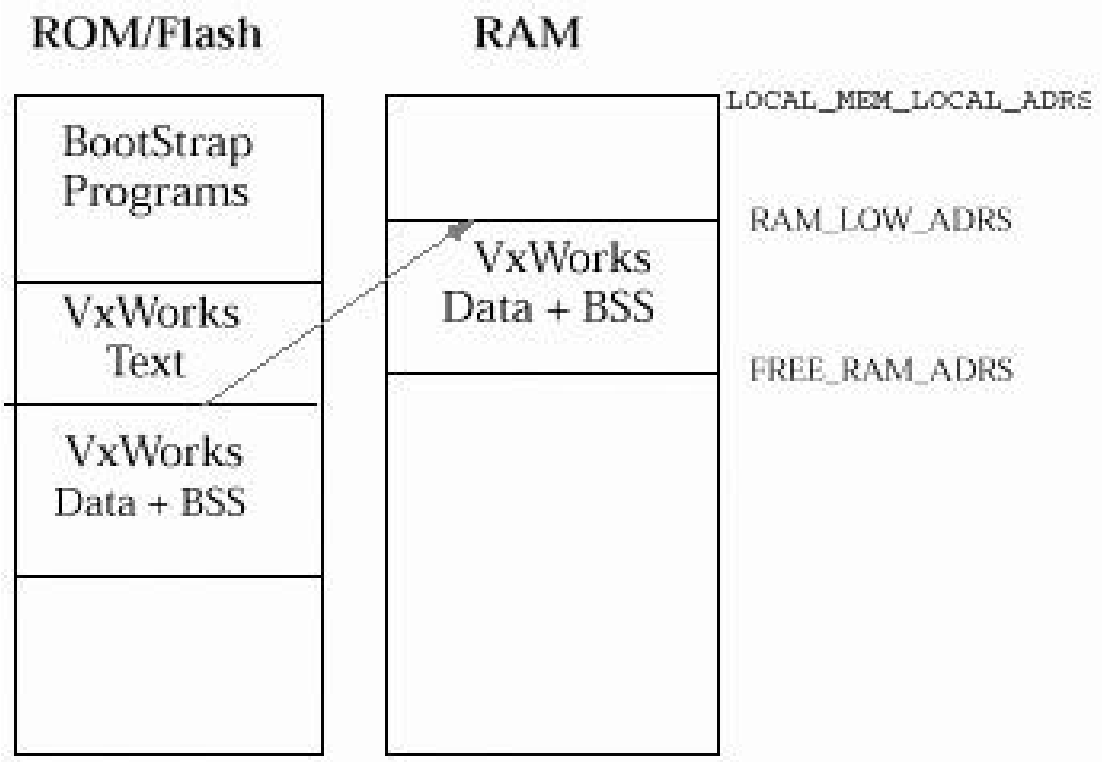
# ROM-based VxWorks Image



# Startup Sequence - ROM-resident VxWorks Image

- Bootstrap code executes and loads data segment of VxWorks image (from ROM or Flash) into RAM.
- Control branches to VxWorks initialization code in ROM or Flash.
- System initialization code statically linked into VxWorks image executes (in ROM or Flash) and completes initialization.

# ROM-resident VxWorks Image



# Startup Sequence - VxWorks Initialization

- After (“end-user”) VxWorks segment(s) are loaded into RAM, system initialization code statically linked into VxWorks image executes to complete the boot sequence.
- This code will:
  - Place hardware environment in a quiet state.
  - Initialize and start the wind kernel.
  - Spawn a task to complete system initialization.
- System initialization task will initialize support for end-user specified facilities, and start the end-user’s application.

# Overview

Integration Issues

VxWorks Boot Sequence

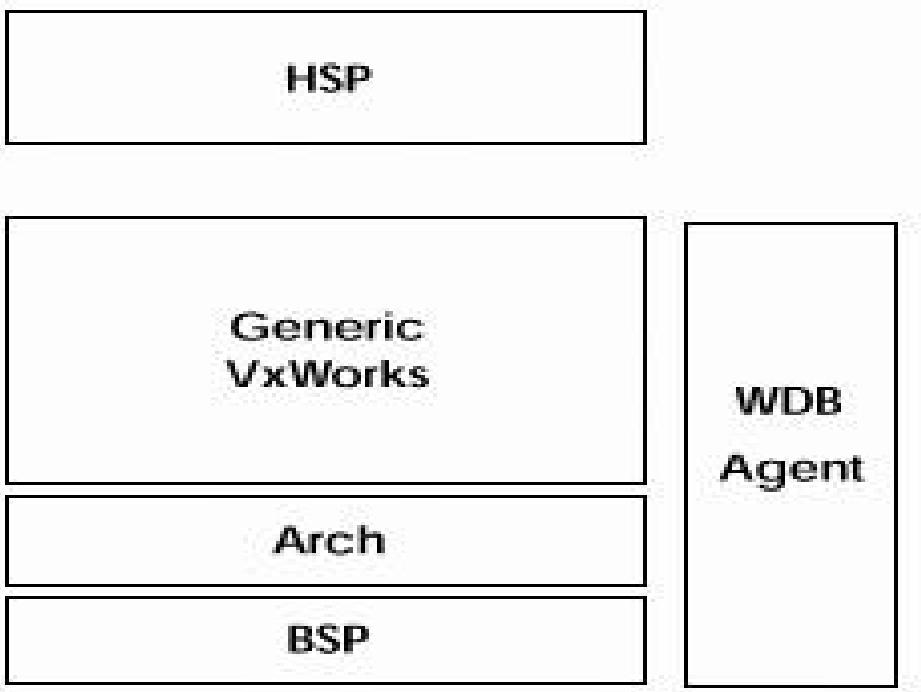
1.3 Tornado Directory Structure

Conventions and Validation

# Tornado Modularity

- Tornado is composed of a set of modular components.
- Modularity aids in portability, flexibility of use, and maintenance.
- Tornado modules are:
  - Host Support Package (HSP).
  - Generic (target independent) VxWorks.
  - Architecture Module.
  - Board Support Package.
  - Wind Debug Agent (WDB Agent).
- Tornado modules have been designed to minimize interdependence.

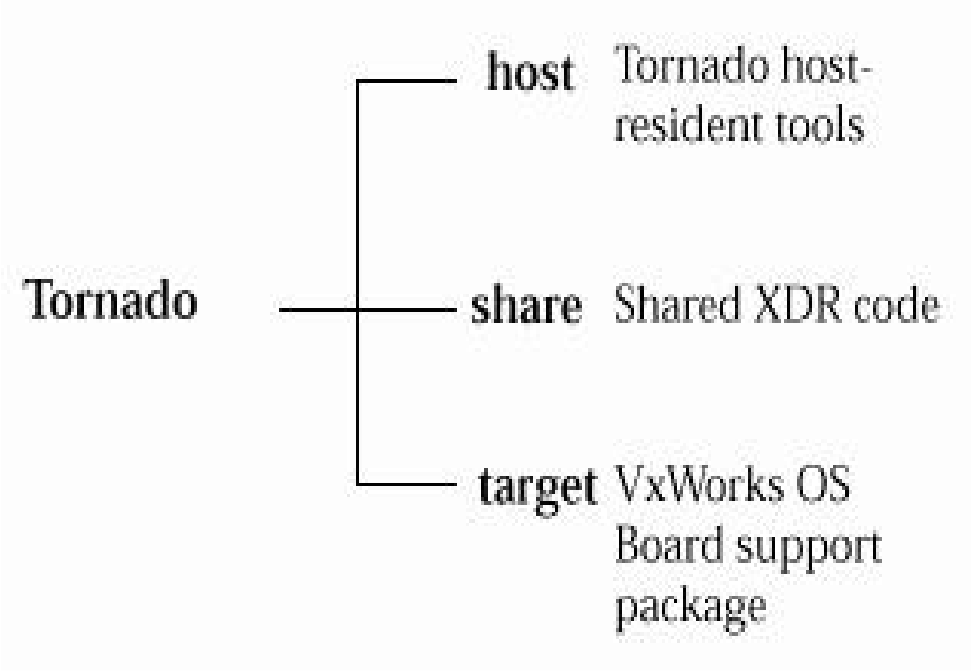
# Tornado Modularity



# Tornado Modularity and the Tornado Directory Tree

- Files which make up Tornado are organized to reflect Tornado's component modularity.
- At the highest level files (relevant for BSP development) are separated into host and target directories.
- All BSP specific files are in the target directory. However, many tools useful in developing a BSP are in the host directory.
- Files which will be modified in developing a BSP are in a configuration sub-directory of the target directory.

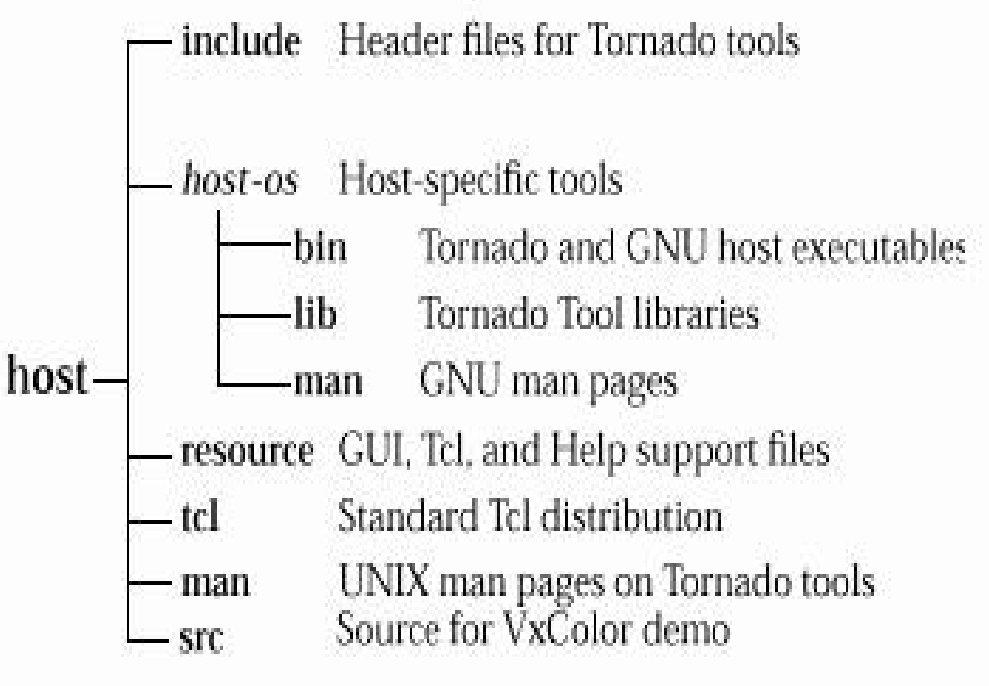
# Tornado Directory Tree



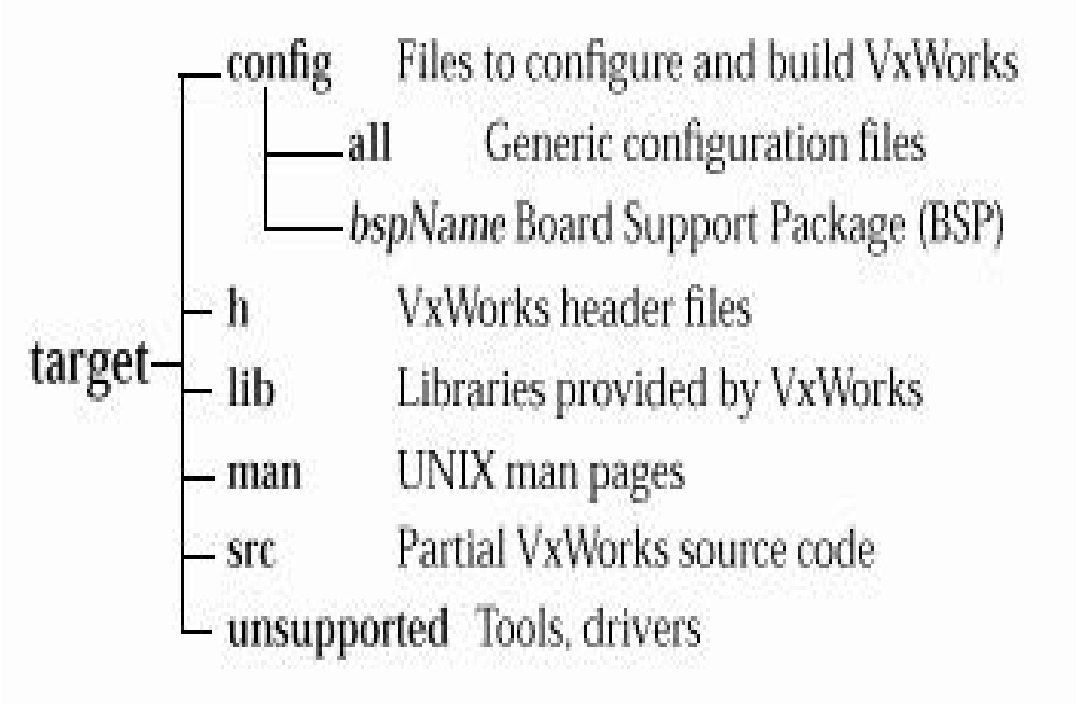



WindRiver SYSTEMS

# Host Directory Tree



# Target Directory Tree



# BSP Relevant Files

- All code which executes at power-up is in files within the config directory.
- BSP code vs. generic driver code:
  - Generic device driver code is designed to be usable with multiple BSPs (network drivers, serial drivers, etc.).
  - BSP (device driver) code is tightly coupled to the target environment and is not designed to be used with other BSPs.
- BSP specific code will always reside in ../<bspName>.
- Generic device driver code not supplied by WRS will reside in the <bspName> directory or a subdirectory of <bspName>.

# Overview

Integration Issues

VxWorks Boot Sequence

Tornado Directory Structure

1.4 Conventions and Validation

# BSP Conventions and Validation

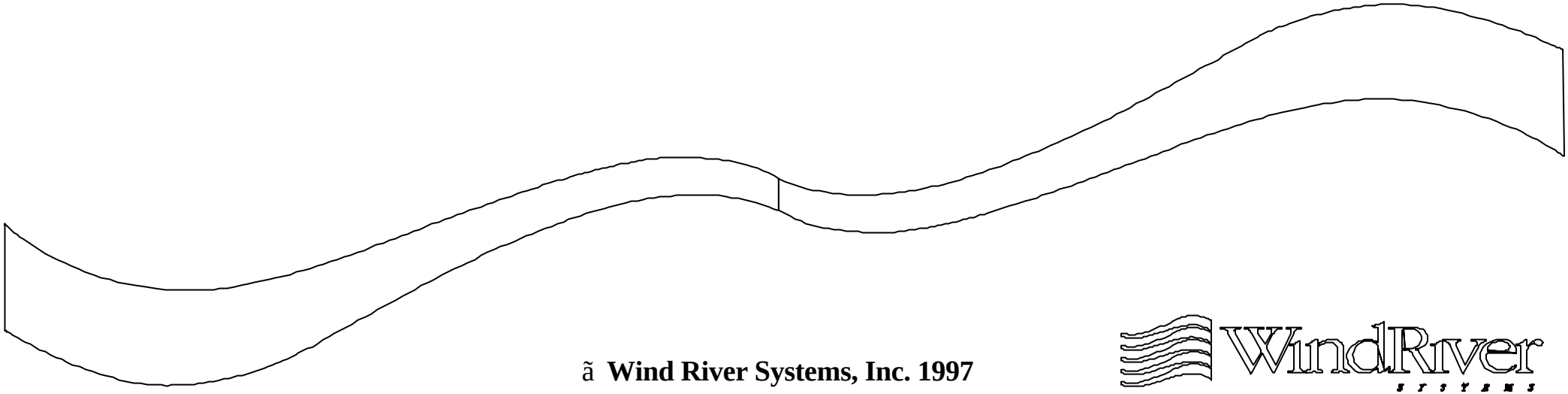
- BSP conventions and validation procedures are designed to help guarantee integrity of BSP.
- BSP conventions fall into categories:
  - Coding conventions
  - Documentation guidelines.
  - BSP packaging.
  - Driver guidelines.
- Validation test:
  - Package validation.
  - Installation test.
  - Functional test (VTS).
  - Code review process and WRS validation process.

# Summary

- BSP responsible for supporting system hardware environment:
  - Initialization of hardware environment.
  - VxWorks/application access to hardware drivers.
  - Hardware/software integration.
- Provides VxWorks with primary interface to hardware environment.
- Components consist of:
  - Source, include, and make files.
  - Derived files.
- May be validated to be WRS compliant.

# Chapter - 2

## System Hardware





WindRiver SYSTEMS

# System Hardware

2.1 Overview

Architecture Issues

Bus Systems

Memory

Devices

# Overview

- BSPs have responsibilities for all components in the hardware environment.
- Embedded hardware categories:
  - Architecture specific (caches, MMUs, interrupt controllers, and floating point hardware).
  - Bus specific (Bus controllers, and bus bridges).
  - Memory specific (memory controllers, and chips).
  - Devices (architecture/bus/memory independent).
- Support issues:
  - Initialization.
  - Access.

# System Hardware

Overview

2.2 Architecture Issues

Bus Systems

Memory

Devices

# BSPs and CPUs

- Libraries for managing CPUs are part of the Tornado architecture module.
- Some CPU specifics will be relevant for BSP development:
  - MMU Support
  - Cache Issues
  - Interrupt Handling
  - Floating-Point Support
- Many of these specifics will be important during the initialization phase of booting a VxWorks image.

# MMUs

- Memory Management Units control memory access for:
  - Allocating/de-allocating memory.
  - Resolving cache coherency issues.
  - Write protecting memory.
  - Virtual memory swapping.
  - Paging and segmentation.
  - Garbage collection.
- Requires RAM resident translation tables which map physical memory into a virtual memory address space.
- May be inappropriate for real-time applications due to latency increases and memory consumption.
- Unit is often on the same ASIC as the CPU.

# MMU Support

- Booting VxWorks:
  - MMU disabled until the wind kernel is activated.
  - Build translation tables.
  - Enable MMU.
- In VxWorks a task does not include a translation table as part of its context:
  - Tasks do not reside in virtual memory.
  - Default virtual memory maps are global and flat.
- MMU may be managed dynamically with:
  - Bundled MMU library.
  - Optional virtual memory management product (provides programmatic access to MMU).

# Caches

- Fast memory interface between CPU and main memory.
- Reduces read/write access time for CPU and local bus activity.
- Most modern processors support separate data and instruction caches.
- Cache is accessed in quantized units called cache lines.
- Cache modes:
  - Write through - Data written to cache by processor flushed to main memory.
  - Copyback - Writes only to cache, conserves processor bandwidth.

# Caches - continued

- Cache is located:
  - On the same ASIC as the CPU - L1 cache
  - External to the CPU ASIC - L2 cache
  - Backside L2 cache - External L1 cache (with special bus to processor).
- Some architectures provide cache management instructions, others bundle cache management with virtual memory support facilities.
- If cache is enabled, it is examined by the CPU for each memory access:
  - If data is there - cache hit.
  - If data is not there, access main memory- cache miss.

# Cache Issues

- Cache coherency:
  - Ideally cached information mirrors main memory.
  - Bus master or device with DMA support may update main memory without updating cache.
  - CPU may update cache without updating main memory, making information in main memory stale.
- To maintain cache coherency:
  - Cache needs to be flushed when updated by CPU.
  - Cache needs to be invalidated (and updated) when main memory is modified by bus master or DMA transfer.
  - Snooping circuitry.
- Copyback with snooping is fastest configuration.

# Cache Support

- Booting VxWorks:
  - Cache(s) disabled until the hardware environment is placed in a quiet state.
  - Invalidate and configure cache mode.
  - Enable cache(s) (before wind kernel is activated).
- Provide cache coherency for VxWorks:
  - Data cache / RAM - Bus master and DMA access to RAM.
  - Instruction cache / data cache - Loader, debugger, and ISR connection routines.
  - Shared cache lines - Multiple task access to cache.
- When MMU is enabled, cache management and mode control are supplied by an MMU library.

# Cache Support - continued

- Cache coherency is maintained:
  - Map off-board addresses as non-cacheable.
  - MMU enabled - Routine to allocate cache aligned memory marked as non-cacheable (returned cacheable).
  - MMU not enabled - Cache library support to flush and invalidate cache when necessary.
- Architecture may have more than one cache implementation. Hardware environment could then support multiple cache systems.
  - BSP must supply support for selecting the appropriate cache libraries.
  - Linker must include needed cache library modules.

# Interrupt Handling

- Hardware interrupt request and acknowledgment transactions are specific to:
  - Hardware requesting interrupt service.
  - CPU.
  - System bus protocol(s).
- Hardware device will:
  - Determine which interrupt service is requested.
  - Terminate device interrupt request upon IACK or device access.
- CPU will determine number of external interrupts and interrupt levels supported using external interrupt lines and/or interrupt controller.

# Interrupt Handling - continued

- IACK cycles are a function of local bus protocol:
  - Vectored - Automatic acknowledgment.
  - Autovectored - Acknowledgment done in software.
- CPU will determine which ISR to execute based on:
  - External interrupt line activated.
  - Interrupt select register of interrupt controller.
  - Interrupt select register on device.
  - Combinations, often requiring interrupt service de-multiplexing in software.
- CPU will reset interrupt status info after ISR completes:
  - Bits in a CPU status register.
  - Interrupt mask on external interrupt controller.

# Interrupt Handling Issues

- Designing interrupt scheme for hardware environment:
  - Interrupt priority levels.
  - Association of interrupts with devices and/or device service requests.
  - Number of ISRs - De-multiplex services in single ISR.
- Hardware support facilities:
  - CPU (and interrupt controller when present).
  - Device specific (enable/disable, IACKs, etc.).
- Interrupt service routines:
  - Latency.
  - Interrupt context and hardware negotiation.
  - Connecting.

# Interrupt Circuitry Guidelines

- To minimize latency in handling interrupts, design efficient interrupt circuitry.
- Choose devices which provide interrupt vectors, with different vectors for each requested service. If not possible, one vector for device.
- Ability to enable/disable each interrupt source separately.
  - Device interrupt control register.
  - Interrupt controller.
- Well documented and diagramed interrupt vector scheme.

# Interrupt Handling Support

- VxWorks uses an interrupt table:
  - ISRs must be connected to unique interrupt vectors after wind kernel is started.
  - Table contains addresses for interrupt handlers.
- VxWorks is interrupt aware, preventing the OS from providing blocking services at interrupt time.
- Device interrupts must be disabled before kernel is activated, and enabled after ISRs are connected.
- Architecture and device driver libraries:
  - Provide ISR code.
  - Provide hardware interrupt management code.

# Interrupt Handling Support -continued

- A BSP:
  - Disables device interrupts at system start-up.
  - Connects ISRs to interrupt table.
  - Supplies interrupt level/vector bindings to devices.
  - Supplies addresses and control values associated with hardware interrupt status and control registers.
  - Supplies routine which transfers control to boot code if an ISR throws an exception.
- BSP interrupt support should make device drivers as flexible as possible so they can be reused in hardware environments with different interrupt structures.

# Floating Point Support

- Support for floating-point operations:
  - Floating-point/math co-processor(s).
  - Software emulation.
- VxWorks contexts do not save/restore floating-point registers by default:
  - Tasks allow an optional context extension which will include floating point registers.
  - ISRs and exception handlers use bundled routine to programmatically manage floating-point registers.
- Architecture is responsible for floating-point support.
- Some BSPs will have configuration macros for FPU or software floating-point emulation.

# Additional Architecture Considerations

- Additional architecture issues which may impact a BSP:
  - Big-endian/little-endian byte ordering.
  - Processor specific initialization.
  - Register and memory alignment.
  - Addressing mode constraints.
  - TAS operations and external bus access.
- Any of these issues may require BSP configuration macros to be defined.

# System Hardware

Overview

Architecture Issues

2.3 Bus Systems

Memory

Devices

# Bus Systems

- Buses are classified relative to the processor:
  - Local bus (processor bus).
  - External buses (all others).
- Hardware environment may have:
  - Multiple external bus systems.
  - No external bus system.
- Bus system characteristics:
  - Cycles ((a)synchronous, multiplexed, etc.).
  - Arbitration (bus-locking, priority levels, etc.).
  - Data transfers (memory-mapped, I/O mapped, etc.).
  - Data properties (width, big/little endian, etc.).
  - Interrupt policies (generation, IACK, routing, etc.).

# Bus System Issues

- BSP developer will need to be aware of bus protocols:
  - Data transfer rates and formats.
  - Bus requirements for DMA transfers.
  - Interrupt protocols.
- For hardware environments with multiple buses, bridge chips may connect the busses, and provide an interface for inter-operability:
  - Data transfer.
  - Interrupt services.
- Bus system initialization:
  - Minimum initialization to boot VxWorks.
  - Complete initialization to support application.

# Bus System Support

- Bus systems and bus bridges have device drivers.
- Generic device drivers should be decoupled from bus specific issues.
- A BSP provides configuration and access support for:
  - Bus drivers.
  - Bus bridge drivers.
  - Generic drivers for bus resident devices.
- A BSP is responsible for bus initialization:
  - Bootstrap code identifies local bus speed, initializes local CPU bus, and necessary bus bridges.
  - Boot code/VxWorks completes initialization, paricularly support for external buses.

# System Hardware

Overview

Architecture Issues

Bus Systems

2.4 Memory

Devices

# Memory Types

- Memory types traditionally supported by embedded systems:
  - RAM - Random Access Memory
  - ROM - Read Only Memory
  - Non-Volatile memory - NVRAM and Flash.
- Memory types represent different technologies with different access characteristics, capabilities, and costs.
- Each memory type itself has multiple sub-types.
- Embedded systems will typically use some or all of these memory types.

# Memory Access

- Memory is accessed via uniquely addressed locations (which may be mapped to a separate address space).
- A memory controller provides hardware support allowing the CPU to access memory. Provides:
  - Address decode logic.
  - Timing control.
  - Memory bus interface support.
  - Control of memory mapped devices.
- To access memory mapped addresses in a different physical address space, the hardware environment may have a separate memory controller ASIC.
- Hardware environment may have multiple maps.

# Memory Access Issues

- Memory access issues:
  - Initialize memory hardware.
  - Provide support for software access.
- Initializing memory hardware:
  - Enable memory controller.
  - Enable memory chips.
  - Enable bridge/memory controller(s).
  - Check integrity of memory.
- Supporting software access:
  - Initialize dynamic memory management facilities.
  - Maintain integrity of system/application memory pools.

# Memory Access Support

- Memory access:
  - Hardware support.
  - Software support.
- Hardware support:
  - Initialization to load ROM code into RAM performed at power-up.
  - Remainder of memory hardware initialized by pre-kernel initialization code in VxWorks.
- Software support:
  - Management of virtual memory maps.
  - Initialize partition library after kernel is activated.
  - Enable and initialize MMU after kernel is activated.

# RAM

- Random Access Memory:
  - Dynamic RAM (DRAM)
  - Static RAM (SRAM)
- DRAM
  - Primary storage technology.
  - Store/hold capacitor technology.
  - Limited read/write cycle times.
  - Refresh cycles required.
- SRAM
  - Most often used for cache storage, often on CPU chip
  - Flip-flop technology, fast read/write cycle times.
  - No refresh cycles.

# RAM Support

- RAM configuration:
  - Main memory.
  - Cache memory.
- Main memory RAM configuration:
  - Zeroed at power-up for cold-boot to prevent parity errors.
  - Configured and enabled by ROM code at system power-up.
- Cache memory RAM configuration:
  - Usually disabled at power-up, enabled by VxWorks pre-kernel initialization code.
  - Provide cache management libraries.

# ROM

- Read Only Memory types:
  - Programmable ROM (PROM).
  - Erasable Programmable ROM (EPROM).
  - Electrically Erasable PROM (EEPROM).
- ROM properties:
  - Non-volatile. Modified using a ROM programmer.
  - Usually longer access times than DRAM/SRAM.
  - Memory controller interfaces ROM to CPU.
- Non-volatile property allows:
  - System boot code storage.
  - Hardware environment configured to jump to a ROM address at power-up.

# Flash

- Flash is non-volatile memory which can be modified programmatically.
- Used as a “silicon” hard disk:
  - System boot code storage.
  - Hardware environment configured to jump to a flash address at power-up.
  - Maintaining data integrity during power-outs.
- Access times slightly slower than DRAM but faster than ROM.
- Flash memory cells have a finite number of erase/ program cycles (~10,000 - 100,000).

# Flash vs. PROM

- Flash and ROM use similar memory cell technology:
  - Storage transistor employs transistor tunnelling.
  - No battery to provide non-volatility.
  - Access times are roughly the same.
- Flash technology requires lower voltage to erase/ program than PROM.
- Flash power supply unit allows flash to be modified without being removed from hardware environment.
  - Contents may be modified over a network interface.
  - Contents may be modified by application code.
- Many flash chips support configurable write protection.

# Flash/PROM Support

- Flash/PROM support facilities:
  - At system start-up.
  - For application code.
- System start-up support:
  - Code executing out of flash/PROM in Tornado tree.
  - Makefile support to build boot code.
  - Flash file system to load VxWorks from flash.
  - Minimum capacity PROM.
- Application code support:
  - Code (driver/file system) to uniformly access flash.
  - Large capacity PROM for ROM-resident and un-compressed VxWorks/boot code images.

# NVRAM

- Non-volatile RAM:
  - Non-volatility usually provided by battery.
  - May be implemented using CMOS RAM, battery-backed SRAM, or flash.
- Units may contain a programmable time-of-day (TOD) clock:
  - TOD information is stored in NVRAM.
- Used to store boot parameters for VxWorks image:
  - VxWorks boot parameters may use up 255 bytes.
- If a hardware environment does not have NVRAM, boot parameters are statically linked into boot code.

# System Hardware

Overview

Architecture Issues

Bus Systems

Memory

2.5 Devices

# Embedded System Devices

- Generic devices are independent of architecture, buses, and memory hardware.
- Typical devices:
  - Timers.
  - Serial Communication Controllers (SCC).
  - Network interfaces.
  - SCSI controllers.
  - Custom ASICs (DSP chips, etc.)
- Devices should support:
  - Read/write access.
  - Any mandatory access timing requirements.

# Timers

- Hardware timers are used for:
  - System clock interrupt (mandatory).
  - Auxiliary clock for high speed polling.
  - Timestamp for WindView.
  - TOD clocks (“Real Time Clocks” - RTCs).
- Timers operate in one of three modes:
  - Periodic interrupt.
  - One-shot interrupt.
  - Timestamp.
- Timer use dictated by mode(s) supported by hardware.
- Timers are initialized after the kernel is activated.

# Timers - cont.

- The three modes for timer operation are:
  - Periodic interrupt - Counts up/down to programmed terminal count, generates interrupt, resets counter.
  - One-shot interrupt - Counts up/down to programmed terminal count, generates interrupt, disables counter.
  - Timestamp - Counts up/down to maximum count, generates interrupt to log counter rollover, restarts count. Unlike periodic interrupt, counter is polled to obtain high-fidelity timestamp, interrupt is only used to mark counter rollover.
- TOD clocks will also contain date information.

# Serial Communication Controllers

- SCCs used as:
  - Download/debug path during BSP/application development.
  - Communication channel for application.
- Support for interrupt and polled mode operation:
  - System level debugging (pre/post kernel).
  - Task level debugging (post kernel).
  - Dual level debugging (post kernel).
- In polled mode, serial interface can be used for system level debugging prior to kernel activation.
- For pre-kernel system level debugging serial port is accessed by WDB agent.

# Network Interfaces

- Provides support for:
  - Application development using Tornado.
  - Distributed applications.
- VxWorks supports two network stacks:
  - 4.3 BSD TCP/IP stack.
  - SENS - Scalable Enhanced Network Stack.
- SENS supports:
  - 4.4 BSD TCP/IP stack.
  - A proprietary MUX interface between the link and protocol layers.
  - END - Enhanced Network driver which decouples network driver from network protocols.

# SCSI Controllers

- Used to provide access to hard disks, tape drives, etc.:
  - For booting VxWorks.
  - Application data storage/retrieval.
- VxWorks supports SCSI-2 systems. Support consist of:
  - Architecture independent libraries.
  - Architecture specific controller driver.
  - Board specific device initialization code.
- Devices accessed by application through:
  - File system (DOS or RAW) for block devices.
  - Tape file system for sequential devices.
  - Customized SCSI commands for unsupported device classes.

# Summary

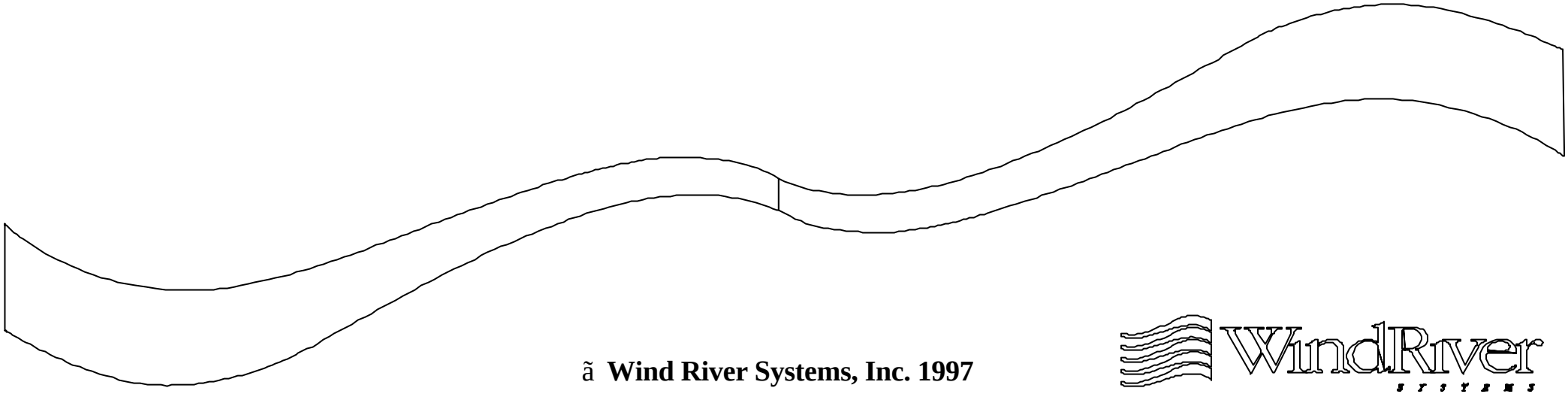
- A BSP will provide support for several hardware categories.
- Architecture specific:
  - Caches.
  - MMUs.
  - Interrupt controllers.
  - Floating point hardware.
- Bus specific:
  - Bus controllers.
  - Bus bridges.

# Summary

- Memory specific:
  - Memory controllers.
  - Memory chips.
- Devices:
  - Timers.
  - Serial Communication Controllers.
  - Network interfaces.
  - SCSI controllers.
  - Custom ASICs.
- Support issues involve initialization and hardware access by application code.

# Chapter - 3

## BSP Development Issues





WindRiver SYSTEMS

# BSP Development Issues

3.1 Development Cycle Overview

Development Environment

Strategies For Getting Started

# Development Cycle Overview

- Development of BSP proceeds in stages, with each stage depending on developments from previous stages:
  - Obtain appropriate reference BSP and template code.
  - Prepare the development environment.
  - Write the VxWorks pre-kernel initialization code.
  - Optionally supply support for Tornado access using a polled serial driver.
  - Once kernel is activated, connect system interrupts.
  - Enable the system clock.
  - Complete BSP by supporting desired features.
  - Test and document BSP following WRS standards.
- Details will depend on development environment and desired BSP features.

# Reference BSP

- Choosing an appropriate reference BSP involves obtaining maximum coverage for desired target features, and reducing development time.
- Must obtain Tornado package for correct development platform and target architecture.
- Give priority to matching system features over generic (BSP independent) devices:
  - Local and external bus support.
  - Target bridges and controllers.
- May be able to obtain drivers for generic devices (e.g. serial, LAN, SCSI, etc.) separately from WRS or third party.

# BSP Template

- BSP template files are obtained when BSP developers purchase the BSP developer's kit.
  - Includes all architectures.
- This should be the starting point for all code development:
  - Do not "hack" reference BSP files, unless development consists of simply adding generic drivers.
  - Examine and analyze reference BSP code, but build BSP using template files.
- The template BSP will compile, but most optional features have been disabled.

# Development Environment

- Primary components of development environment:
  - Technique for downloading code to target.
  - Technique(s) for testing and debugging code.
- Appropriate and available development tools are dependent on development stage:
  - Early pre-kernel initialization phase requires BSP developer to define development environment.
  - Post-kernel initialization phase will have access to Tornado tools.
- Early pre-kernel development environment needs to provide download path and mechanism to jump to code entry points and execute code successfully.

# Pre-Kernel Development Environment

- Common download paths:
  - Target vendor's debug ROM.
  - ROM emulator.
  - In-Circuit Emulator (ICE).
- Common debug tools:
  - Target vendor's debug ROM.
  - ICE.
  - Logic analyzer.
  - Target features such as LEDs.
  - Tornado toolset.

# BSP Development Issues

Development Cycle Overview

3.2 Development Environment

Strategies For Getting Started

# Development Environment Requirements

- Developer must define pre-kernel development environment, often the same tool will provide download mechanism and debug facilities:
  - ICE.
  - Target vendor's debug ROM (if available).
- Sometimes a combination of tools will be required.
  - For a ROM emulator, debug tools will need to be supplied separately.
- A download path will be needed for some debug tools:
  - Logic analyzer.
  - Target status indicators.

# In-Circuit Emulation

- Processor replaced by probe with cables connected to emulation unit:
  - Specific architecture is emulated.
  - Emulation unit contains copy of processor being emulated.
- Technique allows access to processor bus to monitor or inject signals into system via:
  - Emulator processor.
  - Emulator circuitry.
- Emulator processor allows code to execute at full CPU speed providing timing information and allowing race conditions to be caught.

# In-Circuit Emulation - cont.

- Emulator circuitry provides debug capabilities:
  - Source level debugging.
  - Halt emulation (breakpoint) on events not supported by software debuggers (trap code insertion).
- ICE debug extended feature examples:
  - Breakpoints in ROM or RAM.
  - Hardware breakpoints (watched points).
  - Breakpoints on “don’t care” addresses and data (e.g. 0x0247XXXX).
  - Breakpoints on certain bus events.
  - Breakpoints on processor cycles (interrupt acknowledge, cache writeback, etc.)
  - Code fetches with specific data patterns.

# In-Circuit Emulators

- Primary issues:
  - Must have correct emulator for target architecture.
  - Cost.
- ICE systems usually come with support for code downloads via serial or network interface:
  - To RAM (loadable VxWorks image).
  - To Flash (ROM-based and ROM-resident images).
- Most emulators contain memory for code storage:
  - Allows re-mapping of target memory regions to emulator (e.g. mapping ROM to emulator RAM).
  - Off-loads system software from host, reducing host load while using ICE.

# In-Circuit Emulators - cont.

- Often ICE will bundle enhanced facilities:
  - Logic state analyzer.
  - Logic timing analyzer.
  - Pulse and pattern generator.
  - Multiple trace capability.
  - User-friendly GUI.
  - Integrated debugger.
  - Expansion busses for additional hardware access.
  - Event ID and isolation capability.
  - Real-time trace and filter.
- If target server backend is developed for an ICE, it can be used with Tornado tools.

# Tornado / ICE Integration

- Two ICEs which can be used with Tornado:
  - visionICE
  - TRACE 32 ICE
- Both products provide ethernet download capability, support for several popular architectures, and many enhanced features for real-time systems.
- Target server backend for these ICEs have been developed:
  - Provides access to Tornado tools prior to kernel activation.
- Contact WRS for more information concerning ICEs which can be integrated with Tornado.

# ROM Emulators

- ROM emulator gives host machine access to target via ROM socket:
  - Target ROM chip is removed and emulator pods are plugged into (standard) ROM socket.
  - Pods are connected to emulator, which in turn connects to host machine via serial or network link.
- ROM emulators not architecture dependent, can be used with any target which has ROM socket.
- ROM emulator unit contains memory which replaces target ROM:
  - Code downloaded from host to emulator memory.
  - Emulator memory appears as part of target memory.

# ROM Emulators - cont.

- VxWorks can be executed:
  - Out of emulator memory (ROM-resident image).
  - Out of target memory (ROM-based image).
- ROM emulator provides download path, but not debug tools. Debug tools obtained via:
  - Logic analyzer.
  - Debug agent linked with software loaded to emulator memory.
- Most ROM emulators provide communication path and protocol to pass debug messages from target to host.
- If target server backend is developed for a ROM emulator it can be used with Tornado tools.

# NetROM

- One ROM emulator which can be used with Tornado tools is NetROM:
  - Converts target with ROM socket to network node.
  - 1 MB of emulation memory.
  - Ethernet downloads via tftp or TCP.
  - 2 KB of dual ported RAM for debug communication.
- NetROM target server backend option bundled with Tornado:
  - Provides access to Tornado tools prior to kernel activation.
- See Tornado User's Guide for more information on configuring and using NetROM.

# Vendor Debug ROM

- Vendor native debug ROM (when available) comes with target board.
- Development software burned into ROM on target:
  - Dynamic loader.
  - Supported download path.
  - Ability to jump to address and begin execution.
  - Debug tools.
  - Diagnostics.
- Target environment may have jumper allowing board to boot from ROM or some non-volatile RAM.
- After development can be replaced by ROM containing VxWorks image.

# Vendor Debug ROM - cont.

- Downloading code:
  - Serial or network interface.
  - Download path to RAM (VxWorks loadable image).
  - Often download paths to Flash (ROM-based or
- ROM-resident images).
- Debug ROM code supplies device driver for communication interface port (serial or network).
- Example debug/diagnostic facilities (support varies):
  - Set breakpoints and step code.
  - Examine and modify memory.
  - Set environmental parameters (bus clock speed, etc.).
  - Self-test to verify integrity of target.

# Logic Analyzer

- Provides tool to monitor processor's address, data, control, and status lines:
  - Connects to processor pins via multiple probes.
  - Contains memory for data capture.
- Provides pre-kernel development services such as:
  - Tracing on clock cycle or bus pattern triggers to monitor device responses.
  - Locate hardware interrupt sources.
  - Monitor address access to check memory mapping.
  - Processor state evolution to monitor code execution sequences.
- Bundled with many ICEs.

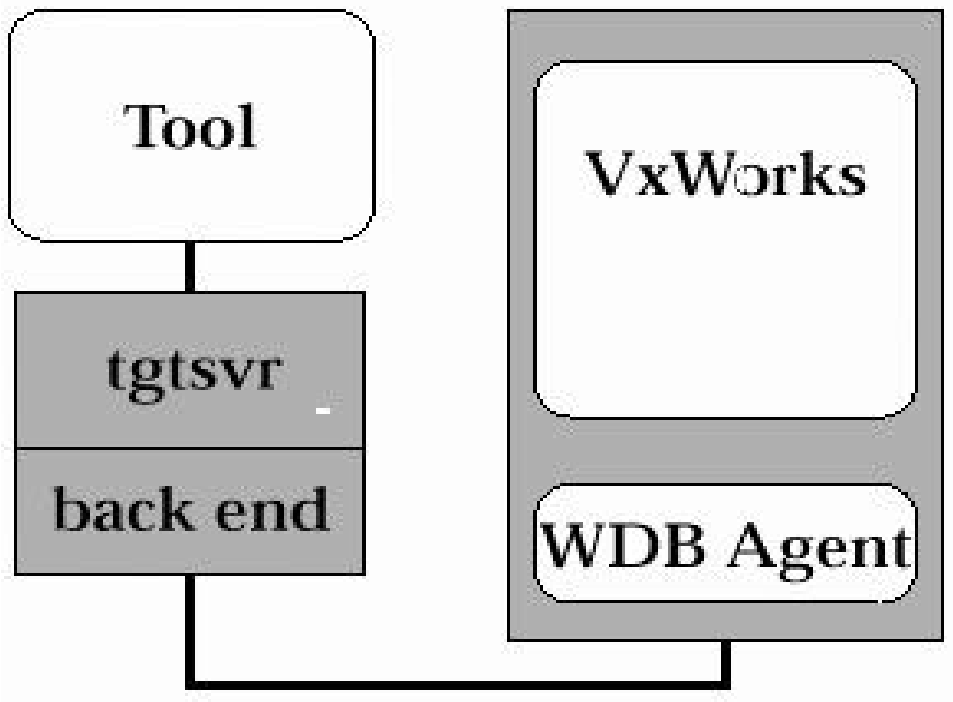
# Target Feature Debug Tools

- Native target environment may provide several features which can be used as diagnostic/debug tools:
  - Accessible memory mapped LEDs.
  - Local or off board persistent memory.
  - Serial port.
- Write library to manage target feature to be used as diagnostic/debug tool:
  - Flash LEDs N times to mark code sequence events.
  - Write state information to persistent memory for analysis after system reboot.
  - Polled serial driver to send character(s) to host to mark code sequence events.

# WDB Agent And Tornado Tools

- Wind DeBug (WDB) agent is statically linked with VxWorks image. Provides advantages over traditional ROM monitor:
  - One set of initialization code for agent and OS.
  - Reduced size as agent shares libraries with OS.
  - Provides access to full featured debug environment,
  - Can run in system or task mode (post-kernel).
- Requires target server backend support to access Tornado tools. For pre-kernel development:
  - NetROM (bundled).
  - Serial line (bundled).
  - Some ICEs (vendor or third party supplied).

# Host - Target Interaction



# BSP Development Issues

Development Cycle Overview

Development Environment

3.3 Strategies For Getting Started

# Getting Started

- Obtain appropriate reference BSP, Tornado BSP Developer's Kit for VxWorks, and device drivers.
- Once WRS software is provided but before starting development code:
  - Determine development environment and obtain appropriate hardware.
  - Configure target hardware based on documentation and development strategy.
  - Choose appropriate development image type.
  - Identify and configure required software tools.
- Once Tornado access has been achieved, development environment and strategy may be changed.

# Development Environment

- Will need to determine:
  - Development tools.
  - Target download path.
- For ICE or ROM emulator:
  - Download tools provided.
  - Determine to execute image in emulator memory or target memory (ROM-resident or ROM-based development image).
  - Will need to provide debug tools for ROM emulator.
- For ROM monitor:
  - Download and debug tools supplied.
  - Development image may be loadable or ROM.

# Development Environment - cont.

- For vendor debug ROM:
  - Download and debug tools provided.
  - Load image to Flash (ROM-resident or ROM-based).
  - Load image to RAM (can use loadable VxWorks image if ROM debug code initializes RAM).
- For tools without download facilities (logic analyzer and target features):
  - Use with tools suppling download path.
  - Burn development image into ROM.
- Target environment may need hardware configuration - boot from Flash or ROM, external bus status, activation of download port, ROM swap, etc.

# Loadable vs. ROM images

- Difference between ROM and loadable image is loadable image does not initialize RAM.
- If development image is a loadable image RAM will need to be initialized by some other facility:
  - If RAM initialization facility is not be present in production environment, start with ROM image.
- If loadable image is only option for development environment, and RAM initialization facility will not be available in production environment:
  - Will need alternative environment to develop and test VxWorks RAM initialization code.
- Course will assume development image is ROM image.

# Code Preparation

- Development modules must be fully linked:
  - Set appropriate cross-compiler, assembler, and linker flags.
  - Linker must produce image with appropriate format for download methodology.
- WRS provides cross-development tools:
  - Appropriate cross-compiler, assembler, and linker.
  - Architecture specific nmX command to dump object module symbol table.
  - Architecture specific objdumpX command to compare host assembly code with assembly code loaded to target.
  - Routines to convert various object modules to hex (with appropriate S-record format).

# Code Preparation - cont.

- VxWorks image code is built using makefiles. Macros provided to specify build details:
  - Cross-compiler and linker flags.
  - Module components linked with image.
  - Image types.
- For code not linked into a VxWorks image, developer will need to create makefile(s) to produce fully linked object module(s) which can be downloaded:
  - Use WRS supplied cross-development tools.
  - May need different cross-compiler and linker options during different phases of development.
  - Build libraries an place them outside of Tornado directory structure.

# Code Preparation Example

- A routine to establish initial contact with target:
  - Blinks target LED on 68k board.
  - Loaded via serial line from UNIX host.
  - Target RAM previously initialized.
- Routine in file talkToTarget.c:

```
/* talkToTarget.c */
/* Routines to blink lights on target board. */
#define WAIT_CNT 100000 /* Loop count for delay. */
/* Forward declarations. */
void lightBlink (void);
void wait (void);
```

# Code Preparation Example - cont.

```
void lightBlink (void)
        {
        int * pBrdLight= (int *) 0xFFF40060;
        for(;;)
              {
              *pBrdLight = 0x43000000; /* Turn light on. */
              wait ();
              *pBrdLight = 0x41000000; /* Turn light off. */
              wait ();
              }
      return;
      }
void wait (void)
      {
      int i;
for(i=0;i<WAIT_CNT;i++)
;
```

# Code Preparation Example - cont.

- First talkToTarget.c is cross-compiled using WRS cross-compiler:

  cc68k -c -DCPU=MC68040 - FLAGS talkToTarget.c
- Produces relocatable object module talkToTarget.o.

  Symbol table dumped with nm68k:

  00000000 T _lightBlink

  00000028 T _wait

  00000000 t gcc2_compiled.
- Convert to fully linked module with desired download address (0x100000) using WRS link-load tool:

  ld68k -Ttext 100000 talkToTarget.o

# Code Preparation Example - cont.

- Produces a.out file, dumping symbol table with nm68k:

```
00000000 A __DYNAMIC
00120000 B __end
00120000 D _edata
00120000 B _end
00120000 T _etext
00100000 T _lightBlink
00100028 T _wait
00100000 t gcc2_compiled.
00100000 t talkToTarget.o
```

- Convert to S-record format for serial line download using WRS tool:

```
hex a.out > talkToTarget.hex
```

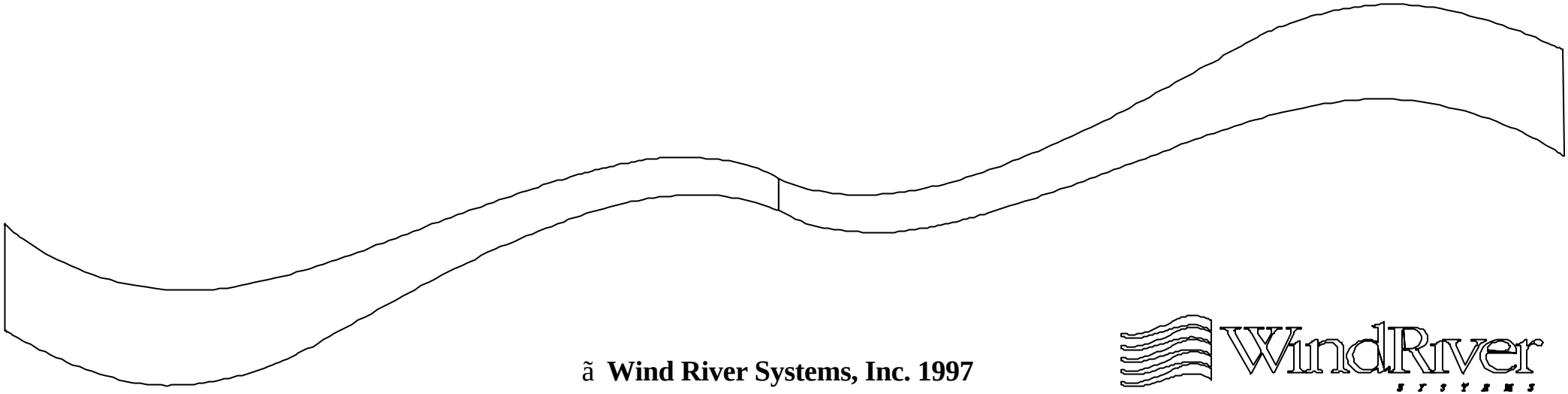
- This file may now be downloaded to target RAM.

# Summary

- First stage of BSP development involves:
  - Obtaining appropriate reference BSP, template code, and device drivers.
  - Define pre-kernel development environment.
- Pre-kernel development environment will specify:
  - Target download mechanism.
  - Diagnostic and debug tools.
  - Appropriate VxWorks development image.
- WRS cross-development tools will be useful for preparing code not statically linked with VxWorks.
- Post-kernel development may employ a different development environment using Tornado tools.

# Chapter - 4

## Pre-Kernel Initialization Overview





WindRiver SYSTEMS

# Pre-Kernel Initialization Overview

4.1 Pre-Kernel Initialization

Sequence

BSP Files

Building VxWorks Images

# VxWorks Boot and ROM Images

- As outlined in the Overview chapter, on power-up bootstrap code executes:
  - The processor is “jumped” to a routine romInit() in ROM/Flash.
  - romInit() resets processor, initializes memory system, and performs any other required hardware initialization.
  - romInit() branches to romStart() which loads the ROM image (boot or VxWorks) into RAM.
  - Processor jumps to pre-kernel initialization code statically linked into VxWorks image (usrInit()).
- For ROM-resident images:
  - romStart() only loads data segment of image into RAM.

# VxWorks Loadable Images

- After “end-user” VxWorks image is loaded in RAM, the processor is “jumped” to the VxWorks load address.
  - A routine sysInit() resides at this address. This routine resets the processor, and performs other hardware initialization if required.
  - sysInit() branches to usrInit() which completes the pre-kernel initialization.
- Both of these routines are statically linked into a loadable VxWorks image.
- sysInit() is functionally similar to romInit(). The difference is that romInit() initializes memory and sysInit() does not. (DRAM and memory controller usually need to be initialized once.)

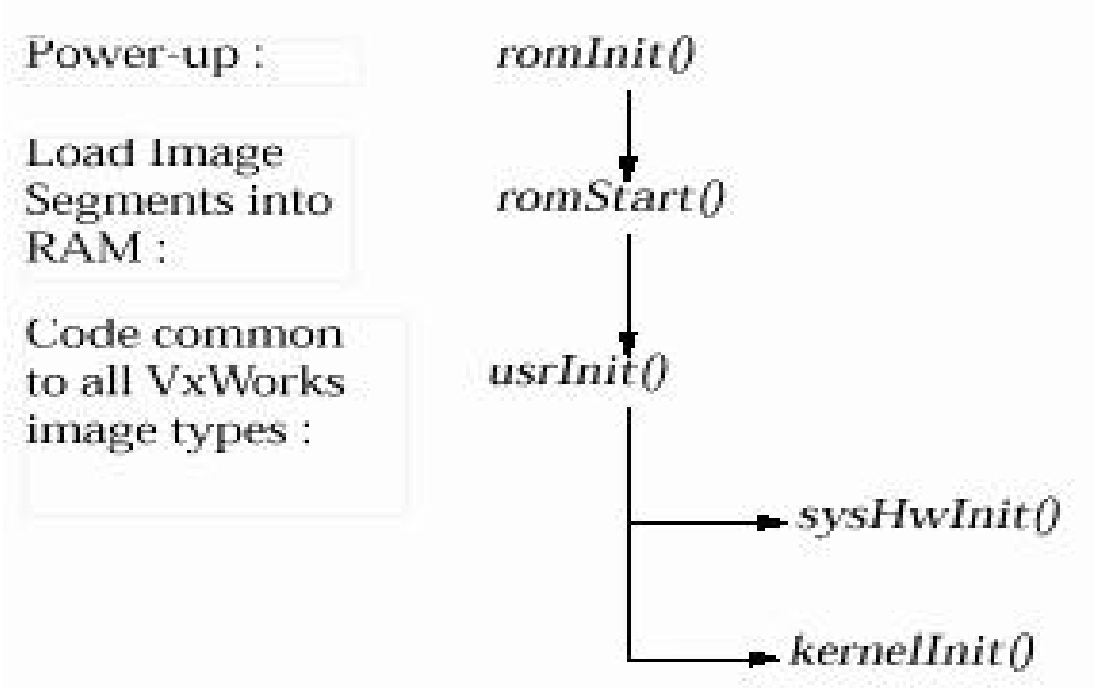
# Generic Pre-Kernel Initialization

- The routine usrInit() is a generic routine:
  - Statically linked into all VxWorks image types.
  - Calls routine which activates VxWorks kernel.
- Primary responsibility to place hardware in a quiet state so kernel can be activated:
  - Disable all hardware interrupts.
  - Initialize hardware to a known quiescent state.
- romInit()/sysInit() perform the minimal initialization necessary to allow usrInit() to execute.
- “VxWorks” provides the remainder of the hardware initialization via usrInit().

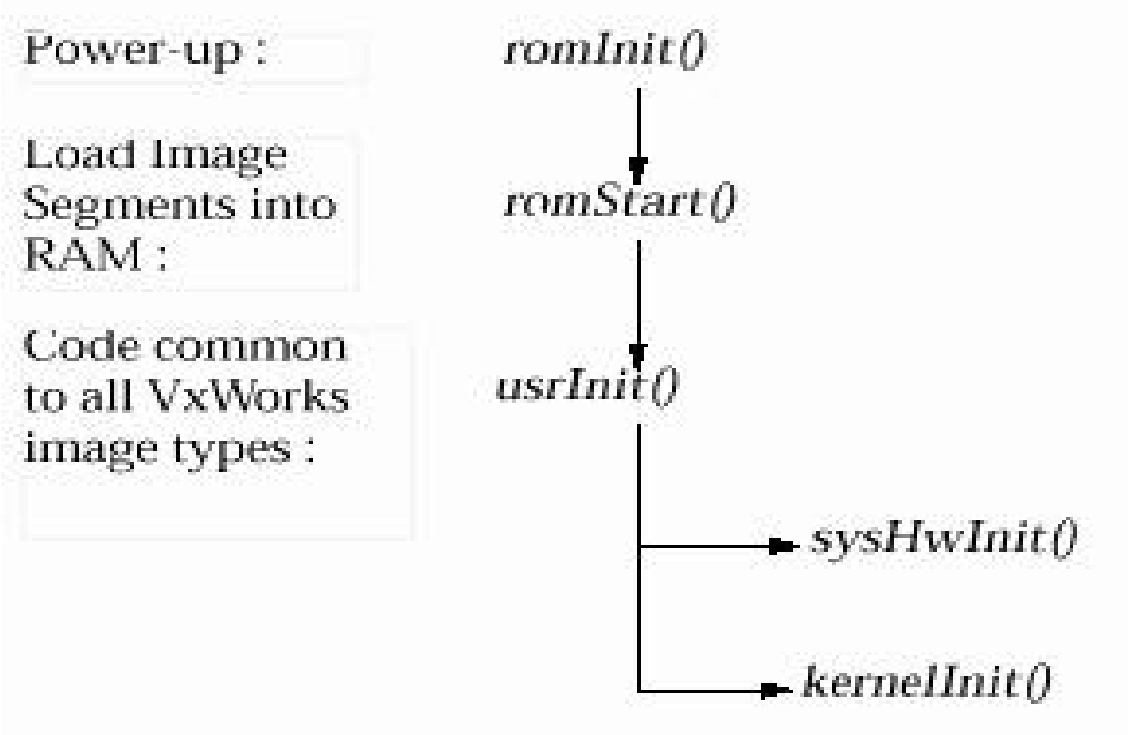
# Generic Pre-Kernel Initialization - cont.

- The routine which places hardware in the initial quiet state prior to activating the kernel is sysHwInit().
- The routine which activates the VxWorks kernel is kernelInit().
- kernelInit() activates the multitasking environment and spawns a task which:
  - Installs drivers and creates devices.
  - Initializes VxWorks library facilities.
  - Calls application start-up code.

# Pre-Kernel Initialization Sequence
# VxWorks Boot and ROM Images



# Pre-Kernel Initialization Sequence Loadable VxWorks Image



# What Executes Where?

- romInit() always executes in ROM/Flash, and jumps to romStart() in ROM/Flash.
- romStart() always begins execution in ROM/Flash.
  - ROM-resident images load data segment to RAM and continue to execute in ROM.
  - Images which are not ROM-resident copy start-up code to a RAM address, and then jump to that RAM address.
- usrInit() executes out of RAM except for ROM-resident boot and ROM-resident VxWorks images.
- sysInit() always executes out of RAM.

# Pre-Kernel Initialization Overview

Pre-Kernel Initialization Sequence

4.2 BSP Files

Building VxWorks Images

# BSP Files - Overview

- BSP component files are located in:
  - ../config/<bspName>
- Directories containing BSP related files are:
  - ../config/all
  - ../h/make
- Related directories providing support for device drivers will also be referenced during BSP development:
  - ../src/drv
  - ../h/drv
- BSP development will focus on BSP files. All files which must be customized are in ../config/<bspName>.

# BSP Files - Overview

- Files in the ../config/all directory are delivered as part of the Tornado distribution. These files should not be modified.
- Files in the ../config/<bspName> directory are not delivered as part of the Tornado distribution:
  - BSP is a separate product (sales and installation).
- All VxWorks image type builds are controlled by the Makefile in the ../config/<bspName> directory.
- Support makefiles containing rules and dependencies are located in ../h/make. Modifications will be needed in the primary Makefile as part of BSP development.

# BSP Files - Overview

- The components of a BSP are:
  - Source files.
  - Include files.
  - Makefiles.
  - Derived files.
  - Document files.
- BSP, and BSP related files will be presented as follows:
  - Source files in the ../config directories.
  - Include files in the ../config directories.
  - Makefiles in ../config/<bspName> and ../h/make.
  - Derived files in ../config/<bspName>.
  - Document files in ../config/<bspName>.

# BSP Related Files - Source Files

- BSP related source files in ../config/all:
  - bootConfig.c - The primary initialization file for VxWorks boot images. Contains the routine usrInit().
  - usrConfig.c - The primary initialization file for VxWorks images. Contains the routine usrInit().
  - bootInit.c - Consists of the routine romStart() and two support routines which romStart() calls.
  - version.c - Used for each VxWorks build. Provides VxWorks version ID as well as date and time of build using the ANSI _DATE_ and _TIME_ macros.
  - dataSegPad.c - Insures that text and data segments of VxWorks images do not share a MMU page when using VxVMI. Not used in the pre-kernel initialization sequence.

# BSP Files - Source Files

- BSP source files in ../config/<bspName>:
  - romInit.s - Assembly language source for romInit().
  - sysALib.s - Assembly language source for sysInit().
  - sysLib.c - File containing routines providing board-level access in a generic fashion. It #includes all driver modules (or causes them to be linked into VxWorks images). Contains the routines sysHwInit(), sysHwInit2(), as well as many other routines which must be provided as part of a BSP. Primary BSP source file.

# BSP Files - Source Files

- Optional BSP source files in ../config/<bspName>:
  - sysSerial.c - File containing routines to provide initialization for serial I/O devices. Some routines in this file are called via sysHwInit() as part of pre-kernel initialization. Not required if serial I/O interface is not used.
  - sysScsi.c - File containing SCSI configuration routines. These routines execute after the kernel is activated. Not required if SCSI support not needed.
  - sysNet.c - File containing routines for initialization and configuration of network interface devices. Not required if LAN interface is not present.
- If the BSP requires any unique drivers they should be located in ../config/<bspName> (not ../src/drv).

# BSP Files - Include Files

- BSP related include files in ../config/all:
  - configAll.h - This file establishes the default configuration for VxWorks. It should not be modified.
- BSP include files in ../config/<bspName>:
  - config.h - This file is used to modify VxWorks and BSP hardware configurations. This file will be modified as BSP development evolves.
  - <bsp>.h - This file contains fixed hardware values (hardware addresses, hardware interrupt levels, etc.). Should not be modified unless hardware environment is modified.

# BSP Files - Makefiles

- BSP makefile in ../config/<bspName>:
  - Makefile - Controls building of all VxWorks image types. Probably will need to be modified as part of the pre-kernel code development.
- BSP related sub-makefiles in ../h/make:
  - rules.bsp - Contains the rules for building the various VxWorks image types, as well as the rules for BSP object modules which are used in VxWorks builds.
  - defs.bsp - Contains definitions of BSP build control macros for compilation and linking.
- Other sub-makefiles in ../h/make control host and architecture specific build parameters.

# BSP Files - Derived Files

- BSP derived files in ../config/<bspName>:
  - VxWorks images.
  - VxWorks boot images.
  - Object modules generated when source files in the ../ config directories are compiled (bootConfig.o, usrConfig.o, bootInit.o, romInit.o, sysALib.o, and sysLib.o).
  - depend.<bspName> - Make will generate this dependencies file when a VxWorks build is done.
  - C files and associated object module files for a target resident symbol table (symTbl.c and symTbl.o) and C++ constructors/destructors (ctdt.c and ctdt.o).
- VxWorks image types will be discussed in greater detail later.

# BSP Files - Document Files

- BSP documentation files in ../config/<bspName>:
  - target.nr - File containing board specific information necessary to execute VxWorks image types. File is nroff format and divided into sections involving supported/unsupported features, instructions for using boot ROMs, summary of hardware devices, target environment layout, and description of board jumpers.
  - target.txt - ASCII version of target.nr.
  - README - File contains BSP release record. This information is a version number/revision number.
- A BSP version number identifies the BSP's generation, a BSP revision number incrementally identifies a release within a BSP generation.

# Pre-Kernel Initialization Overview

Pre-Kernel Initialization Sequence

BSP Files

4.3 Building VxWorks Images

# VxWorks Builds

- VxWorks builds are controlled by the Makefile in the ../config/<bspName> directory.
- The type of VxWorks image which will be built is specified by the object type name specified. These “target” names appear in the file ../h/make/rules.bsp.
- VxWorks image types can be divided into:
  - VxWorks images - Loadable, ROMable, and ROM-resident.
  - VxWorks boot images - ROMable and ROM-resident.
- All ROMable (non-ROM-resident) images can be sub-divided as compressed or uncompressed.

# VxWorks Image Types

- The build rules for VxWorks will produce images for the following object type names:
  - vxWorks - Loadable binary VxWorks image. (Also builds a separate vxWorks.sym symbol table file).
  - vxWorks_rom - Uncompressed ROMable binary VxWorks image.
  - vxWorks.st - Stand-alone loadable binary VxWorks image. Symbol table linked in.
  - vxWorks.st_rom - Compressed ROMable version of vxWorks.st.
  - vxWorks.res_rom - Uncompressed ROM-resident version of vxWorks.st.
  - vxWorks.res_rom_nosym - ROM-resident version of vxWorks.st without symbol table.

# VxWorks Image Types - continued

  - bootrom - Compressed ROMable binary VxWorks boot image.
  - bootrom_uncmp - Uncompressed rommable binary VxWorks boot image.
  - bootrom_res - ROM-resident binary VxWorks boot image.
- S-record formatted versions for all rommable and ROM-resident images can be built by using the object type name show here and adding a ".hex" extension.
- Note, uncompressed ROMable images may require extra capacity ROMs.
- All ROMable and ROM-resident images can be configured to be "burned" into Flash or PROM.

# The VxWorks Makefile

- The file Makefile in the ../config/<bspName> directory controls all VxWorks builds. It contains:
  - Required BSP specific macros.
  - Additional (non-required) support macros.
  - Includes of support makefiles in ../h/make.
- Some macros defined in Makefile are also defined in ../config/<bspName>/config.h. Definitions must be identical.
- Compilation rules, linking rules and support macro definitions for building images are in ../h/make.

# VxWorks Makefile Macros

- The BSP developer is responsible for defining the following required BSP specific macros in ../config/ <bspName>/Makefile:
  - CPU - Target CPU.
  - TOOL - Host tool chain (e.g., gnu)
  - TGT_DIR - By default set to $(WIND_BASE)/target.
  - TARGET_DIR - BSP directory name.
  - VENDOR - Board manufacturer’s name
  - BOARD - Name of board.
  - ROM_TEXT_ADRS - Boot ROM entry address in hexadecimal. Will be a Flash address if processor is “jumped” to Flash on power-up.

# VxWorks Makefile Macros -continued

- ROM_SIZE - Size of ROM area in hexadecimal.
- RAM_LOW_ADRS - Address at which non-ROM-resident application VxWorks images begin. (It is also the initial load address for compressed VxWorks boot images. This will be discussed in the next chapter.)
- RAM_HIGH_ADRS - Destination address for non-ROM- resident VxWorks boot images. (Also initial load address for non-ROM-resident compressed VxWorks ROM images.)
- HEX_FLAGS - Architecture specific flags for building S-record formatted versions of images.
- MACH_EXTRA - Extra machine-dependent files to be linked. Initialize as empty declaration.

# VxWorks Makefile Macros -continued

- The following macros must be identically defined in ../h/<bspName>/config.h:
  - ROM_TEXT_ADRS
  - ROM_SIZE
  - RAM_LOW_ADRS
  - RAM_HIGH_ADRS
- There may also be some architecture specific macros required in the Makefile file. (Example, the i960 CPU needs to know where to link the Initial Boot Record.)
- Hexadecimal addresses used in macro definitions should not have a leading 0x in Makefile.

# VxWorks Makefile Macros For Customized Builds

- Additional (non-required) macros to customize VxWorks builds fall into two categories:
  - Those used by application developers.
  - Those used by BSP developers.
- Macros for application developers contain ADDED in their name. These macros allow the user to specify compile time options.
- Macros for BSP developers contain EXTRA in their name. These macros allow additional object modules to be compiled and linked with VxWorks.

# The Makefile and Sub-makefiles

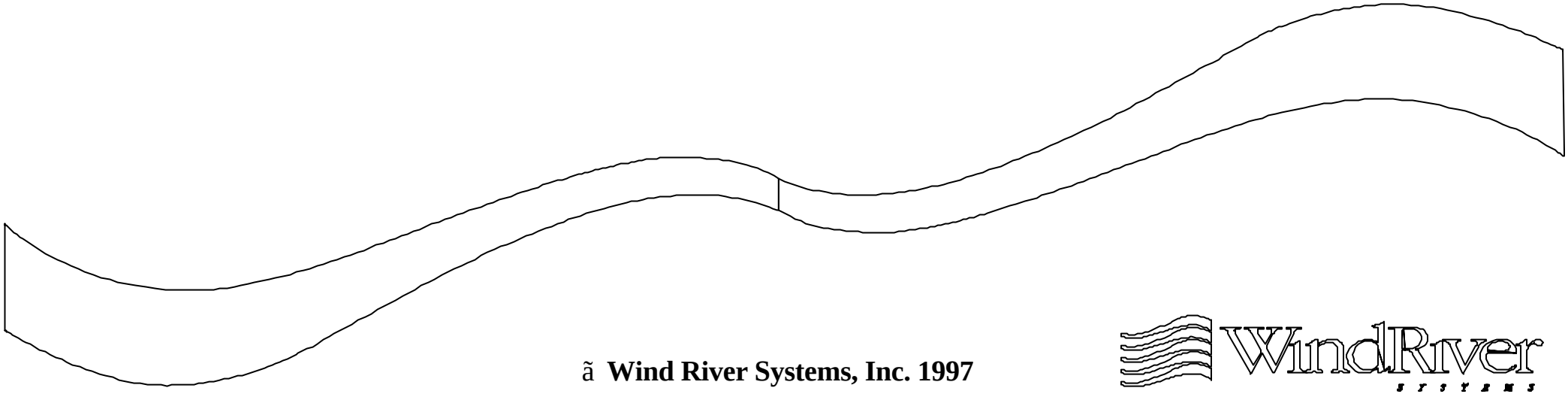
- The Makefile file contains includes of sub-makefiles containing definitions and rules necessary for VxWorks builds. These sub-makefiles are in ../h/make:
  - defs.bsp - File containing default make definitions. These definitions can be customized in Makefile.
  - make.$(CPU)$(TOOL) - File contains CPU specific macros for a specific tool chain.
  - defs.$(WIND_HOST_TYPE) - File where host specific macros are defined.
  - rules.bsp - File containing rules for VxWorks builds.
  - rules.$(WIND_HOST_TYPE) - Files contains host specific build rules.

# Summary

- Pre-kernel initialization code is responsible for placing the hardware environment in a state which allows the VxWorks kernel to be activated.
- Pre-kernel initialization code is specific to boot strategy and statically linked into the appropriate VxWorks image type:
  - Loadable image - Contains application code.
  - ROM image - May contain application or boot code.
  - ROM-resident image - May contain application or boot code.
- VxWorks builds controlled by Makefile file which uses sub-makefiles containing make definitions and rules.

# Chapter - 5

## Pre - Kernel
## Initialization - Boot
## Specific Code





WindRiver SYSTEMS

# Pre-Kernel Initialization - Boot Specific Code

5.1 Boot Specific vs. Generic Code

romInit.s : romInit()

PIC and VxWorks

bootInit.c : romStart()

sysALib.s : sysInit()

# VxWorks Image Types and Generic Code

- Details of pre-kernel initialization depend on VxWorks image type characteristics:
  - ROM image - Boot or “end-user” image.
    a. compressed
    b. uncompressed
  - ROM-resident image - Boot or “end-user” image.
  - Loadable image - “End-user” image.
- Generic pre-kernel code common to all image types is usrInit() and the routines it calls. These will be discussed in the next chapter.

# Boot Specific Pre-kernel Initialization Code

- VxWorks image type specific code:
  - romInit()
  - romStart()
  - sysInit()
- romInit() and romStart() execute for all images “burned” into ROM.
- sysInit() only executes for all loadable VxWorks images.
- romInit() and sysInit() are similar routines except romInit() initializes memory and sysInit() does not (this is done by romInit() in the boot image).

# Choice of First Image

- Which type of image is developed first depends on download path:
  - Download to RAM - Use vxWorks.
  - Download to ROM - Use vxWorks_rom or vxWorks.res_rom_nosym.
- The initial image should not be compressed or contain a symbol table. These features can be added later.
- The first image for a download path to ROM:
  - vxWorks_rom - Allows software breakpoints for code which executes in RAM.
  - vxWorks.res_rom_nosym - Provides a smaller RAM footprint (and possibly reduced start-up time).

# Pre-Kernel Initialization - Boot Specific Code

Boot Specific vs. Generic Code

5.2 romInit.s : romInit()

PIC and VxWorks

bootInit.c : romStart()

sysALib.s : sysInit()

# romInit() Basics

- First code to execute on power-up. Entry point for all VxWorks ROM images.
- Performs minimum required setup to execute romStart(). The remainder of hardware initialization is performed by generic pre-kernel code.
- Routine must:
  - Mask processor interrupts and reset processor.
  - Initialize the memory system.
  - Initialize stack pointer and other registers to begin executing romStart() and passing the boot type.
- Routine is written is assembly language and resides in file romInit.s.

# Architecture vs. BSP Specific Issues

- Much of what romInit() needs to do is processor specific and can be copied from the reference BSP:
  - Masking processor interrupts.
  - Initializing on-processor caches.
  - Initializing the stack pointer.
- Non-processor specific initialization involves DRAM and will be specific to the hardware environment.
  - Wait states.
  - Refresh rates.
  - Chip selects (bridge/bus/memory controllers, etc.)
  - Disabling of L2 caches (if necessary).

# Cold vs. Warm Boots

- Two boot types:
  - Cold boot - Power-up of hardware environment.
  - Warm boot - Call to reboot(), ^X, or exception at interrupt level. The routine which passes control to the ROM monitor is sysToMonitor() in sysLib.c.
- Where romInit() begins execution is a function of the boot type:
  - Cold boot - Execution begins at the entry point romInit(). Boot type is forced to be BOOT_COLD.
  - Warm boot - Execution begins at romInit() plus a small offset (usually 4 bytes). Boot type is saved.
- Boot type (cold/warm) is stored in an architecture specific register and passed to romStart().

# Stack Pointer Initialization

- Macro which configures beginning of stack is STACK_ADRS in configAll.h.
- For ROM-resident images the stack will begin:
  - In RAM at the start of the VxWorks data segment for stacks which grow down.
  - In RAM at the start of the VxWorks data segment less the size of the stack for stacks which grow up.
- For non-ROM-resident images the stack will begin:
  - In RAM at the start of the text segment of the VxWorks image for stacks which grow down.
  - In RAM at the start of the text segment of the VxWorks image less the size of the stack for stacks which grow up.

# romInit() - PIC

- romInit(), which runs in ROM/Flash, must be written as Position Independent Code (PIC) to support the various boot strategies for VxWorks images.
- PIC code is program counter (PC) relative.
- If a ROM address cannot be made program counter relative then it must be recomputed by:
  - Subtracting _romInit (The entry point for romInit().)
  - Adding ROM_TEXT_ADRS (Boot ROM/Flash entry address. Where ROM code is “burned”.)
- This algorithm ensures that a ROM address is expressed relative to the PC value for romInit() regardless of the address assigned to romInit() by the compiler/linker.

# romInit() - Some do's and don'ts

- Perform minimum necessary initialization. Leave most hardware initialization to generic routine sysHwInit().
- Do not call out to other modules or routines:
  - May cause linking problems for compressed images.
  - Call outs to C routines may use absolute not PC relative addressing.
- Make sure romInit() is the first routine in romInit.s.
- Start with romInit() from reference BSP.
- Make sure macros in Makefile and config.h are correct:
  - ROM_TEXT_ADRS
  - ROM_SIZE

# Pre-Kernel Initialization - Boot Specific Code

Boot Specific vs. Generic Code

romInit.s : romInit()

5.3 PIC and VxWorks

bootInit.c : romStart()

sysALib.s : sysInit()

# PIC and VxWorks Builds

- romInit() which executes in ROM needs to be PIC to support various VxWorks image types.
- This is because romInit() is linked into all non-loadable VxWorks images, all of which do not execute in ROM.
- To understand how romInit() (as well as other routines) are linked into VxWorks images the build rules in ../h/ make/rules.bsp must be examined.
- Examine the link instructions for vxWorks_rom:
  - Uncompressed rommable binary image.
  - Begins execution in ROM.
  - Transfers execution to RAM in romStart().

# vxWorks_rom Build

- The link instructions for the target vxWorks_rom are:

```
vxWorks_rom : ....
....
$(LD) $(LDFLAGS) $(LD_PARTIAL_FLAGS) \
    -o ctmp.o usrConfig.o \
    $(MACH_DEP) version.o $(LIBS)
....
$(LD) $(LDFLAGS) -e $(ROM_ENTRY) $(LD_LOW_FLAGS) \
    -o $@ romInit.o bootInit_uncmp.o dataSegPad.o \
    ctmp.o ctdt.o
....
```

- Dots, "...." indicate missing code. Missing code consists of compilation and file management instructions.

# vxWorks_rom Build Link Flags

- The first link builds most of the image and places it in the relocatable module ctmp.o.
- The second link builds the final fully linked relocatable image vxWorks_rom (this is target name for “$@”).
- The link flags are:
  - LD = ldppc (make.$(CPU)$(TOOL) )
  - LDFLAGS = -X -N (defs.bsp)
  - LD_PARTIAL_FLAGS = -X -r (defs.bsp)
- Note, it is the “-r” flag which produces a partially linked relocatable module (ctmp.o) for the first link, but not for the second which produces vxWorks_rom.

# vxWorks_rom Build - ctmp.o

- The temporary relocatable module ctmp.o uses the macro expansions (defs.bsp):
  - MACH_DEP = sysALib.o sysLib.o ..
  - LIBS = ../lib/lib$(CPU)$(TOOL)vx.a
- Pre-kernel initialization code in ctmp.o:
  - sysInit() - in sysALib.o
  - sysHwInit() - in sysLib.o
  - usrInit() - in usrConfig.o
- Remainder of the ctmp.o contains modules from the appropriate VxWorks library archive and version.o.
- The remainder of the pre-kernel initialization code is included in the second link.

# vxWorks_rom Image

- The final link includes the remainder of the kernel initialization code:
  - romInit() - in romInit.o
  - romStart() - in bootInit_unmcp.o
- The final vxWorks_rom image uses the macro expansions (defs.bsp):
  - ROM_ENTRY = _romInit
  - LD_LOW_FLAGS = -Ttext $(RAM_LOW_ADRS)
- The ROM_ENTRY macro for the “-e” flag insures that romInit() will be the execution entry point.
- The LD_LOW_FLAGS produces text addresses starting in RAM not ROM!

# vxWorks_rom Image and PIC

- The vxWorks_rom image is “burned” into ROM (or Flash) at the address ROM_TEXT_ADRS (Makefile). This is how ROM_ENTRY = ROM_TEXT_ADRS.
- Execution will begin in ROM even though the linker has assigned RAM addresses to the text for romInit().
- This is the reason why romInit() must be PIC code for this image.
- For addresses which are not program counter relative, address recalculations are usually done with a macro:

```
#define ROM_OFFSET(x)((x) - _romInit+ROM_TEXT_ADRS)
```

# vxWorks.res_rom_nosym

- Next consider a ROM-resident image:

```
vxWorks.res_rom_nosym: ....
$(LD) -o $@ $(LDFLAGS) $(ROM_LDFLAGS) \
    -e $(ROM_ENTRY) $(RES_LOW_FLAGS) \
    romInit_res.o bootInit_res.o \
    ctmp.o ctdt.o
```

- The linker flag RES_LOW_FLAGS expands to:

```
-Ttext $(ROM_TEXT_ADRS) -Tdata $(RAM_LOW_ADRS)
```

  - Text is assigned ROM addresses.
  - Data is assigned RAM addresses.
- romInit() does not need to be PIC for this image, or any ROM-resident image.

# Pre-Kernel Initialization - Boot Specific Code

Boot Specific vs. Generic Code

romInit.s : romInit()

PIC and VxWorks

5.4 bootInit.c : romStart()

sysALib.s : sysInit()

# romStart() Basics

- Jumped to by romInit() which places the start type on the stack for romStart().
- Performs necessary code relocation, uncompression, and RAM initialization for ROM images:
  - Copies appropriate ROM image segments to RAM.
  - Clears portions of RAM not being used (cold boot).
  - Performs uncompression if required.
  - Passes control to generic pre-kernel code (usrInit()).
- Code is written in C and resides in ../all/bootInit.c. Portion which executes in ROM should be PIC.
- Do not modify code. Functionality is controlled by configuration macros.

# Code Relocation

- Which image segments are relocated by romStart():
  - ROM images - Text and data.
  - ROM-resident images - Data.
- Final RAM destination for ROM image segments:
  - Uncompressed VxWorks boot - RAM_HIGH_ADRS
  - Compressed VxWorks boot - RAM_HIGH_ADRS
  - Uncompressed VxWorks - RAM_LOW_ADRS
  - Compressed VxWorks - RAM_LOW_ADRS
  - ROM-resident VxWorks boot - RAM_HIGH_ADRS
  - ROM-resident VxWorks - RAM_LOW_ADRS
- For uncompressed ROM images there is only one relocation from ROM to the final RAM destination.

# Compressed Image Relocations

- Compressed ROM images contain an uncompressed component and a compressed component.
- romInit.s, and bootInit.c code is in the uncompressed component. Remainder of image is compressed.
- There are two relocations for these images:
  - First relocation moves uncompressed component from ROM to RAM.
  - Second relocation occurs when compressed component of image is uncompressed and relocated from ROM to final destination in RAM.
- Second relocation is performed by romStart() in RAM. romStart() is moved into RAM during first relocation.

# Compressed Binary Images

- Compressed VxWorks boot images:
  - Relocate uncompressed component of ROM image to RAM location RAM_LOW_ADRS.
  - Uncompression code executes in RAM uncompressing and relocating VxWorks boot image from ROM to RAM location RAM_HIGH_ADRS.
  - Execution jumps to usrInit().
- Compressed VxWorks application images:
  - Relocate uncompressed component of ROM image to RAM location RAM_HIGH_ADRS.
  - Uncompression code executes in RAM uncompressing and relocating VxWorks image from ROM to RAM location RAM_LOW_ADRS.
  - Execution jumps to usrInit().

# Clearing Memory For Cold Boots

- For cold boots RAM is re-initialized.
- Mitigates parity error generation for some hardware (usually activated by read access without initialization).
- After romStart() relocates ROM image to RAM (but prior to uncompression if necessary) it clears all memory not filled with text and data.
- Memory is re-initialized to zero a long at a time.
- Additional memory which is not re-initialized:
  - Reserved using USR_RESERVED_MEM (config.h).
  - Reserved using RESERVED (configAll.h).
  - Reserved using STACK_SAVE (configAll.h).

# romStart() Stack

- Stack pointer for romStart() start initialized to STACK_ADRS by romInit().
- romStart() does not return. It's stack is used until kernel is activated.
- The VxWorks kernel is activated by kernelInit() which spawns a task (with its own stack) to complete system configuration and start user application (usually by spawning another task).
- Memory for romStart() stack is not re-initialized on cold boot.

# Checking Initialization

- After final relocation of image, there may still be problems:
  - RAM access not working properly.
  - For ROM-resident images data segment may not have been relocated to correct address in RAM.
  - Download environment problems not solved.
  - Code problems.
- Verify RAM access by writing to an un-initialized global:

```
int dummyVar; /* BSS segment variable */
....
    dummyVar = 13;
    if  (dummyVar != 13)
        somethingWrongWithRAM();
```

# ROM-resident Data Segment

- For ROM-resident images verify that the data segment has been correctly initialized after the final relocation:

```
static int testVal = 13; /* data segment variable */
....
    if  (testVal != 13)
        somethingWrongWithData();
```

- If there are problems and RAM access works, check relocation of data segment to RAM.
- For ROM-resident images romStart() copies the data segment to an architecture specific RAM address computed as an offset from the end of text in ROM.
- Check offset, particularly if WRS tools were not used to make ROM-resident image.

# Modifying romStart()

- During BSP development debug code may need to be placed into bootInit.c.
- Do not modify ../config/all/bootInit.c. Make a copy of this file in the BSP directory, and modify the copied file.
- To link the copy and not the original file, add the following line to Makefile after the macro HEX_FLAGS:

  BOOTINIT = bootInit.c
- Macro BOOTINIT is used to access bootInit.c during VxWorks builds in rules.bsp.
- Default value for this macro is defined in defs.$(WIND_HOST_TYPE) as ../config/all/bootInit.c.

# romStart() Configuration Macros

- Configuration macros control behavior of romStart(). These macros are defined in config.h, Makefile, configAll.h, and bootInit.c.
- BSP developers are responsible for the configuration macros in config.h, <bsp>.h, and Makefile.
- bootInit.c should not be modified. Macros in this file are:
  - BSP (or architecture) dependent. - Controlled from config.h, Makefile, and configAll.h.
  - Image type specific. - Controlled at compile time in rules.bsp. BSP developer does not need to modify.
  - Specific to code in bootInit.c.

# romStart() Configuration Macros continued

- romStart() configuration macros defined in config.h:
  - LOCAL_MEM_LOCAL_ADRS - Start of RAM.
  - LOCAL_MEM_SIZE - Size of RAM.
  - USER_RESERVED_MEM - Number of reserved bytes.

    Memory reserved from top of RAM and will not be cleared on cold boot or used by VxWorks.
  - RAM_HIGH_ADRS - RAM load address for non-ROM-resident VxWorks boot images.
  - RAM_LOW_ADRS - RAM load address for non-ROM-resident VxWorks application images.
  - ROM_TEXT_ADRS - Boot ROM entry address.
  - ROM_SIZE - Size of ROM.
  - ROM_BASE_ADRS - Base address of ROM.

# romStart() Configuration Macros continued

- romStart() configuration macros defined in Makefile:
  - RAM_HIGH_ADRS - Must agree with config.h
  - RAM_LOW_ADRS - Must agree with config.h
  - ROM_TEXT_ADRS - Must agree with config.h
  - ROM_SIZE - Must agree with config.h
- romStart() configuration macros defined in configAll.h:
  - RESERVED - Number of reserved bytes. Memory reserved from bottom of RAM, and will not be cleared on cold boot.
  - STACK_SAVE - Maximum stack size for romStart(). Architecture specific. Not cleared on cold reboot.

# romStart() Configuration Macros continued

- romStart() configuration macros defined in bootInit.c:
  - USER_RESERVED_MEM - Will be defined as zero if not defined in config.h.
  - SYS_MEM_BOTTOM - For cold boot, memory will be cleared starting at this address. It expands to:

    LOCAL_MEM_LOCAL_ADRS + RESERVED.
  - SYS_MEM_TOP - For cold boot, memory will be cleared up to (but not including) this address. It expands to:
    LOCAL_MEM_LOCAL_ADRS +

    LOCAL_MEM_SIZE - USR_RESERVED_MEM.
  - UNCMP_RTN - Name (address) of uncompression routine.
  - ROM_OFFSET - Macro to re-compute absolute addresses which are not PIC compatible.

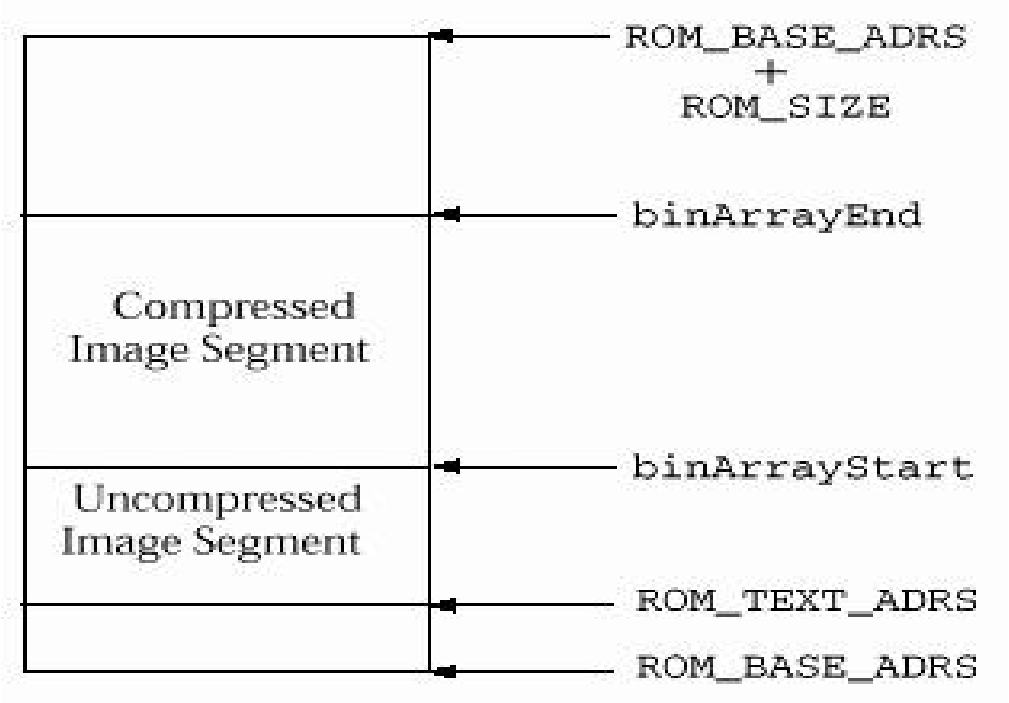
# romStart() Configuration Macros continued

- RAM_DST_ADRS - Final relocation address for compressed image. Default value is RAM_HIGH_ADRS, redefined when necessary at compile time in rules.bsp.
- RESIDENT_DATA - Architecture specific. Defined as RAM_DST_ADRS for MIPS and PowerPC. Defined as the start of the data segment otherwise.
- ROM_COPY_SIZE - For uncompressed and ROM-resident images size of image to relocate.
- ROM_BASE_ADRS - Defined in config.h. Redefined as romInit if BOOTCODE_IN_RAM is defined.
- binArrayStart - Start of compressed binary image.
- binArrayEnd - End of compressed binary image.

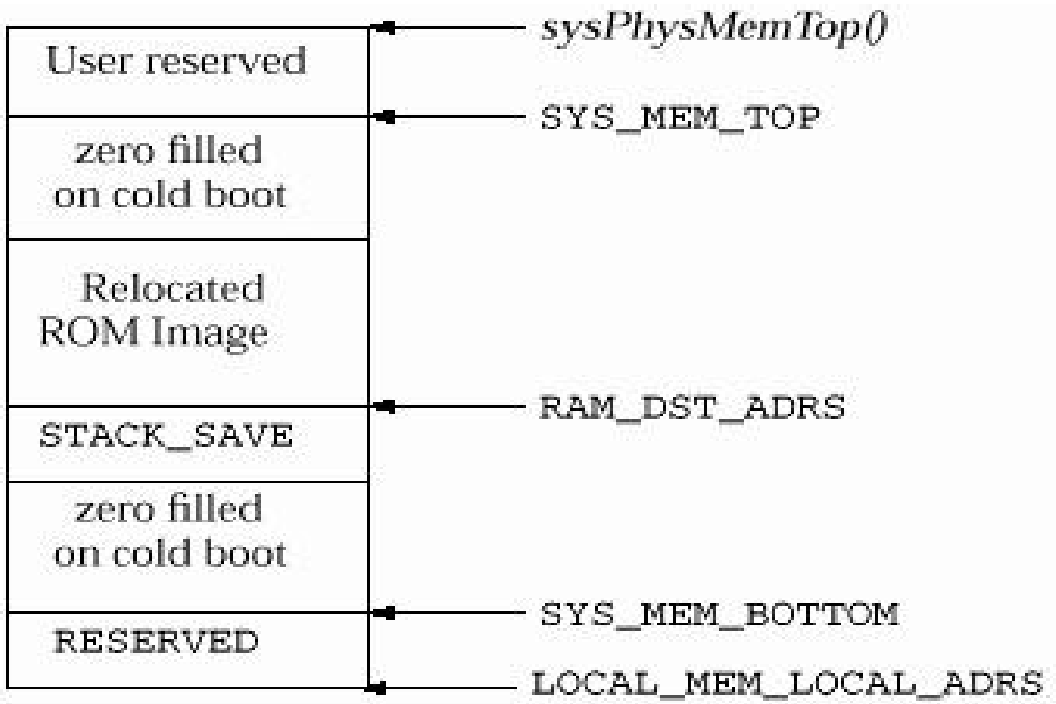
# romStart() Configuration Macros continued

- Optionally defined configuration macros for romStart():
  - BOOTCODE_IN_RAM - Used to not clear RAM on cold boot. If RAM is already initialized this macro allows BSP developer to avoid RAM initialization on cold boot. Must be defined in config.h (x86 architecture only).
  - UNCOMPRESS - Defined at compile time in rules.bsp for uncompressed image. Does not need to be redefined.
  - ROM_RESIDENT - Defined at compile time in rules.bsp for ROM-resident image. Does not need to be redefined.

# ROM Layout


ROM_BASE_ADRS
+
ROM_SIZE
binArrayEnd
Compressed
Image Segment
binArrayStart
Uncompressed
Image Segment
ROM_TEXT_ADRS
ROM_BASE_ADRS

# RAM Layout


User reserved
zero filled
on cold boot
Relocated
ROM Image
STACK_SAVE
zero filled
on cold boot
RESERVED
sysPhysMemTop()
SYS_MEM_TOP
RAM_DST_ADRS
SYS_MEM_BOTTOM
LOCAL_MEM_LOCAL_ADRS

# romStart() - Some do’s and don’ts

- Do not modify code. Functionality is controlled by modifying configuration macros.
- Code is written in C. Portion which executes in ROM should be PIC and should use a macro to compute PC relative addresses when necessary.
- Sequence of execution:
  - Copies appropriate ROM image segments to RAM.
  - Clears portions of RAM not being used (cold boot).
  - Performs uncompression if required.
  - Passes control to generic pre-kernel code (usrInit()).
- usrInit() stack begins at final relocation address and grows away from relocated image.

# Pre-Kernel Initialization - Boot Specific Code

Boot Specific vs. Generic Code

romInit.s : romInit()

PIC and VxWorks

bootInit.c : romStart()

5.5 sysALib.s : sysInit()

# sysInit() Basics

- Entry point for loadable VxWorks images. Processor is jumped to sysInit() after image is loaded into RAM.
- sysInit() resides at the load address for loadable VxWorks images RAM_LOW_ADRS.
- Performs minimum required setup to execute usrInit(). The remainder of hardware initialization is performed by generic pre-kernel code.
- Performs all the functions of romInit() except for memory system initialization.
- Routine is written in assembly language and resides in file sysALib.s.

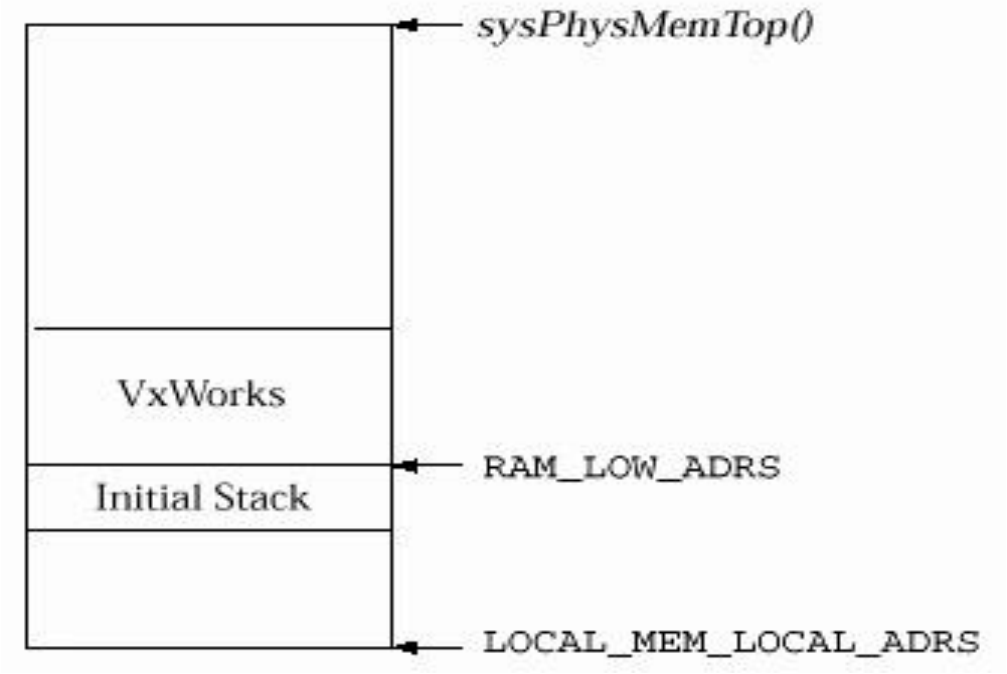
# sysInit() Code

- Routine must:
  - Mask processor interrupts and reset processor.
  - Initialize stack pointer and other registers to begin executing usrInit() and passing the boot type.
- Hardware initialization completed in sysHwInit().
- Once romInit() code has been written, it will only need to be modified to create sysInit():
  - Memory initialization code removed.
  - Upon completion jump to usrInit() not romstart().
  - Code executes in RAM, does not need to be PIC.
- Linked into all VxWorks image types but only executed for loadable images.

# Stack Initialized by sysInit()

- Stack for usrInit() set up by sysInit() grows away from VxWorks image to lower addresses in memory.
- Must be accounted for when determining load address for VxWorks image.
- Memory between RAM_LOW_ADRS and LOCAL_MEM_LOCAL_ADRS contains parameters which should not be over-written by the stack for usrInit() (which never returns). Some of these parameters are target environment specific others are generic:
  - Exception description message.
  - Shared memory anchor address.
  - Boot line.

# RAM Layout

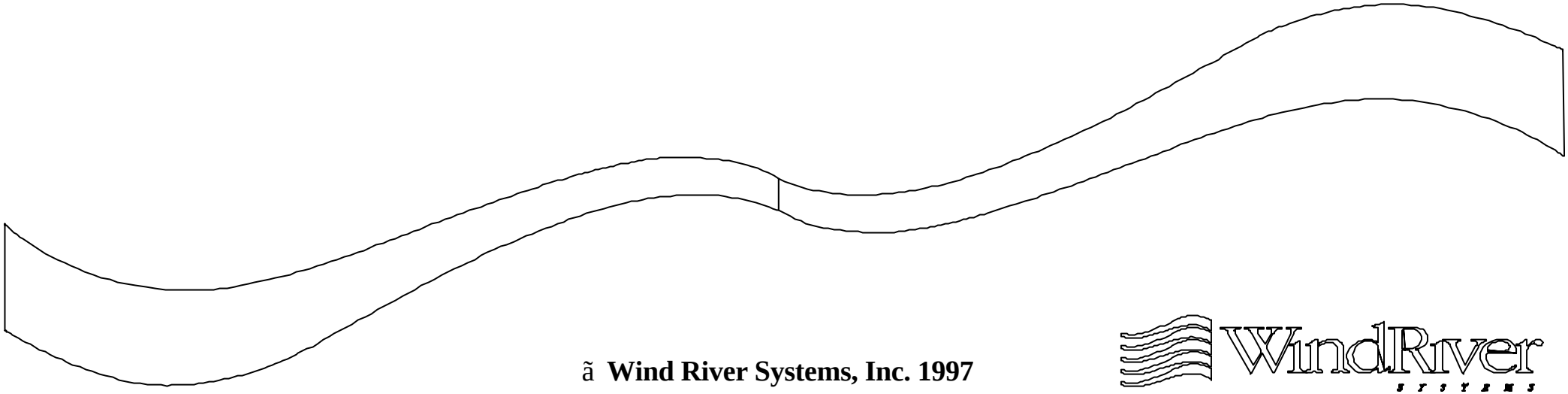


# Summary

- Details of pre-kernel initialization depend on VxWorks image type.
- VxWorks image type specific code:
  - romInit()
  - romStart()
  - sysInit()
- romInit() and romStart() execute for all images “burned” into ROM.
- sysInit() only executes for loadable VxWorks images.
- Next stage of pre-kernel initialization (following romStart() or sysInit()) is the generic routine usrInit().

# Chapter - 6

## Pre - Kernel Initialization - Generic Code





WindRiver SYSTEMS

# Pre-Kernel Initialization - Generic Code

6.1 Generic Code Overview

sysHwInit()

Activating the Kernel

# Generic Pre-Kernel Initialization Responsibilities

- The generic phase of pre-kernel initialization must produce an environment which allows the VxWorks kernel to be activated.

- Prior to kernel activation the system memory pool has not been initialized. Some implications:
  - No multi-tasking.
  - No interrupt handlers.
  - No I/O access to hardware.
  - No network interface access.

- Post-kernel code performs initialization requiring system memory pool or multi-tasking environment.

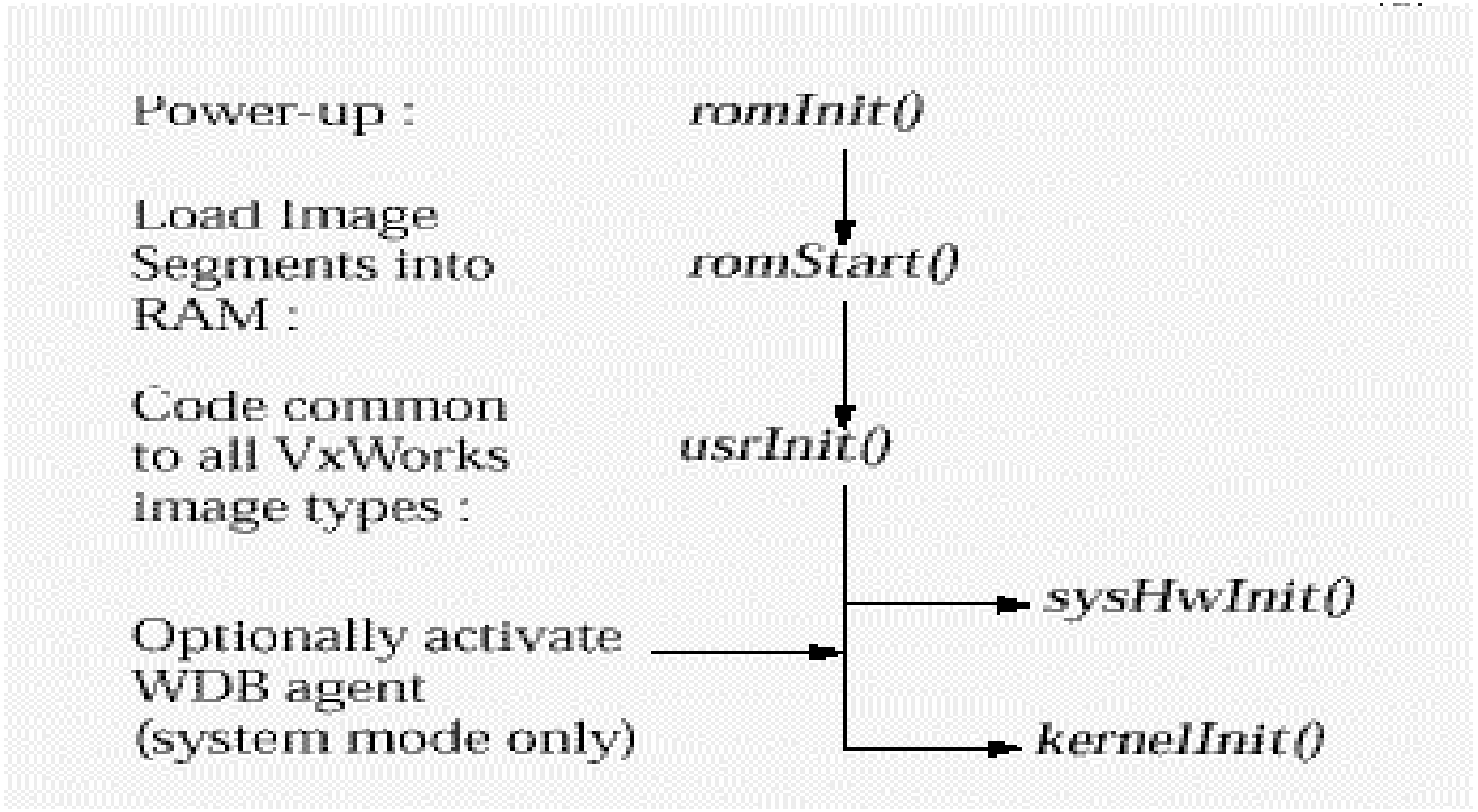
# Generic Pre-Kernel Initialization

- Generic pre-kernel initialization is performed by the C routine *usrInit()*:
  - Statically linked into all VxWorks image types.
  - Calls routine which activates VxWorks kernel.
- Primary responsibility to place hardware in a quiet state so kernel can be activated (sysHwInit()):
  - Disable all hardware interrupts.
  - Initialize hardware to a known quiescent state.
- romInit()/sysInit() perform the minimal initialization necessary to allow usrInit() to execute.
- usrInit() performs the minimal initialization necessary to activate kernel.

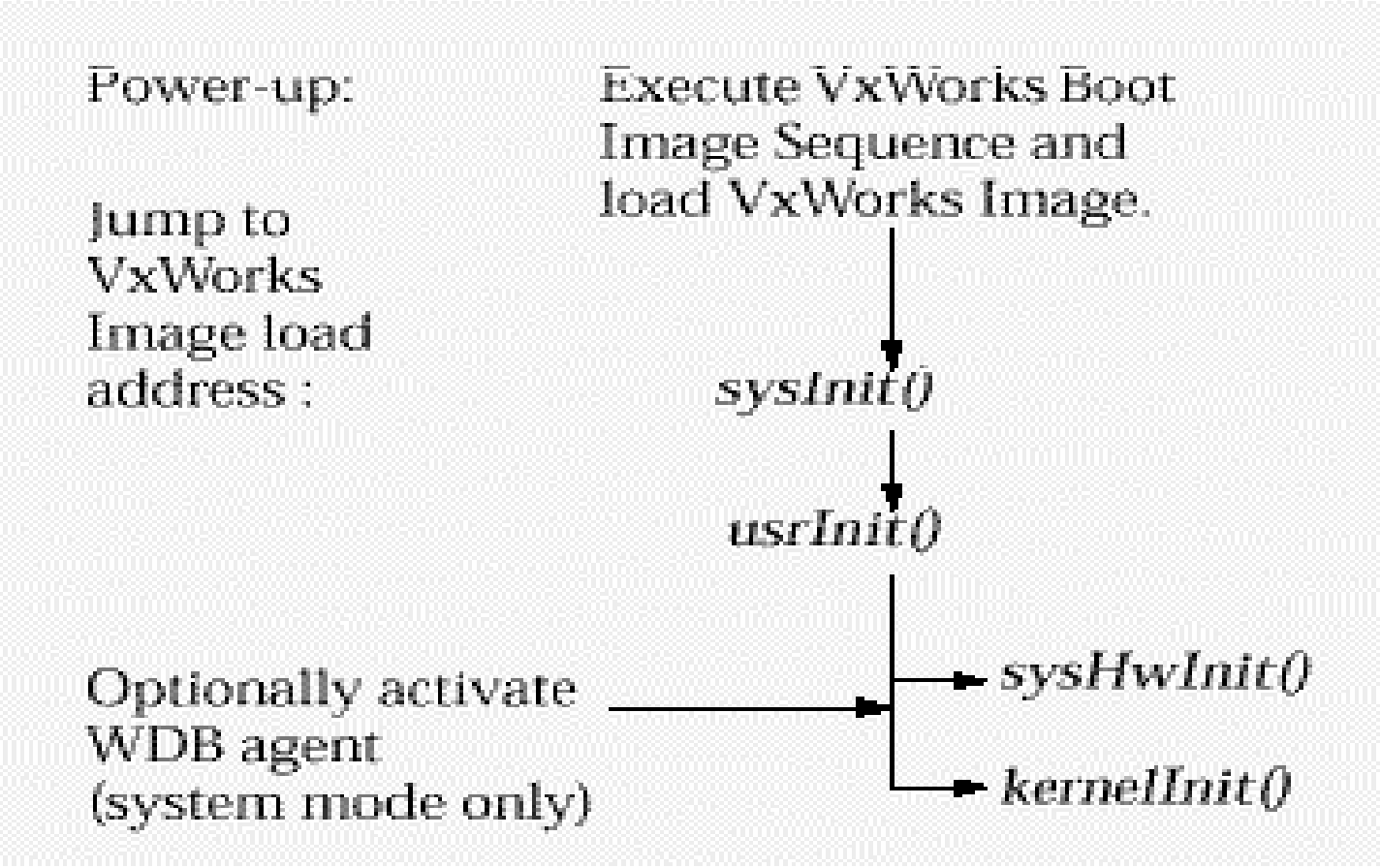
# Activating the WDB Agent Before the Kernel

- Tornado tools may be accessed using the WDB agent in system mode prior to kernel activation.
- WDB agent can be activated in system mode using a polled serial driver after sysHwInit() has executed:
  - Interrupts are masked until kernelInit() executes.
  - Necessary serial controller device initialization is performed in sysHwInit().
- System mode debugging will allow developer to access CrossWind:
  - Debug interrupt handlers.
  - Debug post-kernel device initialization.

# Pre-Kernel Initialization Sequence -VxWorks Boot and ROM Images



# Pre-Kernel Initialization Sequence -Loadable VxWorks Image



# usrInit() Stack

- usrInit() stack begins at:
  - Image load address (grows down).
  - Image load address less stack size (grows up).
- Stack initialized by:
  - sysInit() for loadable image.
  - romInit() for ROM image.
- usrInit() routine does not return.
- Its stack is used until kernelInit() spawns post-kernel initialization task tUsrRoot.
- Stack size is controlled by STACK_SAVE.
- Can examine stack using CrossWind in external mode.

# usrInit() Code

- Code resides in ../config/all/usrConfig.c.
- Should not be modified except to activate WDB agent in system mode.
- Functionality of routine is controlled by support routines and configuration parameters.
- BSP developer will write or modify reference BSP version of:
  - *sysHwInit()*.
  - Support routines - sysX().
  - BSP and driver configuration files.
  - Device driver code.

# Pre-Kernel Initialization - Generic Code

Generic Code Overview

6.2 sysHwInit()

Activating the Kernel

# sysHwInit() Responsibilities

- Place hardware environment in quiet state prior to activating VxWorks kernel:
  - Initialize features of hardware environment.
  - Disable hardware interrupts.
  - Initialize device control/status registers.
- Initialization requirements for device registers and environment parameters is BSP specific.
- Hardware interrupts must be disabled for all devices:
  - sysHwInit() executes with interrupts locked.
  - Interrupts are unlocked when kernel is activated by kernelInit().
  - ISRs cannot be installed until kernel is activated.

# Environment, Devices and Interrupts

- Environment and device initialization is performed primarily to support disabling interrupts.
- Generally the environment must be partially configured before specific devices can be accessed. May need to:
  - Initialize memory controller.
  - Configure bus access to devices.
- Environment may include an interrupt controller.
- Generally devices must be partially initialized before interrupts can be disabled:
  - Device may need to be configured to allow interrupt control registers to be accessed.
  - Interrupt controller may need to be initialized.

# Environment Initialization For Device Access

- To access devices, must have:
  - CPU access to system bus(es).
  - Devices accessible via system bus(es).
- CPU access to system bus(es) is usually provided by:
  - romInit() or sysInit() if local bus is the only bus.
  - sysHwInit() if local bus connected to other system bus(es).
- Device accessibility over system bus(es) provided by:
  - sysHwInit().

# Device Initialization

- After hardware devices are accessible, sysHwInit() can begin initialization.
- Generally device initialization in sysHwInit() is confined to disabling interrupts:
  - Interrupt handlers not available.
  - Devices not used until after kernel is activated.
- Remainder of device initialization performed after kernel activation.
- Exceptions are:
  - Devices controlled by BSP (examples: bus bridges and interrupt controllers).
  - Serial controller (polled mode).

# Devices Under BSP Control

- Generic device driver code:
  - Controls hardware (other than the CPU).
  - Can be used in multiple BSP environments
  - WRS code in **../src/drv** and ../**h/drv** directories.
- Devices managed by code which is specific to a particular BSP are under BSP control:
  - Unique devices and devices not supported by WRS.
  - Code should reside in the BSP directory.
- In addition to disabling interrupts, provide initialization for BSP devices in sysHwInit():
  - Impacts target environment and generic device operation.

# BSPs With An Interrupt Controller

- Interrupt controller is initialized in sysHwInit().
- For devices connected to interrupt controller, interrupts are masked (disabled) at the controller.
  - For devices with drivers, interrupt and other control registers are initialized after the kernel is activated using driver code except for serial controllers.
  - For devices which are under BSP control, interrupt and other control registers are initialized in sysHwInit().
- Non-interrupt related initialization may consist of:
  - Probing to test for presence of device.
  - Minimal device specific initialization to allow access to device.

# BSPs Without An Interrupt Controller

- Without an external interrupt controller device interrupt request lines connect directly to CPU.
- Device interrupt control registers are accessed:
  - Directly over local bus.
  - Mediated by an external memory controller.
- Memory controller may need to be initialized to access devices.
- Interrupts must be disabled in sysHwInit():
  - Individually.
  - On a per device basis. (Through master interrupt control register for the device.)

# Serial Controller Initialization

- Serial controller initialized for polled mode operation to allow system level debugging prior to kernel activation.
- Initialization routine is sysSerialHwInit():
  - Called by sysHwInit().
  - Code in sysSerial.c
- sysSerialHwInit():
  - Initializes SCC control structures for each channel.
  - Calls driver initialization code which disables interrupts.
- SCC control structure contains callback routines which are called by WDB agent to access the device in polled mode.

# Additional Initialization

- In addition to disabling interrupts for all devices and configuring SCCs for polled mode access, sysHwInit() may:
  - Initialize devices under BSP control.
  - Perform memory autosizing (if supported).
  - Extract hardware addresses for network interfaces.
- Board network interface hardware addresses are often stored in NVRAM or battery-backed RAM.
- Autosizing is activated by a call to sysPhysMemTop() if:
  - Supported.
  - Macro LOCAL_MEM_AUTOSIZE is defined (config.h).
  - Macro LOCAL_MEM_SIZE is set to zero.

# sysHwInit() and sysLib

- sysHwInit() is part of the sysLib library:
  - Hardware access for VxWorks and “end-user” code.
  - Hardware environment independent interface.
  - Most routines not called by “end-user” applications.
- Routines in sysLib.c are members of sysLib, however, sysLib also contains routines in other files. These routines are of the form sysX().
- sysLib.c does #include support files for:
  - Driver code which must be accessed by the BSP in the ../src/drv and BSP directories.
  - Environment and driver control parameters in the ../target/config, ../h, and ../h/drv directories.

# Debugging sysHwInit()

- If after kernel activation the initialization task tUsrRoot is not spawned, device initialization in sysHwInit() is probably not complete.
  - One or more devices may be generating interrupts.
  - Source(s) of interrupt(s) must be found.
- Debug techniques:
  - Modify sysHwInit() to connect debug routines to suspect device interrupts.
  - Use an ICE to set breakpoints in the interrupt vector table.
  - Use a logic analyzer to check for instruction access to interrupt vector table.

# Connecting Debug ISRs

- Connect debug ISR(s) using the routine intVecSet() if supported for architecture.
  - intConnect() cannot be used until kernel is activated.
- First locate the interrupt vector of the suspect interrupt:
  - Relative to base of interrupt vector table.
  - Base of interrupt vector table configured prior to calling sysHwInit() in usrInit().
  - Use INUM_TO_IVEC() macro to compute WRS interrupt vector.
- Connect debug handler to interrupt vector:

```
intVec = INUM_TO_IVEC(intNum);
intVecSet (intVec, debugISR);
```

- The debug ISR should indicate the source of interrupt.

# Caveat For Connecting Debug ISRs

- The base of the interrupt vector table is initialized by:

  intVecBaseSet ((FUNCPTR *) VEC_BASE_ADRS)

- This routine initializes the CPU's vector base register to the value of the macro VEC_BASE_ADRS (configAll.h)
- Not all architectures have an interrupt vector base register. intVecBaseSet() is no-op in this case.
  - These architectures may also not support intVecSet().
- If intVecSet() is not supported:
  - Statically create a system interrupt table.
  - Write support routine for intVecSet().
  - Supporting intVecSet() will be discussed in an upcoming chapter.

# System Restarts

- If sysHwInit() encounters an error which prevents it from continuing initialization it will restart the system:

```
STATUS sysToMonitor (startType)
```

startType — Restart type. Variable type: int.

- Transfers control to ROM monitor. BSP Specific.
- Restart types defined in ../h/sysLib.h:
  - BOOT_NORMAL - Normal reboot with countdown.
  - BOOT_NO_AUTOBOOT - No autoboot.
  - BOOT_CLEAR - Clear memory.
  - BOOT_QUICK_AUTOBOOT - Fast autoboot.

# Review of Initialization In sysHwInit()

- sysHwInit() initialization details will be specific to BSP environment. Generic responsibilities:
  - Complete any initialization performed in romInit()/ sysInit() which is required to access devices.
  - Disable all hardware interrupts.
  - Leave additional configuration to driver routines after kernel is activated, except for serial controller(s) and hardware under BSP control.
  - Initialize serial controller to be accessible in polled mode for system level debugging prior to kernel activation.
  - Configure autosizing for physical memory if supported.

# Pre-Kernel Initialization - Generic Code

Generic Code Overview

sysHwInit()

6.3 Activating the Kernel

# Kernel Activation

- Kernel is activated by the kernelInit() routine:
  - Initializes and starts the kernel.
  - Defines system memory partition.
  - Activates a task tUsrRoot to complete initialization.
  - Unlocks interrupts.
  - Uses usrInit() stack.
- Kernel data structures are configured by usrKernelInit() which is called after sysHwInit() but before kernelInit():
  - Routine in ../src/config/usrKernel.c
  - Uses configuration macros in configAll.h to initialize appropriate libraries.
  - Data structures include: binary semaphores, watch-dog timers, kernel queues, etc.

# System Memory Partition

- Top and bottom of system memory partition passed as arguments of kernelInit().
- Bottom of system memory partition:
  - FREE_RAM_ADRS if INCLUDE_WDB is not defined.
  - FREE_RAM_ADRS + WDB_POOL_SIZE if INCLUDE_WDB is defined.
- Top of the system memory partition will be the return value of sysMemTop().
- tUsrRoot will:
  - Initialize memory partition management libraries.
  - Optionally initialize MMU management facilities.

# tUsrRoot

- Activated in kernelInit(). First task to run, stack allocated from system memory partition:
  - Priority 0.
  - Stack size controlled by ROOT_STACK_SIZE.
- tUsrRoot will:
  - Initialize memory partition library.
  - Initialize the system clock.
  - Initialize the I/O system - optional.
  - Create devices - optional.
  - Configure network - optional.
  - Activate WDB agent - optional.
  - Activate application.

# Unlocking Interrupts

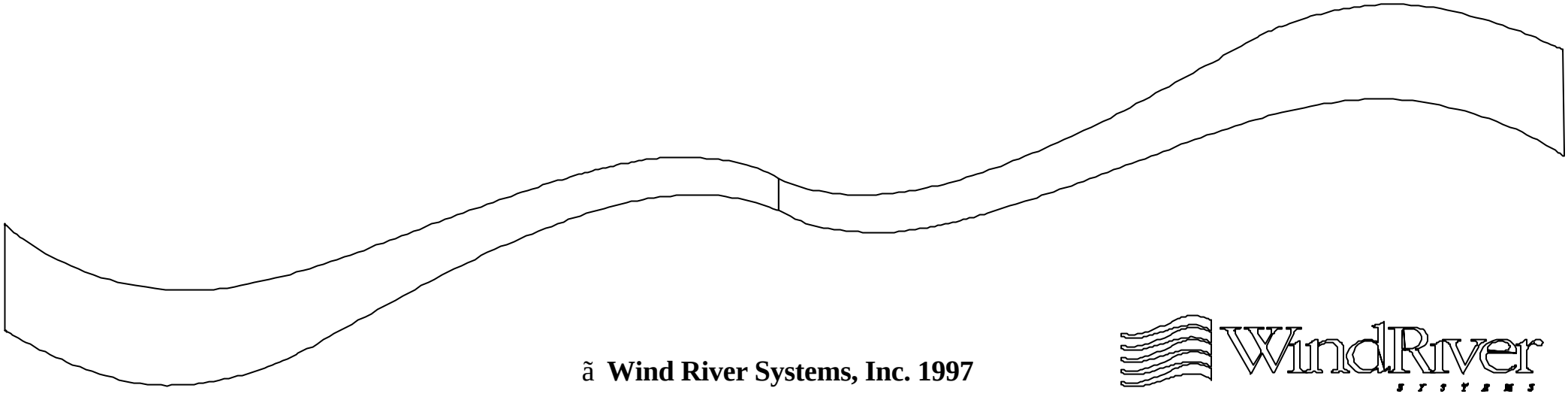
- Interrupts unlocked as part of the kernel call which initializes tUsrRoot (taskInit()).
- If all interrupts have not been disabled in sysHwInit() ***tUsrRoot*** may not execute properly.
- If architecture supports a dedicated interrupt stack:
  - Stack memory size specified through argument list.
  - Placed at beginning of system memory partition.
  - Filled with 0xee for checkStack().
- Interrupts will be enabled after appropriate ISRs are placed on the interrupt vector table. Examples will be given as the course progresses.

# Summary

- Generic phase of pre-kernel initialization must produce an environment which allows the VxWorks kernel to be activated by kernelInit().
- Generic pre-kernel initialization is performed by the C routine usrInit() which is statically linked into all VxWorks image types.
- usrInit() calls sysHwInit() to disable all hardware interrupts.
- WDB agent may be activated in system mode after sysHwInit() returns. Provides access to Tornado tools.
- kernelInit() activates kernel and unlocks interrupts.

# Chapter - 7

## Pre - Kernel
## Initialization - Debugging
## With Tornado





WindRiver SYSTEMS

# Pre-Kernel Initialization - Debugging With Tornado

7.1 Overview

Using the WDB Agent

SCC Support For WDB Agent

Debugging Techniques

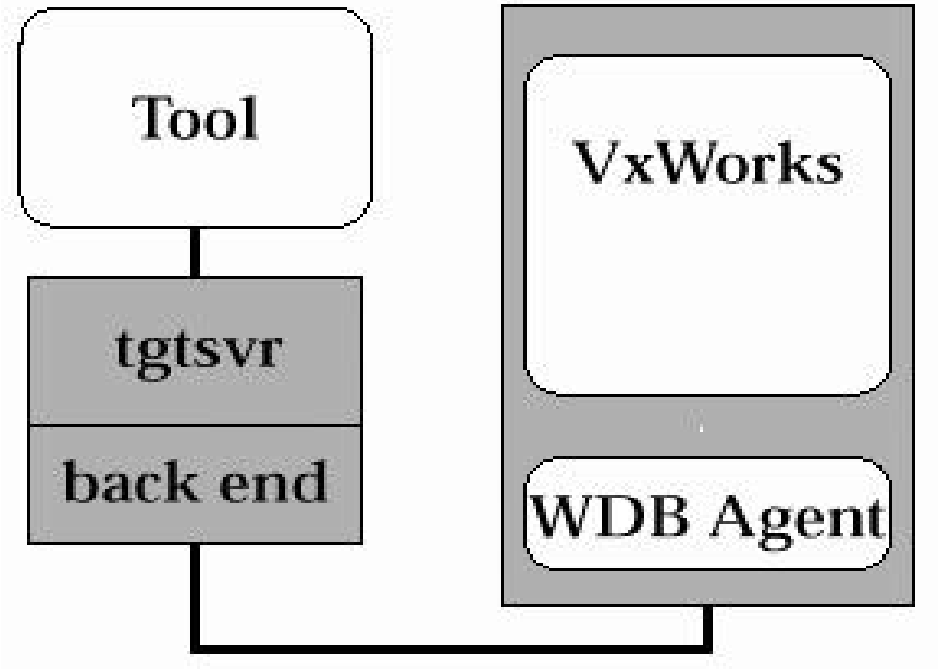
# Pre-Kernel Tornado

- Tornado tools may first be accessed after sysHwInit() completes:
  - Tools available prior to kernel activation.
  - Tools available after kernel activation but prior to interrupts being enabled for target backend.
- Access will require WDB agent to execute in system mode.
- In system mode either VxWorks executes or WDB agent executes:
  - Similar to ROM monitor, only debug agent statically linked into VxWorks image.

# Tornado Tools

- Tornado tools include host based:
  - Debugger - CrossWind.
  - Shell - WindSh.
  - Configuration tool - WindConfig.
  - Object loader/unloader.
  - Symbol table management.
  - Customized tools.
- Tools can be modified and enhanced using the Tool Command Language (Tcl).
- Target server mediates host platform access to target.
- Target agent (WDB agent) can execute as a VxWorks task or independently of the VxWorks kernel.

# Host - Target Interaction



# Activating the WDB Agent Before the Kernel

- Tornado tools may be accessed using the WDB agent in system mode prior to kernel activation.
- WDB agent can be activated in system mode using a polled driver after sysHwInit() has executed:
  - Interrupts are masked until kernelInit() executes.
  - Necessary device initialization is performed in sysHwInit().
- System mode debugging will allow developer to access CrossWind:
  - Debug interrupt handlers.
  - Debug post-kernel device initialization.

# Pre-Kernel Initialization - Debugging With Tornado

Overview

7.2 Using the WDB Agent

SCC Support For WDB Agent

Debugging Techniques

# Using the WDB Agent - Overview

- Supporting access to Tornado tools prior to kernel activation requires:
  - NetROM.
  - Customized backend for ICE.
  - Serial Interface.
- WDB agent must be configured for the appropriate backend connect strategy.
- For a serial interface:
  - Appropriate SCC channel must be configured to be accessed in polled mode.
  - usrInit() must be modified to initialize WDB agent and suspend execution.

# Configuring the WDB Agent

- Configuration of WDB agent requires specification of:
  - Backend connection policy.
  - Agent execution mode.
- Configuration done in config.h.
- Backend connections (prior to kernel activation):
  - WDB_COMM_NETROM - NetROM interface.
  - WDB_COMM_SERIAL - SCC interface.
  - WDB_COMM_CUSTOM - Custom interface.
- Agent mode must be system (external):
  - WDB_MODE_EXTERN
- Run-time configuration done by a routine wdbConfig().

# Activation of External WDB Agent

- WDB agent must be configured and then activated in usrInit() after sysHwInit() executes.
- Configuration and activation require the following routines to be called:
  - wdbConfig() - Configures agent.
  - wdbSuspendSystemHere() - Activates agent.
- wdbConfig() sets up the external agent's context and returns.
- wdbSuspendSystemHere() transfers control of the CPU to the external agent.
- Tornado tools can then access target.

# Modifications to usrInit()

- InusrConfig.c insert the following code:

```
....
sysHwInit ();           /* initialize system hardware */
/* System debug mode */
    wdbConfig();
    wdbSuspendSystemHere(NULL, 0);
    usrKernelInit ();   /*configure the Wind kernel*/
```

- InusrConfig.c comment out the call to wdbConfig() (in usrRoot()).

# Suspending the System

- The routine wdbSuspendSystemHere():
  - Locks interrupts before transferring control to the external agent (this is not relevant here as interrupts are locked).
  - Calls the underlying routine wdbSuspendSystem() which is the routine called by Tornado's breakpoint library to halt execution.
  - Allows a callback routine (with one argument) to be executed after system is suspended. Parameters are passed through argument list.
- To begin a CrossWind debug session use the attach command as usual:

```
(gdb)  attach system
```

# Pre-Kernel Initialization - Debugging With Tornado

Overview

Using the WDB Agent

7.3 SCC Support For WDB Agent

Debugging Techniques

# System Level Debugging With Serial Backend

- For a serial backend in external mode, WDB agent code will call:
  - Polled serial drivers for input and output.
  - Configuration routines to access serial device.
- If target contains WRS supported SCC, only configuration which is required:
  - Backend interface - WDB_COMM_SERIAL
  - WDB agent mode - WDB_MODE_EXTERN
  - Undefine INCLUDE_WDB_VIO
- If target does not contain WRS supported SCC, use template serial driver to develop driver.

# Serial Support For WDB Agent

- Support for WDB agent in system mode prior to kernel activation requires:
  - Initialization of serial I/O control structures.
  - Ioctl() routine to set access mode as polled.
  - Input and output poll routines.
  - Identification serial channel to be used.
- Providing this support will require modification of files:
  - ../src/drv/sio/templateSio.c
  - ../config/<bspName>/sysSerial.c
  - ../h/drv/sio/templateSio.h
- Support routines are invoked by sysSerialHwInit(), wdbConfig(), and wdbSuspendSystemHere().

# Structure of A Serial Driver

- One control structure for each serial device (refer to templateSio.h):
  - Contains control structure for each SCC channel (TEMPLATE_CHAN).
  - Each channel control structure has a SIO_CHAN structure as its first member.
- SIO_CHAN structure has one member; a pointer to a SIO_DRV_FUNCS structure.
- SIO_DRV_FUNCS structure has five members; used to manage serial device. Will be invoked by WDB agent.
- Both SIO_CHAN and SIO_DRV_FUNCS are defined in sioLib.h.

# Channel Control Structure

- Declared in templateSio.h:

```
/* device and channel structures */
typedef struct
    {
    /* must be first */
    SIO_CHAN            sio;        /* SIO_CHAN element */
....
    int                 mode;       /* current mode */
    int                 baudFreq;   /* clock freq */
    int                 options;    /* Hardware ops */
    } TEMPLATE_CHAN;
```

# Device Control Structure

- For template serial device with two channels (portA and portB):

```
typedef struct
    {
    TEMPLATE_CHAN portA;
    TEMPLATE_CHAN portB;
    volatile char * masterCr;
    } TEMPLATE_DUSART;
```

  - Note master control register operates on a chip (not channel) level.
  - Separate channel control structure for each channel.
  - Declared in templateSio.h.
  - Can also use array elements for SCC channels.

# SIO_CHAN Structure

- SIO_CHAN structure is a pointer to a structure containing driver callbacks, SIO_DRV_FUNCS:

```
LOCAL SIO_DRV_FUNCS templateSioDrvFuncs =
    {
    templateIoctl,
    templateTxStartup,
    templateCallbackInstall,
    templatePollInput,
    templatePollOutput
    };
```

  - Declared in templateSio.c.
  - Will need to modify templateIoctl(), templatePolloutput(), and templateInput().
  - Other routines can be NULLed.

# Initialization of SCC

- For each SCC device, SCC control structure initialized by sysSerialHwInit() (refer to sysSerial.c):
  - Initialization control structures for each channel of SCC (TEMPLATE_CHAN). Control macros defined in templateSio.h.
  - Call serial driver routine, templateDevInit(), to initialize SIO_DRV_FUNCS structure and hardware (code in driver templateSio.c).
- Each SCC control structure declared in sysSerial.c.
  - Array of channel control structures with one element per channel (refer to sysSerial.c).
  - Structure with one channel control structure member per channel (refer to templateSio.h).

# Configuring SCC Access

- Baud rate and access mode for the SCC controlled by the templateIotcl() routine. Required commands:
  - SIO_MODE_SET
  - SIO_BAUD_SET
- Mode configuration:
  - Modify TEMPLATE_INT_ENABLE in templateSio.h.
  - If necessary modify interrupt disable code in templateModeSet() routine.
- Baud rate configuration:
  - Modify code to support setting baud rate.
  - Modify TEMPLATE_BAUD_MIN and TEMPLATE_BAUD_MAX macros in templateSio.h.

# Controlling SCC Access

- Access is provided by the polling routines templatePollInput() and templatePollOutput():
- Need to define parameters in templateSio.h:
  - Modify TEMPLATE_TX_READY macro for output.
  - Modify TEMPLATE_RX_AVAIL macro for input.
- For many SCCs will not need to modify code.
- For some SCCs will need to modify data transfer register management code.
- These routines will be called in a loop by the WDB agent after wdbSuspendSystemHere() is called.

# Channel Access

- The WDB agent uses the configuration macro WDB_TTY_CHANNEL to determine which channel it will use to access a SCC. The default value is 1.
- To configure the SCC access channel the agent must first obtain address of SIO_CHAN structure for that channel.
- The routine sysSerialChanGet() converts a channel number to an address of a SIO_CHAN structure.
- As part of wdbConfig(), sysSerialChanGet() will be called with the argument WDB_TTY_CHANNEL.
- Should not need to modify sysSerialChanGet() in sysSerial.c (except channel control structure name).

# Pre-Kernel Initialization - Debugging With Tornado

Overview

Using the WDB Agent

SCC Support For WDB Agent

7.4 Debugging Techniques

# Tornado Tools and Debugging

- Primary Tornado debug tool will be CrossWind in system mode.
- WindSh may also be useful for:
  - Dynamically allocating memory for new global variables eliminating the need to re-build and re-load a VxWorks image.
  - Using lkup to examine the symbol table.
- Custom debug tools can also be developed:
  - Customize CrossWind.
  - Write a new loader to support unsupported OMF.
  - Build and install a custom Tornado tool using WTX protocol.

# CrossWind Customization

- Standard system mode debugging facilities available.
- CrossWind can be modified to support features such as:
  - Hardware breakpoints.
  - Menu access to customized debug commands.
  - Defining new buttons.
  - Initialization of debug session(s).
- Modifications can be made to:
  - GUI interface.
  - GDB debug engine.
- CrossWind interaction with target server can be customized using the WTX libraries.

# CrossWind Customization Files

- CrossWind can be modified using Tcl and GDB commands.
- Place Tcl scripts in ~/.wind/cosswind.tcl to:
  - Modify graphical presentation.
  - Define new buttons.
  - Provide menu access to customized commands.
- Place Tcl scripts in ~/.wind/gdb.tcl to:
  - Define Tcl procedures for new GDB commands.
- Place GDB commands in ~/.gdbinit to:
  - Initialize a debug session. (This script file is executed each time a GDB session, not just a CrossWind session, is started.)

# Dynamic Callbacks and Debugging

- Time to place VxWorks image in target environment at this phase of development may be expensive.
  - Low bandwidth serial line load to RAM or Flash.
  - Burning VxWorks ROM image.
- Using dynamic callback code may be more efficient.
- Code can be dynamically replaced using CrossWind:
  - Declare a callback (or hook routine) in loaded Vxworks image.
  - Conditionally call routine in loaded image.
  - Load module with desired code using CrossWind.
  - Use GDB set command to assign desired function to callback.

# Dynamic Callback Example

- Include in loaded image:

```
void (*debugCallback) (void);
....
void sysBspCode (void)
    {
    if (debugCallback != NULL)
        {
        debugCallback();
        return;
        }
    ....
    }
```

- From CrossWind load the module containing routine of interest (e.g. debugRoutine()); and from GDB prompt:
  - (gdb) set debugCallback = debugRoutine

# Other CrossWind Applications

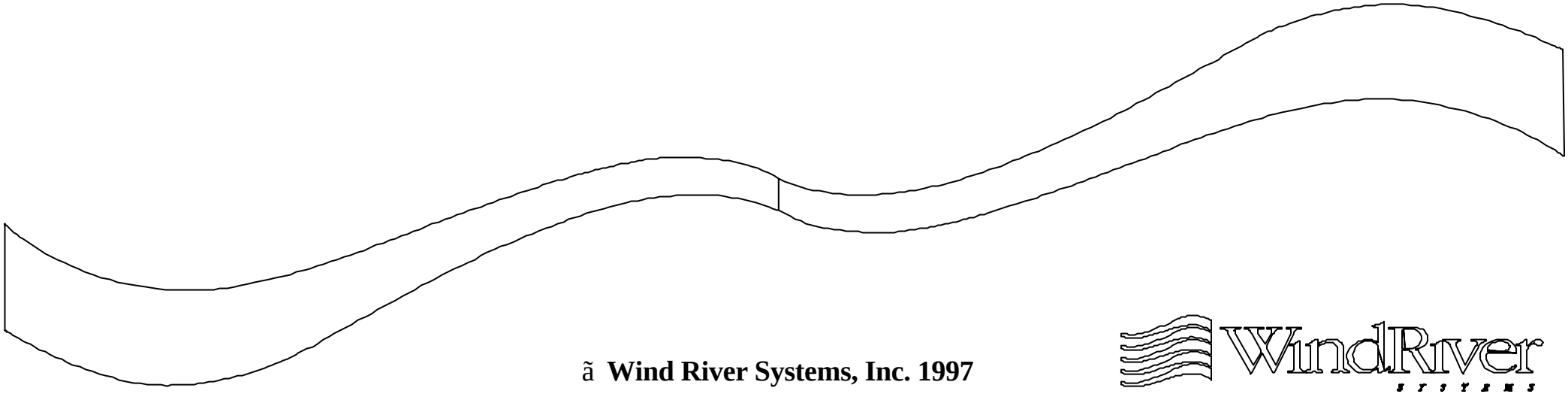
- Instead of hot-swapping, dynamically loaded code can be invoked using the GDB call command:
  - (gdb) call debugRoutine
- Disadvantage of using call command is that original code still executes.
- If hardware environment has abort switch consider connecting a debug routine to the abort interrupt using intVecSet().
- Use CrossWind to set breakpoint in abort debug ISR:
  - Allows developer to gain control of system when it dies.

# Summary

- Tornado tools can be accessed prior to kernel activation using the WDB agent:
  - Configured for external mode.
  - Configured to use appropriate backend connection.
  - Configured not to include virtual I/O.
- In system (external) mode either VxWorks executes or WDB agent executes.
- usrInit() must be modified after the call to sysHwInit() to call:
  - wdbConfig()
  - wdbSuspendSystemHere()
- CrossWind can attach to system to provide debug tools.

# Chapter - 8

## Memory





WindRiver SYSTEMS

# Memory

8.1 Overview

Configuring Memory

MMU Issues

Cache Issues

Memory Probes

# Overview

- Primary memory management issues for BSP:
  - Initialization.
  - Access interface.
- Main memory initialized by romInit(), bus access (for devices) initialized in sysHwInit() if required.
- BSP will need to support memory access and management strategies:
  - Configuration of main memory.
  - Access to NVRAM.
  - Virtual maps for MMU.
  - Cache strategies.
  - Memory probes.

# Memory

Overview

8.2 Configuring Memory

MMU Issues

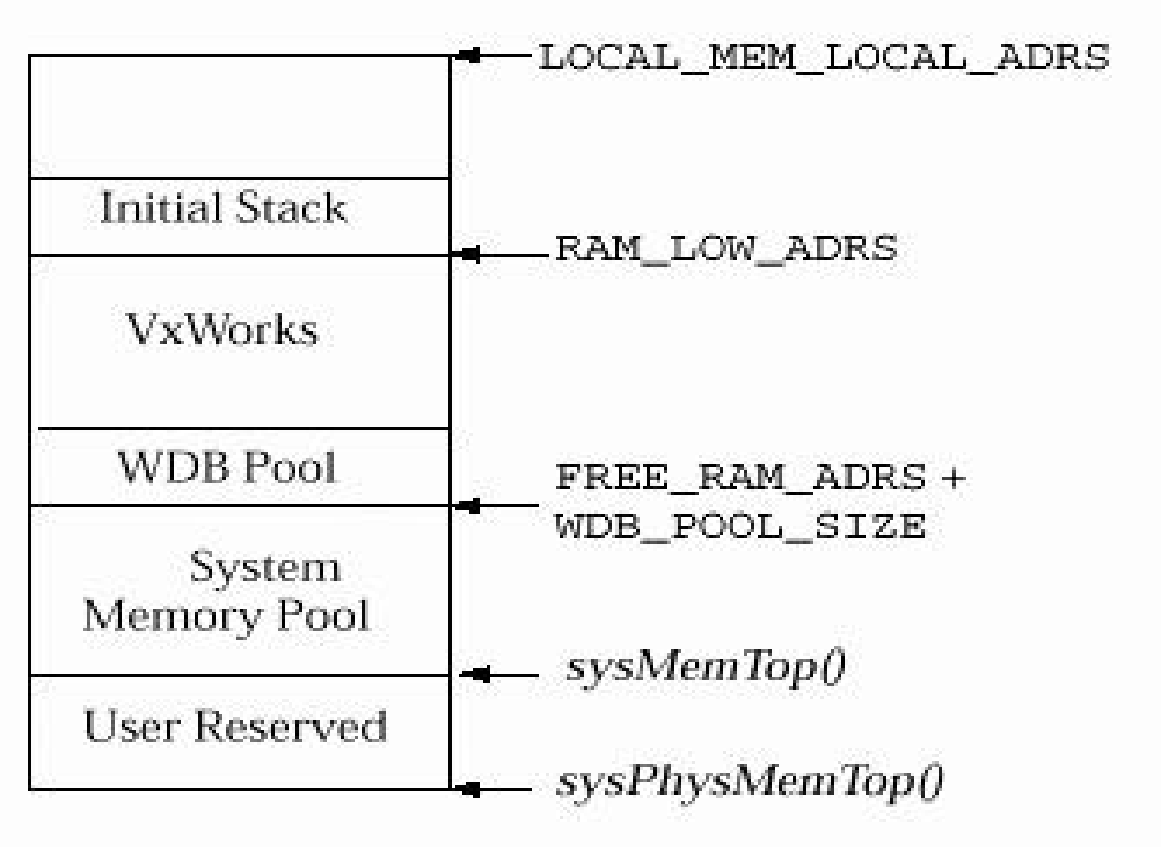
Cache Issues

Memory Probes

# Memory Configuration

- BSP responsible for configuring main memory for post-kernel operation:
  - Critical addresses must be defined.
  - If MMU is used memory maps must be specified.
  - Support routines must be provided.
- Memory addresses specified in:
  - config.h - User configurable.
  - <bsp>.h - Target dependent not user configurable
- Required BSP memory support routines:
  - sysMemTop().
  - sysNvRamSet().
  - sysNvRamGet().

# RAM Layout



# Top of System Memory

char * sysMemTop (void)

- Routine returns address of the top of system memory:

```
char * sysMemTop (void)
    {
    static char * memTop = NULL;
    if (memTop == NULL)
            {
        memTop = sysPhysMemTop() - USER_RESERVED_MEM;
            }
    return memTop;
    }
```

- Code in sysLib.c.

# Memory Autosizing

- Memory autosizing allows the size of physical memory to be configured during initialization.
  - If autosizing is not activated (or supported) size of physical memory is statically defined as LOCAL_MEM_SIZE in config.h.
- Autosizing details are architecture dependent, typically:
  - When DRAM is initialized in romInit(), configuration information is stored in memory controller registers or/and software structures.
  - During autosizing, configuration information is read and interrupted to compute the total size of physical memory.
- Routine to support autosizing is sysPhysMemTop().

# Memory Autosizing - cont.

char * sysPhysMemTop (void)

- Routine returns address of top of physical memory.
- This routine will provide dynamic memory sizing if LOCAL_MEM_AUTOSIZE is defined in config.h.
- BSP autosizing support is optional. If reference BSP code is not modified statically defined default value will be returned.
- sysPhysMemTop() is called by sysHwInit():
  - Must be called before kernelInit() as this is when sysMemTop() is called.

# NVRAM Configuration

- All BSPs must have an NVRAM interface even if there is no non-volatile RAM in the target environment. Interface must support:
  - sysNvRamSet()
  - sysNvRamGet()
- Total NVRAM size must be defined in config.h as NV_RAM_SIZE.
  - If no NVRAM present define as NONE.
- If present, NVRAM is used to store boot parameters for loadable images.
- Default configuration reserves 255 bytes at the beginning of NVRAM for boot parameters.

# NVRAM Configuration - cont.

- NVRAM size and location for boot parameters defined in configAll.h:
  - BOOT_LINE_SIZE defines NVRAM size reserved for boot parameters. Default is 255 bytes.
  - NV_BOOT_OFFSET defines beginning of NVRAM reserved for boot parameters. Default is 0.
  - To override default values, redefine macros in config.h.
- Routines to set/get NVRAM contents:
  - Part of driver located in ../src/drv/mem.
  - If no NVRAM present use ../src/drv/mem/ nullNvRam.c.
  - Included in sysLib.c.

# NVRAM Support Routines

STATUS sysNvRamSet (string, strLen,
offset)

| | |
|---|---|
| string | String to be copied into NVRAM. Variable type: char * |
| strLen | Number of bytes to copy. Variable type: int. |
| offset | Byte offset into NVRAM, Variable type: int. |

- Routine will:
  - Copy string to location NV_BOOT_OFFSET + offset.
  - Enable NVRAM read/write and write data.

# NVRAM Support Routines

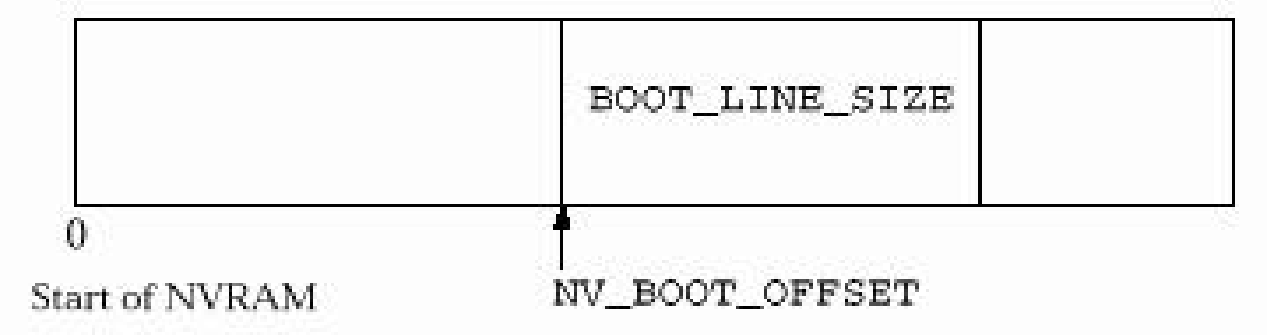
STATUS sysNvRamGet (string, strLen,
offset)

| | |
|---|---|
| string | Where to copy NVRAM. Variable type: char * |
| strLen | Number of bytes to copy. Variable type: int. |
| offset | Byte offset into NVRAM, Variable tyoe: int. |

- Routine will:
  - Copy contents of NVRAM location NV_BOOT_OFFSET + offset to string.
  - Read data and terminate string with EOS.

# Caveat For NVRAM Access

- NVRAM set/get routines displace offset parameter by NV_BOOT_OFFSET before accessing NVRAM:
  - offset += NV_BOOT_OFFSET;
- If NV_BOOT_OFFSET is greater than zero, provide access to NVRAM bytes before boot code with a negative offset values.



# Memory

Overview

Configuring Memory

8.3 MMU Issues

Cache Issues

Memory Probes

# MMU Overview

- MMU is primarily under the control of architecture library, however, BSP is responsible for providing support with physical memory description.
  - Physical memory description used by MMU to create initial maps to virtual address space.
  - Default maps are flat, one-to-one between physical and virtual memory spaces.
- MMU initialized by tUsrRoot call to usrMmuInit() which initializes and enables MMU:
  - First initialization of system memory pool facilities.
  - Second initialization of MMU.
  - MMU available for remainder of post-kernel initialization.

# Physical Memory Descriptor

- Initial (static) physical memory map defined in sysLib.c. It is an array of structures of type SYS_PHYS_MEM_DESC defined in ../h/vmLib.h:

```
typedef struct phys_mem_desc
    {
    void *virtualAddr;          /* Virtual address. */
    void *physicalAddr;         /* Physical address. */
    UINT len;                   /* Length of mapping */
    UINT initialStateMask;      /* State mask for map. */
    UINT initialState;          /* State for map. */
    } PHYS_MEM_DESC;
```

- States for maps:
  - Valid or invalid.
  - Writable or not.
  - Cacheable or not.

# Physical Memory Descriptor - cont.

- Example virtual-to-physical map element:

```
PHYS_MEM_DESC sysPhysMemDesc [] =
        {
        {
        /* Local DRAM */
        (void *) RAM_LOW_ADRS,
        (void *) RAM_LOW_ADRS,
        LOCAL_MEM_SIZE - RAM_LOW_ADRS,
        VM_STATE_MASK_VALID |
        VM_STATE_MASK_WRITABLE
    |   VM_STATE_MASK_CACHEABLE,
        VM_STATE_VALID | VM_STATE_WRITABLE
    |       VM_STATE_CACHEABLE
        },
```

- Configuration macros used by architecture library to initialize MMU translation tables.

# Virtual Memory Mapping

- To modify physical-to-virtual memory map(s):
  - Modify sysPhysMemDesc[] (static maps).
  - Call vmBaseStateSet() (dynamic modification).
- Memory mapped on a per page basis:
  - Page size controlled by macro is VM_PAGE_SIZE defined in configAll.h, Default is 8K (except for PowerPC architectures - 4K).
  - Length of maps for sysPhysMemDesc[] should be integral number of page size.
- Each table entry will require a page table entry in physical memory:
  - Sets an upper limit on how many address maps can be defined.

# Dynamic Virtual Mapping

STATUS vmBaseStateSet (context,
pVirtual, len, stateMask state)

| | |
|---|---|
| context | Context for map. Variable type: VM_CONTEXT_ID. |
| pVirtual | Virtual address to modify state of. Variable type: void *. |
| len | Length of mapping. Variable type: int. |
| stateMask | State Mask. Variable type: Unsigned int. |
| state | State. Variable type: Unsigned int. |

- Routine changes the state of a block of virtual memory.
  - Use to modify initial memory maps defined by sysPhysMemDesc[].

# Virtual Memory Mapping - cont.

- Must map all of physical memory which is to be accessed. This includes memory mapped devices (Ethernet, SCSI, etc.).
- Writing to addresses not included in the virtual-to-physical maps will result in a bus error when the MMU is enabled.
- Usually virtual-to-physical maps are configured:
  - Local RAM - valid, writable, cacheable.
  - ROM - valid, read-only, often cacheable.
  - Flash - valid, writable, non-cacheable.
  - I/O devices - valid, writable, non-cacheable.
  - Off-target memory - valid, writable, non-cacheable.

# Memory

Overview

Configuring Memory

MMU Issues

8.4 Cache Issues

Memory Probes

# Cache Overview

- Cache and MMU configuration is architecture dependent, may be highly integrated or independent.
- In VxWorks, if MMU is enabled cache is under MMU control.
- Architecture library (cacheLib) provides basic cache management support. BSP responsibilities:
  - Select appropriate cache library and modes for multiple cache implementations.
  - If MMU is enabled, memory maps labeled as cacheable or not.
  - Follow cache strategy guidelines for device drivers, for system devices under BSP control.

# Cache Library Initialization

STATUS cacheLibInit (instMode,
dataMode)

| | |
|---|---|
| instMode | Specifies mode for instruction cache. Variable type: CACHE_MODE. |
| dataMode | Specifies mode for data cache. Variable type: CACHE_MODE. |

- Initializes cacheLib facilities:
  - Calls architecture specific initialization routine.
  - Places cache in quiet state.
  - Called by usrInit() before sysHwInit().
- Arguments specify modes for instruction/data caches.

# Cache Library Initialization - cont.

- Cache mode configuration macros:
  - USR_I_CACHE_MODE - first argument.
  - USR_D_CACHE_MODE - second argument.
- Default values for cache mode configuration macros defined in configAll.h. If necessary BSP redefines in config.h. Choices are defined in ../h/cacheLib.h:

```
#define CACHE_DISABLED          0x00
#define CACHE_WRITETHROUGH      0x01
#define CACHE_COPYBACK          0x02
#define CACHE_WRITEALLOCATE     0x04
#define CACHE_NO_WRITEALLOCATE  0x08
#define CACHE_SNOOP_ENABLE      0x10
#define CACHE_SNOOP_DISABLE     0x20
#define CACHE_BURST_ENABLE      0x40
#define CACHE_BURST_DISABLE     0x80
```

# Cache Library Initialization - cont.

- If target supports multiple cache implementations BSP is responsible for selecting appropriate library package:
  - Macro _ARCH_MULTIPLE_CACHELIB must be defined as TRUE or FALSE in ../h/arch/<someArch>/ arch<someArch>.h.
  - If TRUE supply correct cache initialization routine by declaring and initializing sysCacheLibInit in sysLib.c or config.h:

```
FUNCPTR sysCacheLibInit = (FUNCPTR) cacheXLibInit;
```

- For L2 cache, BSP may need to supply separate cache management library:
  - Maybe obtained from processor manufacturer.

# Cache Enable

STATUS cacheEnable (cache)

| | |
|---|---|
| cache | Cache to enable. Variable type: CACHE_TYPE. |

- Enables specified cache type, instruction, data, or branch using architecture specific routine.
- Must undefine macros in config.h to disable:
  - INCLUDE_CACHE_SUPPORT for any cache type.
  - USER_I_CACHE_ENABLE for instruction cache.
  - USER_D_CACHE_ENABLE for data cache.
  - USER_B_CACHE_ENABLE for branch cache.
- Routine is called in usrInit() just before kernelInit() call

# Cache and Device Code

- Guidelines for managing code which accesses devices are valid whether or not device is BSP independent:
  - Maintaining cache coherency with respect to DMA devices and hardware registers.
  - Manage device memory access methods.
  - Preventing out of order instruction execution with RISC processors.
- Desired cache management is implemented using cacheLib:
  - To allocate cache safe buffers.
  - Assign attributes to a driver (system device or BSP independent device driver) and apply implementation method(s).

# Cache and Memory Access

- Cache management facilities for devices are related to how memory accessed by device is allocated.
- Memory accessed by devices allocated using:
  - cacheDmaMalloc().
  - malloc() and memalign().
  - Data and bss segment memory.
  - Special memory region outside of system pool.
  - Allocation method unknown.
- Device register memory:
  - If memory mapped, will be allocated using one of the methods listed above.
  - If not memory mapped, will not be cacheable.

# cacheLib and Memory Allocation

void * cacheDmaMalloc (bytes)

| | |
|---|---|
| bytes | Number of bytes to allocate. Variable type: size_t. |

- Routine allocates cache-aligned, cache-safe buffer for DMA devices. Returns pointer to start of memory.
- Cache coherency management for memory allocated with cacheDmaMalloc() is dependent on MMU being enabled or not:
  - If MMU is enabled, allocated memory is marked as non-cacheable.
  - If MMU is not enabled, flush and invalidate macro routine calls must be inserted into code.

# Cache Macro Routines

- cacheLib manages flush, invalidate, and other macro routines with CACHE_FUNCS and CACHE_LIB structures (see ../h/cacheLib.h). Example:

```
typedef struct /* Driver Cache Routine Pointers */
    {
    FUNCPTR flushRtn;
    FUNCPTR invalidateRtn;
    FUNCPTR virtToPhysRtn;
    FUNCPTR physToVirtRtn;
    } CACHE_FUNCS;
```

- Macro routines use routine pointers in CACHE_FUNCS and CACHE_LIB structures. Example:

```
#define CACHE_DRV_FLUSH(pFuncs, adrs, bytes) \
(((pFuncs)->flushRtn == NULL) ? OK : \
((pFuncs)->flushRtn) (DATA_CACHE, (adrs), (bytes)))
```

# Cache Macro Routines - cont.

- In general if a particular macro routine is a no-op it's CACHE_FUNCS or CACHE_LIB routine pointer member is set to NULL.
  - For full snooping, flush and invalidate routines are set to NULL.
- Macro routines are classified into two groups:
  - CACHE_DMA_xxxx - These routines flush, invalidate, and perform other operations on memory regions allocated with cacheDmaMalloc().
  - CACHE_USER_xxxx - These routines flush, invalidate, and perform other operations on(user) memory not acquired using cacheDmaMalloc().

# Cache Macro Routines - cont.

- CACHE_DMA_XXXX and CACHE_USER_XXXX macros defined using lower level macro routines CACHE_DRV_XXXX.
- CACHE_DRV_XXXX macros allow flexibility in providing cache coherency independent of memory allocation method:
  - Routines have additional first argument which is a pointer to a CACHE_FUNCS structure.
- Developer can use CACHE_DRV_XXXX macros to control cache coherency for:
  - Driver controlled memory.
  - Customized cache management.

# Cache Management Example

```
STATUS drvDmaExample (void * pBuf)
        {
        LOCAL BOOL freeFlag = FALSE;
        if (pBuf != NULL)
                {
                /* No buffer cache coherency problems. */
                pDrvFuncs = cacheNullFuncs;
                }
        else
                {
                pBuf = cacheDmaMalloc (BUF_SIZE);
                pDrvFuncs = cacheDmaFuncs;
                if (pBuf == NULL)
                        return (ERROR);
                freeFlag = TRUE;
                }
```

# Cache Management Example - cont.

```
/* Driver initialization and buffer filling. */
CACHE_DRV_FLUSH (pDrvFuncs, pBuf, BUF_SIZE);
drvWrite (pBuf);        /* Output data to device. */
/* Driver code. */
CACHE_DRV_INVALIDATE (pDrvFuncs, pBuf, BUF_SIZE);
drvWait ();             /* Wait for device data. */
/* Read and handle input data from device. */
if (freeFlag)
    cacheDmaFree (pBuf);    /* Return buffer. */
return (OK);
}
```

# Cache Strategies and Attributes

- Cache strategies for devices will determine which cacheLib facilities to use.
- BSP developer should develop cache strategy based attributes of driver. Attributes:
  - WRITE_PIPING
  - SNOOPED
  - MMU_TAGGING
  - USER_DATA_UNKNOWN
  - DEVICE_WRITES_ASYNCHRONOUSLY
  - SHARED_CACHE_LINES
  - SHARED_POINTERS
- Attributes will dictate how cacheLib will be of use.

# Example Cache Attribute - WRITE_PIPING

- Most RISC processors use write pipelining which can delay delivery of commands or data to a device.
- Macro routine CACHE_PIPE_FLUSH will flush write pipeline. Calls placed at appropriate locations in code.
- Will not resolve cache issues, driver must still flush cache.
- Must know device:
  - Some devices not impacted by pipelining delays.
  - Some devices may not function correctly without frequent pipeline flushes to sync. driver and device.

# Memory

Overview

Configuring Memory

MMU Issues

Cache Issues

8.5 Memory Probes

# Memory Probes and Busses

- VxWorks provides support for bus probes of memory:
  - vxLib routine to probe an address on the local bus for memory read/write bus errors. Routine is vxMemProbe().
  - vxALib routine to perform an atomic test and set on local bus. Routine is vxTas().
- BSP may optionally support system specific bus inquiry routines to probe addresses not on local bus:
  - Use hook routine supplied for vxMemProbe() to access system busses (including an off-board bus if present).
  - Create test and set routine for external system bus (if present) using vxTas().

# Memory Bus Error Probe

STATUS vxMemProbe (adrs, mode,
length, pVal)

| | |
|---|---|
| adrs | Address to be probed. Variable type: char *. |
| mode | Read or write. Variable type: int. |
| length | 1, 2, or 4 bytes. Variable type: int. |
| pVal | Where to return value, or pointer to value to be written. Variable type: char *. |

- Routine will trap read/write bus errors; returns OK if no bus error, and returns ERROR after handling bus error. Will not trap other errors, task making call will be suspended if no handler installed.

# Memory Bus Error Probe Code

- vxMemProbe() provides routine pointer to hook BSP specific memory probe routine:

```
STATUS status;
if (_func_vxMemProbeHook != NULL)
        /* BSP specific probe routine */
        status = (* _func_vxMemProbeHook)
                ((void *)adrs, mode, length,(void *)pVal);
else
        /* architecture specific probe routine */
```

- _func_vxMemProbeHook variable should be initialized in sysHwInit() after initialization of busses.

# System Memory Probes

- Memory probe hook routine called by vxMemProbe() should determine which bus is being probed based on input address, and call an appropriate probe routine.
  - If address corresponds to local bus BSP probe routine should call VxWorks supplied architecture specific probe routine.
- BSP probe routine is also responsible for managing any errors generated during probe:
  - Reset bridge or bus controller registers if necessary.
  - Execute any device specific exception handlers if necessary.
- Useful for probing system busses: PCI, VME, ISA, etc.

# System Test And Set

BOOL sysBusTas (adrs)

routine — Address to be tested and set. Variable type: char *.

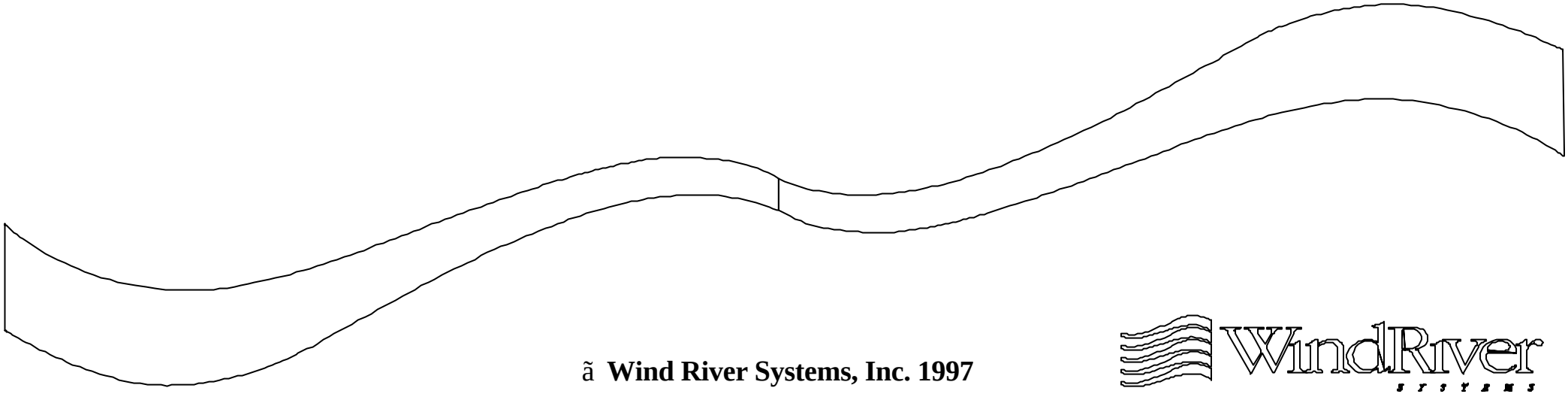
- Routine to test and set an address across the system external bus if present. Returns TRUE if the value had not been set but is now, returns FALSE if values was set already.
- Routine should provide atomic test and set using indivisible Read Modify Write cycles across external bus.
- Routine calls vxTas() if this is meaningful.

# Summary

- BSP is responsible for initializing memory and providing configuration support:
  - Configure system memory pool parameters and autosizing if supported.
  - Provide NVRAM access routines.
  - Provide physical memory descriptor for MMU.
  - Develop a cache strategy for system devices, initialize the appropriate cache library (or libraries) with appropriate modes, and re-initialize cacheLib function pointers in sysHwInit() if required.
  - Provide libraries for L2 caches if present.
  - Provide system specific memory probes for addresses not on local bus if desired. Hook to vxMemProbe() in sysHwInit().

# Chapter - 9

## Managing Interrupts





WindRiver SYSTEMS

# Managing Interrupts

9.1 Overview

Installing ISRs

Supporting Interrupt Libraries

Initializing An Interrupt Controller

Optional Interrupt Support

# Interrupts and VxWorks

- VxWorks uses an interrupt table to provide interrupt level services:
  - ISRs connected to unique interrupt vectors.
  - Table contains addresses for interrupt handlers at appropriate interrupt vectors.
- A VxWorks interrupt handler:
  - Saves the CPU interrupt context.
  - Calls a C language ISR.
  - Restores the CPU context after the ISR returns.
  - Manages an interrupt context variable which allows VxWorks to be interrupt aware.
- Base of table was configured in usrInit().

# Interrupt Vectors

- An interrupt vector is the address of the table entry for a interrupt handler. The address is relative to the base of the interrupt table.
- Interrupt numbers sequentially label interrupt table entries. Must refer to hardware documentation to obtain interrupt numbers.
- The macro INUM_TO_IVEC() converts and interrupt number to an interrupt vector. Usage:

```
intVector = INUM_TO_IVEC (intNumber());
```

- Macro definition in ../h/arch/<archName>.
- WRS definition of interrupt vector is not universal.

# BSP Responsibilities

- BSP is responsible for managing interrupts. Must supply routines to:
  - Connect interrupts.
  - Enable/disable interrupts for BSP hardware.
  - Manage interrupt hardware control registers.
  - Transfer control to boot code if ISR throws exception.
- Details depend on:
  - Presence of external interrupt controller(s) or not.
  - Interrupt policies of system bus(es).
  - Form factor for CPU interrupt support (number of external interrupt pins).
  - Protocol of devices requesting interrupt services.

# Reboot From Interrupt Level

STATUS sysToMonitor (startType)

| | |
|---|---|
| startType | Parameter passed to ROM/Flash to tell how to boot. Variable type: int. |

- This routine transfers control to ROM/Flash if ISR throws and exception:
  - Routine is generic and is called by reboot().
  - Routine may reset board environment.
  - Jumps to romInit() (warm boot).
- User can create startType values to support custom boot strategies. Default is a warm boot for exceptions thrown at interrupt level.

# Interrupt Stack

- If architecture supports a dedicated interrupt stack, stack size is configured when kernel is activated.
  - Control macro is ISR_STACK_SIZE defined in configAll.h (default is 1000 bytes).
  - Interrupt stack memory is allocated from system memory pool.
  - If dedicated interrupt stack is not supported task stack for current task is used.
- To determine if a processor supports a dedicated interrupt stack see:
  - VxWorks Programmer's Guide.
  - Tornado BSP Developer's Kit for VxWorks User's Guide.
  - Processor documentation.

# Interrupt Management Routines

- The application interface for managing interrupts is provided by intArchLib and intLib. API supplies support for:
  - Locking/unlocking interrupts.
  - Setting interrupt lock level.
  - Connecting interrupts.
  - Enabling/disabling interrupts.
  - Determining current depth of interrupt nesting.
- Some routines in intArchLib require BSP support for some architectures, implementation is BSP dependent.
- To illustrate different implementations this chapter will discuss 68k and PowerPC based BSP implementations.

# 68k and PowerPC Interrupts

- For 68k targets:
  - 68k based local bus.
  - Three IRQ lines supporting seven interrupt levels.
  - 255 distinct interrupt vectors (via local bus).
  - Vectored external interrupts.
  - No interrupt controller.
- For PowerPC targets:
  - PowerPC local bus and bridge controller.
  - One external interrupt line.
  - Interrupt controller(s).
  - Auto-vectored external interrupts.
  - No interrupt vector base register.

# Managing Interrupts

Overview

9.2 Installing ISRs

Supporting Interrupt Libraries

Initializing An Interrupt Controller

Optional Interrupt Support

# Installing Interrupts - Overview

- Interrupts should be connected to the interrupt table by the BSP.
- The routine used to connect interrupts is intConnect():
  - Builds a wrapper around ISR to create an interrupt handler.
  - Registers handler on the interrupt table at the appropriate entry.
- Construction and registration of interrupt handlers will vary with different architectures. For some architectures BSP is involved.
- When interrupts should be connected will be discussed in an later chapter.

# Installing Interrupts

STATUS intConnect (vector, routine, arg)

| | |
|---|---|
| vector | Interrupt vector to attach to. Variable type: VOIDFUNCPTR pointer. |
| routine | Routine to be called as ISR. Variable type: VOIDFUNCPTR. |
| arg | Optional argument. Variable type: int. |

- Returns OK or ERROR on error.
- The macro INUM_TO_IVEC() is used to compute the first argument to intConnect().
- Depending on target architecture, BSP may or may not be involved in supporting intConnect().

# Building Interrupt Handlers - 68k

- intConnect() calls intHandlerCreate() to create interrupt handler.
- intHandlerCreate() routine:
  - Calls malloc() to obtain memory for wrapper.
  - Places an intEnt() routine at beginning of handler.
  - Places call to routine (with argument) after intEnt().
  - Places an intExit() routine at end of handler.
  - Returns address returned from malloc() call.
- intEnt()/intExit() routines:
  - Manage intCnt variable tracking interrupt nesting.
  - Masking/unmasking interrupt levels.
  - Saving/restoring contexts.

# Creating A Handler

FUNCPTR intHandlerCreate (routine, arg)

| | |
|---|---|
| routine | Routine to be called as ISR. Variable type: FUNCPTR. |
| arg | Optional argument. Variable type: int. |

- Returns address of interrupt handler or NULL on error.
- This routine may called directly for 68K and some other architectures (not PowerPC). See Tornado Reference Guide.
- Note, architectures which support intHandlerCreate() may use different implementations.

# Attaching Interrupt Handlers - 68k

- intConnect() calls intVecSet() to install an interrupt handler on interrupt vector table.
- intVecSet() routine:
  - Computes table entry address by offsetting interrupt vector by address for base of interrupt vector table.
  - Places interrupt handler at this address.
- INUM_TO_IVEC() macro is used to obtain interrupt vector:
  - Converts interrupt number to interrupt vector by multiplying interrupt number by four.
- Attaching interrupt handlers is completely handled by intArchLib.

# Attaching A Handler

void intVecSet (vector, function)

| | |
|---|---|
| vector | Interrupt vector to attach to. Variable type: FUNCPTR pointer. |
| function | Address of handler. Variable type: FUNCPTR. |

- vector = INUN_TO_IVEC (interruptNumber)
- Can be called directly for many architectures (including 68k but no-op for PowerPC).
- Useful for installing debug ISRs during pre-kernel porting phase when interrupt wrapper code is not necessary.

# Installing Interrupts - PowerPC

- Interrupts installed with intConnect(), however, implementation involves BSP unlike the 68k case.
- WRS PowerPC targets use external interrupt controllers. Controllers have drivers which are part of the BSP. These drivers help support intArchLib:
  - This is how the BSP is involved with interrupt management routines.
- Driver routines for interrupt controller are in the ../src/ drv/intrCtl or BSP directory:
  - Initializing interrupt controller.
  - Installing interrupts.
  - Enabling/disabling interrupts.

# Installing Interrupts - PowerPC cont.

- intArchLib uses a hook routine, _func_intConnectRtn() to install interrupts.
  - intConnect() calls _func_intConnectRtn().
  - BSP assigns the appropriate routine(s) to _func_intConnectRtn() during initialization of interrupt controller in sysHwInit().
- Interrupt controller installation routine is sysXIntConnect(), where X labels the particular interrupt controller.
- Interrupt controller initialization routine, sysXInit(), must make the assignment:

```
_func_intConnectRtn =  sysXIntConnect;
```

# Installing Interrupts - PowerPC cont.

- Implementation of sysXIntConnect() will be architecture and interrupt controller dependent.
  - For example, many 603 and 604 PowerPC BSPs have multiple interrupt controller devices.
- sysXIntConnect() routine:
  - Calls malloc() to obtain memory for handler.
  - Connects ISR and optional argument to interrupt table.
  - Returns OK.
- Note, sysXIntConnect() does not install wrapper code in handler (like intHandlerCreate() for 68K) or call intVecSet() to register handler.

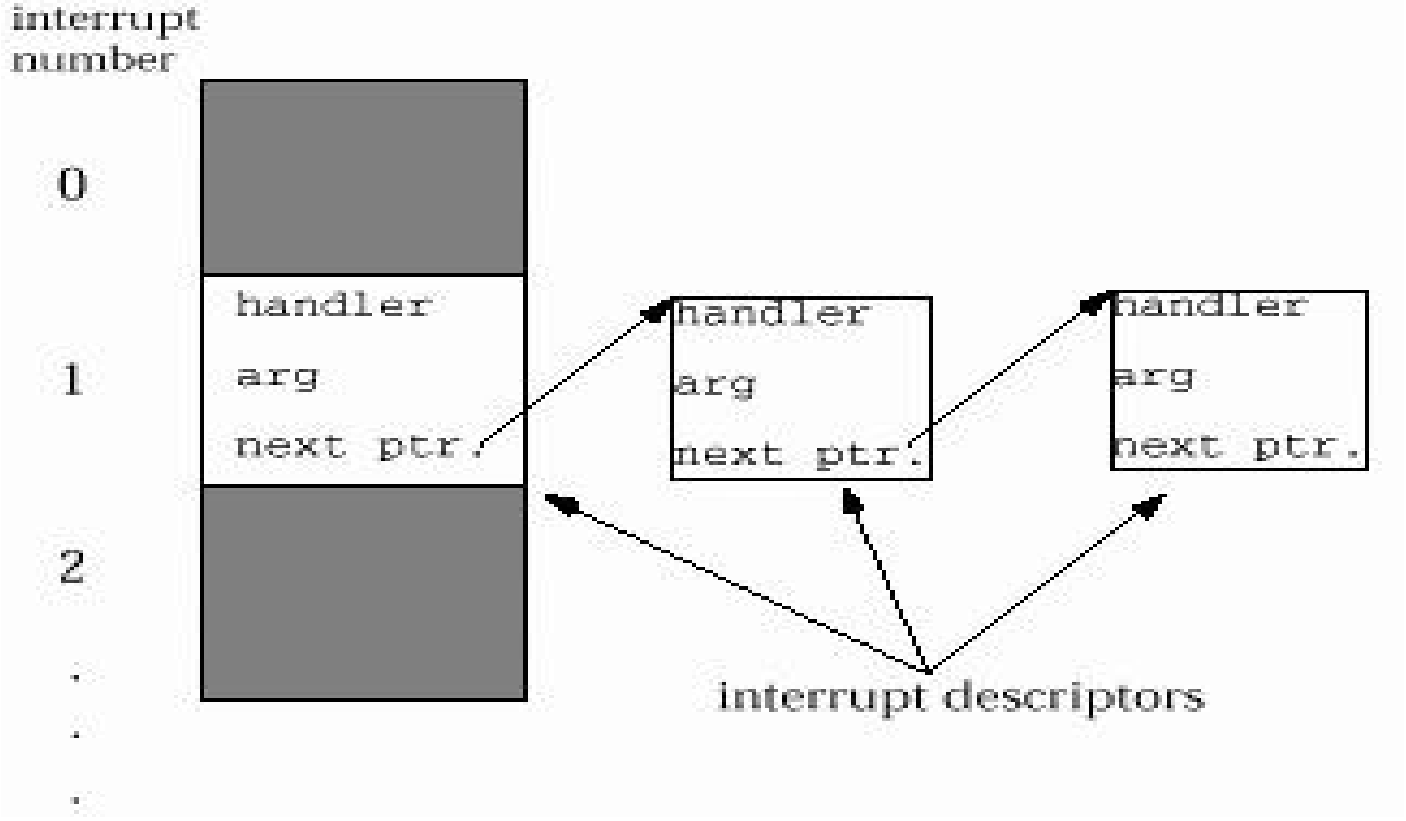
# Interrupt Table - PowerPC

- PowerPC architecture does not have an interrupt vector base register. Default is no support for:
  - intVecBaseSet()
  - intVecSet()
- System interrupt table is a statically declared array of pointers to interrupt handler descriptors. Serves as interrupt vector table.
- An interrupt handler descriptor contains:
  - Address of ISR.
  - The optional integer parameter.
  - Table index. (To label interrupt or manage nested interrupts.)

# Interrupt Table - PowerPC cont.

- The routine sysXIntConnect():
  - Receives same arguments as intConnect().
  - Calls malloc() to obtain memory for an interrupt handler descriptor structure.
  - Initializes interrupt handler descriptor structure.
  - Inserts return value of malloc() in appropriate element of system interrupt table.
- System interrupt table index is the interrupt number:
  - INUM_TO_IVEC(intNum) returns intNum.
- Note, no code to manage hardware or contexts on system interrupt table:
  - Interrupt demultiplexer routine handles these issues.

# Interrupt Table - PowerPC cont.



# Interrupt Demultiplexer - PowerPC

- PowerPC has one external interrupt, when asserted:
  - Processor jumps to (interrupt dispatch) stub code.
  - Saves registers (context) and increments intCnt.
  - Calls interrupt demultiplexer routine.
  - When demultiplexer routine returns, (exit) stub code restores registers (context) and decrements intCnt.
- Interrupt demultiplexer code:
  - Completes interrupt acknowledgment cycle.
  - Gets interrupt vector from interrupt controller.
  - Manages processor interrupt mask.
  - Resets external interrupt line.
  - Calls interrupt handler on system interrupt table using interrupt vector.

# Installing Demultiplexer Routine

STATUS excIntConnect (_EXC_OFF_INTR,
sysXIntHandler)

_EXC_OFF_INTR Interrupt vector to attach to. Variable type: VOIDFUNCPTR pointer.

sysXIntHandler Routine to be called as demultipler routine. Variable type: VOIDFUNCPTR.

- Returns OK or ERROR on error.
- Called by sysXInit() in sysHwInit().
- Interrupt demultiplexer is stored on exception table not on system interrupt table.
- Managed by architecture libraries, BSP not involved.

# Pre-Kernel Debug ISRs - PowerPC

- kernelInit() will unmask the external interrupt line, CPU may receive hardware service requests if interrupt controller or/and devices were not properly initialized in sysHwInit().
- To identify interrupt source use excIntConnect() to replace demultiplexer routine with debug ISR (sysXIntHandler() for PowerPC).
- Interrupt acknowledgment will identify interrupt level activated in interrupt controller:
  - Disable this interrupt level in sysHwInit().
- Note, intVecSet() is not used.

# Managing Interrupts

Overview

Installing ISRs

9.3 Supporting Interrupt Libraries

Initializing An Interrupt Controller

Optional Interrupt Support

# Interrupt Libraries

- There are two user accessible interrupt libraries:
  - intLib - Architecture-independent library.
  - intArchLib - Architecture-dependent library.
- intArchLib is responsible for providing routines to:
  - Enable/disable interrupts.
  - Lock/unlock interrupts.
  - Set interrupt lock level.
  - Creating and installing interrupt handlers.
- If target uses an interrupt controller, BSP code is involved in managing interrupts through intArchLib (e.g. PowerPC, i86, and ARM).
  - Required support is architecture/BSP specific.

# Supporting Interrupt Libraries

- If BSP support is required, will consist of:
  - Initializing interrupt controller.
  - Providing run-time support for controller (e.g. ACK).
- Initializing the interrupt controller is a BSP responsibility which does not involve intArchLib.
- Run-time support will involve intArchLib API:
  - For a given architecture BSP support of intArchLib facilities may vary.
- For all architectures WRS supplies support for:
  - Interrupt dispatch and exit code.
  - intLock()/intUnlock() routines.

# intArchLib Support - PowerPC

- Supported routines are architecture dependent (see Tornado Reference Manual).
- For PowerPC intArchLib uses hook routines to provide functionality through interrupt controller:
  - _func_intConnectRtn()
  - _func_intVecSetRtn()
  - _func_intVecBaseSetRtn()
  - _func_intVecBaseGetRtn()
  - _func_intLevelsetRtn()
  - _func_intEnableRtn()
  - _func_intDisableRtn()
- Hook routines prototyped in ../h/arch/ppc/ivPpc.h.

# intArchLib Support - PowerPC cont.

- Hook routines are called by intArchLib routines.
- Hook routines are initialized by BSP:
  - Hook routines initialized to interrupt controller driver routines.
  - Some hook routines may not be supported.
- Hook routines initialized by sysXInit() which is called by sysHwInit().
  - sysXInit() is part of interrupt controller device driver code.
- Note, no hooks to lock/unlock interrupts:
  - Routines provided as part of PowerPC architecture.
  - Routines written in assembler for speed.

# intArchLib Support - PowerPC cont.

- WRS supplies interrupt dispatch and exit code used as wrapper code for interrupt demultiplexer routine (installed by excIntConnect()).
- Hook routines which must be supported for each interrupt controller by BSP:
  - _func_intConnectRtn()
  - _func_intEnableRtn()
  - _func_intDisableRtn()
- PowerPC can disable its single external interrupt line to lock interrupts, so _func_intLevelSetRtn() does not need to be supported.

# Enabling / Disabling Interrupts

- Hook routines initialized to BSP routines sysXIntEnable()/sysXIntDisable().
- Routines enable and disable a particular interrupt level:
  - Interrupt level passed as (sole) argument.
  - Number of supported levels is a function of interrupt controller and BSP.
  - Single level may identify a specific device.
  - Typically check for legal interrupt level before modifying level.
  - Code in ../src/drv/intrCtl or in BSP directory.

# Multiple Interrupt Controllers - PowerPC

- Each interrupt controller will have its own driver to manage interrupt services for connected devices.
- For multiple interrupt controllers, must determine which controller provides single external exception demultiplexer:
  - Usually host controller.
- Controller which provides external exception demultiplexer registers other interrupt controller demultiplexers on its interrupt table:
  - Interrupt services are cascaded using system interrupt table.

# Managing Interrupts

Overview

Installing ISRs

Supporting Interrupt Libraries

9.4 Initializing An Interrupt Controller

Optional Interrupt Support

# Initializing An Interrupt Controller

- Interrupt controllers may be:
  - On-processor.
  - External devices.
- Differences in initialization involve additional initialization required for off-processor controllers:
  - Initialization of bus(es) and bridges connecting processor and controller.
  - Initialization of additional features of chip, such as interrupt controllers on same chip with bus bridge.
  - Initialization of addition controllers if controllers are cascaded.
- Initialization performed in sysHwInit() and interrupt generation capabilities enabled after kernel is activated.

# Initializing An Interrupt Controller - cont.

- Controller features which may need to be initialized:
  - Interrupt source vector registers.
  - Registers controlling internal interrupts.
  - Mode for interrupt trigger (level or edge sensitive).
  - Clearing pending interrupts and interrupt errors.
  - Base register for interrupt table.
  - Software control structures.
  - Master interrupt control register(s) to disable interrupts until software interrupt handlers are available.
- Create separate interrupt controller initialization routine for each controller.

# Managing Interrupts

Overview

Installing ISRs

Supporting Interrupt Libraries

Initializing An Interrupt Controller

9.5 Optional Interrupt Support

# Optional Interrupt Support

- There are several optional BSP routines. WRS provides API, BSP developer must supply implementation code.
- Provided to support system level operations (e.g. managing an external bus if present).
- Some of these routines support system interrupts.
- Interrupt management support routines are:
  - sysBusIntAck() - Ack. external bus interrupt.
  - sysBusIntGen() - Generate external bus interrupt.
  - sysIntEnable() - Enable external bus interrupt level.
  - sysIntDisable() - Disable external bus interrupt level.

# Bus Interrupt Acknowledgment

int sysBusIntAck (intLevel)

| | |
|---|---|
| intLevel | Interrupt level to acknowledge. Variable type: int. |

- Routine should complete the bus interrupt acknowledgment cycle. Return value is bus specific.
- Provided to respond to external bus interrupts.
- May need to be a dummy routine:
  - Hardware completes interrupt acknowledge cycle for vectored systems.
  - ISR may complete interrupt acknowledge cycle for auto-vectored systems.

# Bus Interrupt Generation

STATUS sysBusIntGen (intLevel, vector)

| | |
|---|---|
| intLevel | Interrupt level to generate. Variable type: int. |
| vector | Interrupt vector to generate. Variable type: int. |

- Generate an external bus interrupt with a specified interrupt vector. Returns ERROR if level is out of range or target cannot generate interrupt.
- Routine details depend on bridge or controller for external bus as well a bus protocol.

# Bus Interrupt Enable / Disable

STATUS sysIntEnable (intLevel)

| | |
|---|---|
| intLevel | Interrupt level to enable. Variable type: int. |

STATUS sysIntDisable (intLevel)

| | |
|---|---|
| intLevel | Interrupt level to disable. Variable type: int. |

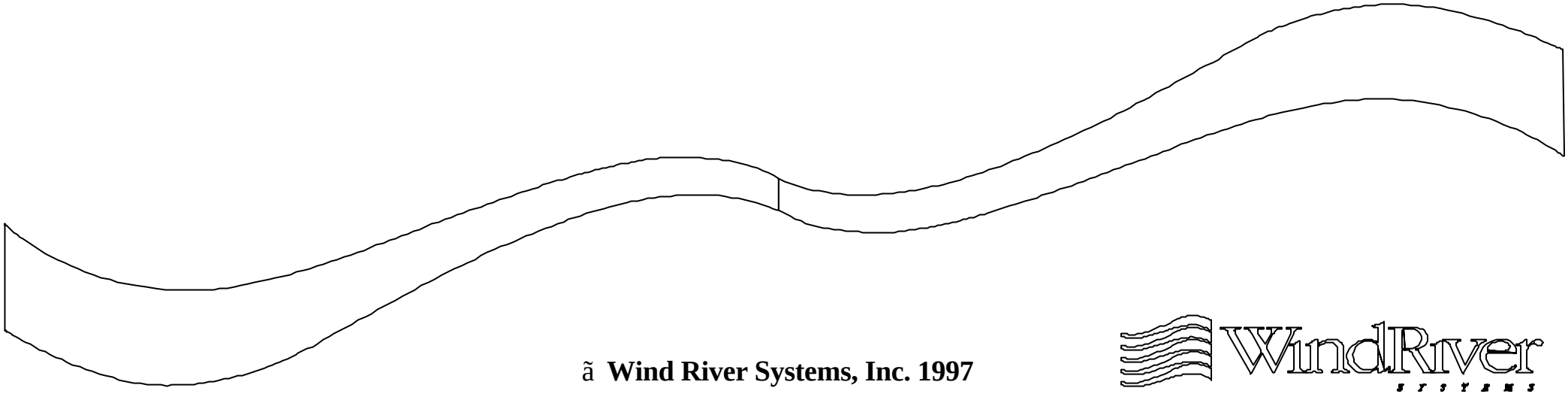
- Enable/disable external bus interrupt. Return ERROR if intLevel is out of range or not supported.

# Summary

- VxWorks uses and interrupt table to supply interrupt level services.
- For some architectures BSP is responsible for managing interrupts. Must supply routines to:
  - Connect interrupts.
  - Enable/disable interrupts.
  - Manage interrupt hardware control registers.
  - Transfer control to boot code if ISR throws exception.
- Details depend on:
  - Presence of external interrupt controller(s) or not.
  - Interrupt policies of local bus(es).
  - Form factor for CPU interrupt support.

# Chapter - 10

## Timers





WindRiver SYSTEMS

# Timers

10.1 Overview

System Clock

Auxiliary Clock

Timestamp

# Overview

- VxWorks uses timers for:
  - System clock.
  - Auxiliary clock.
  - Timestamp.
- VxWorks requires the presence of a single dedicated timer for the system clock.
- BSP designer may support one optional auxiliary clock.
- BSP developer may support one optional timestamp.
- Code for system clock, auxiliary clock, and timestamp are part of driver for appropriate timer:
  - WRS timer drivers in ../src/drv/timer/xxTimer.c.

# System Clock

- System clock support is required, and is a BSP developer's responsibility.
- System clock is initialized and enabled at the start of tUsrRoot after initialization of memory facilities.
- System clock is a software clock provided by an ISR:
  - Highest priority interrupt for system.
  - ISR is usrClock().
- Routine to connect the system clock interrupt:
  - Connects system clock interrupt.
  - Calls routine, sysHwInit2(), to install other system hardware interrupts.

# Connecting Interrupts

void sysHwInit2 (void)

- sysHwInit2() is provided to allow initialization of system devices which are not initialized in sysHwInit().
- Routine's primary responsibility is to install and enable system interrupts:
  - All system (non-generic driver and system clock) interrupts should be installed in this routine.
  - Devices which have generic drivers may enable their interrupts later when device is initialized.
- May also provide configuration for system devices not configured in sysHwInit().

# Auxiliary Clock

- Auxiliary clock support is optional:
  - Used for high (or low) speed polling.
  - Required to use Tornado's spy() routine.
  - BSP may support one auxiliary clock.
  - Cannot share timer with system clock.
- Timer interrupt for auxiliary clock installed in sysHwInit2() by BSP developer:
  - End-user provides routine called by auxiliary clock timer ISR.
- Timer interrupt for auxiliary clock enabled when auxiliary clock is activated.

# Timestamp

- Timestamp support is optional:
  - Implemented as a driver accessed by VxWorks tasks.
  - Driver reads timer register to obtain timestamp.
  - Provides very high fidelity timestamps for WindView or user applications.
  - BSP may support one timestamp.
  - Usually does not share timer with system clock.
- If a timestamp driver requires an interrupt it should be installed in sysHwInit2() and enabled when timestamp is activated.
- Timestamp driver and auxiliary clock may share the same timer, but not at the same time.

# Timer Features

- Hardware timers for system clock, auxiliary clock, and timestamps may be:
  - On processor.
  - Off processor dedicated chip.
  - Off processor ASIC supporting other features (e.g. I/O chip with timer).
- Timers operate in one of three modes. Use of timer is effected by supported mode:
  - Periodic interrupt - Used for system and auxiliary clocks.
  - One-shot interrupt - Currently not supported by WRS BSPs.
  - Timestamp - Used for timestamps.

# Timers

Overview

10.2 System Clock

Auxiliary Clock

Timestamp

# System Clock Support

- Due to generic nature of the system clock, this section will discuss support issues using the WRS template system clock.
- The BSP must provide support for the following user accessible routines:
  - sysClkConnect() - Installs system clock routine.
  - sysClkRateSet() - Sets system clock rate.
  - sysClkRateGet() - Gets system clock rate.
  - sysClkEnable() - Activates system clock.
  - sysClkDisable() - Deactivates system clock.
- BSP must also supply timer ISR for system clock timer. Standard name - sysClkInt().

# Installing The System Clock

- Connecting the system clock consists of:
  - Obtaining and registering name of system clock routine called by system clock timer ISR.
  - Installing system clock timer ISR.
  - Calling sysHwInit2().
- Name of system clock routine:
  - Passed as argument of sysClkConnect().
  - Default is usrClock(). Code in ../all/usrConfig.c.
- System clock timer ISR may be installed explicitly in sysClkConnect() or in sysHwInit2().
- sysClkConnect() will always call sysHwInit2().

# Installing The System Clock - cont.

- Template code:

```
STATUS sysClkConnect
    (
    FUNCPTR routine,
    int arg
    )
    {
    sysHwInit2 ();
    sysClkRoutine = NULL;
    sysClkArg = arg;
    sysClkRoutine = routine;
    return (OK);
    }
```

# System Clock Timer ISR

- This routine will call the system clock routine initialized by sysClkConnect() using the global variables sysClkRoutine and sysClkArg.
- Template code:

```
void sysClkInt (void)
    {
    /* TODO - acknowledge the interrupt if needed */
    /* call system clock service routine */
    if  (sysClkRoutine != NULL)
        (* sysClkRoutine) (sysClkArg);
    }
```

# The System Clock Rate

- Setting the system clock rate consists of specifying the number of system clock ticks (interrupts) per second.
  - The time unit for VxWorks is a system clock tick.
- System clock rate supplied through single argument passed to sysClkRateSet().
  - Default rate is 60 (ticks per second).
  - Clock rate activated when system clock is enabled.
- To obtain the current system clock rate, user calls sysClkRateGet() which returns current rate of system clock.
- Minimum and maximum allowable system clock rates are determined by system clock timer.

# The System Clock Rate - cont.

- Template code for setting the system clock rate:

```
STATUS sysClkRateSet
    (
    int ticksPerSecond
    )
    {
    if  (ticksPerSecond < SYS_CLK_RATE_MIN ||
        ticksPerSecond > SYS_CLK_RATE_MAX)
        return (ERROR);
    sysClkTicksPerSecond = ticksPerSecond;
    if  (sysClkRunning)
        {
        sysClkDisable ();
        sysClkEnable ();
        }
    return (OK);
    }
```

# System Clock Enable/Disable

- Enabling the system clock consists of:
  - Configuring the system clock timer to generate interrupts at a frequency of sysClkTicksPerSecond.
  - Enable system clock timer interrupt.
  - Setting sysClkRunning variable to TRUE.
- In addition, sysClkEnable() should check that a system clock timer ISR has been installed. If not, install one.
  - This check is performed using a local static variable in template routine.
- Disabling the system clock consists of:
  - Disabling system clock timer interrupt.
  - Setting sysClkRunning variable to FALSE.

# Enabling The System Clock

- Template code:

```
void sysClkEnable (void)
    {
    static connected = FALSE;
    if   (!connected)
         {
         /* Connect sysClkInt to interrupt */
         connected = TRUE;
         }
    if   (!sysClkRunning)
         {
         /* Start system timer interrupts */
         sysClkRunning = TRUE;
         }
    }
```

# Testing the System Clock

- System clock can be tested by generating output (e.g. blinking a LED) periodically using taskDelay().
  - Period test several seconds.
  - Use reference clock or count many periods.
- Can modify usrRoot() (use usrConfig.c in BSP directory and modify USRCONFIG macro in Makefile). Example:

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

# Timers

Overview

System Clock

10.3 Auxiliary Clock

Timestamp

# Auxiliary Clock Support

- Auxiliary clock support requirements mirror those of the system clock.
- Functional requirements for support routines identical to corresponding system clock routines.
- The BSP must provide support for the following end-user accessible routines:
  - sysAuxClkConnect() - Installs auxiliary clock routine
  - sysAuxClkRateSet() - Sets auxiliary clock rate.
  - sysAuxClkRateGet() - Gets auxiliary clock rate.
  - sysAuxClkEnable() - Activates auxiliary clock.
  - sysAuxClkDisable() - Deactivates auxiliary clock.
  - sysAuxClkInt() - Auxiliary clock timer ISR.

# Differences From System Clock

- System clock is required, auxiliary clock is not.
- No VxWorks OS overhead associated with auxiliary clock.
- User supplies auxiliary clock routine. No default routine like usrClock() for the system clock.
- User installs auxiliary clock routine with a call to sysAuxClkConnect().
- BSP installs auxiliary clock timer device ISR in sysHwInit2(). Timer device ISR calls sysAuxClkInt().
- Auxiliary clock timer device often has other features in addition to a timer.

# Timers

Overview

System Clock

Auxiliary Clock

10.4 Timestamp

# Supporting Timestamp

- Timestamp supplied by timestamp driver:
  - Some routines mirror system and auxiliary clock routines.
  - Some routines unique to timestamp driver code.
- Timestamp driver works by reading tick count variable on timestamp timer count register. When timer count register rolls over timestamp ISR manages the event.
- Timestamp driver specific routines deal with:
  - Reading timer tick count register.
  - Timer tick count rollover.
- Template timestamp code can be found in ../src/drv/timer/templateTimer.c.

# Timestamp Support Routines

- The BSP must provide support for the following end-user accessible routines:
- Routines mirroring system and auxiliary clocks:
  - sysTimestampConnect() - Installs timestamp ISR.
  - sysTimestampEnable() - Activates timestamp.
  - sysTimestampDisable() - Deactivates timestamp.
- Routines specific to timestamp driver:
  - sysTimestampFreq() - Get timestamp frequency.
  - sysTimestampPeriod() - Get timestamp period.
  - sysTimestamp() - Get timestamp from timer.
  - sysTimestampLock() - Get timestamp from timer with interrupts locked.

# Timestamp Timer Issues

- Primary timer hardware issues impacting timestamp driver design and performance:
  - Read while enabled capability.
  - Existence of prescaler counter.
  - Width of counter register.
  - Preload after disable required.
  - Cache coherency of timer registers.
- Timer read while enable capability allows timer tick counter to be read without stopping timer count. If not supported:
  - Produces time skew which accumulates each time timer count is read.

# Timestamp Timer Issues - cont.

- Time skew must be mitigated possibly using another timer, periodically resetting timer, locking interrupts while reading timer count, etc.
- Degrades real-time performance.

- Prescaler counter divides the input clock frequency to provide lower frequency timestamp.
  - Useful for tuning timer fidelity.
  - Timer resolution = (prescaler) / (input frequency).
  - For an effective WindView timestamp, resolution should be 10 microseconds or less. This requirement arises to ensure all instrumented kernel events will be distinguishable.

# Timestamp Timer Issues - cont.

- Width of timer tick counter register determines maximum count value for timer, which dictates rollover period (referred as period) for timestamp timer:
  - period = (maximum count) X (timer resolution).
  - Performance degrades as period is reduced due to overhead in managing rollover interrupt.
  - Minimum recommend period is at least 10 milliseconds.
- For timers which require the counter to be preloaded with a value before counting a problem arises if timer cannot be read while enabled.
  - Corrects skew, but adds time when timer is disabled, correcting for this can be difficult.

# Timestamp Timer Issues - cont.

- Timer registers must be cache coherent to ensure registers and not just data cache values are accessed:
  - If MMU is not present or enabled, register location must be flushed and invalidated as appropriate for type of cache being used.
  - With MMU make timer registers non-cacheable.
- VxWorks kernel instrumentation for use with WindView requires the following timestamp timer features:
  - Capability to generate rollover interrupt.
  - Resolution of 10 microseconds or less.
  - Period of 10 milliseconds or more.

# Timestamp Period and Frequency

- Timestamp timer frequency and period will configured when the timer is unitized:
  - Routine sysXInt() where X refers to the name of the timer chip.
  - Usually done in sysHwInit2() as part of timestamp device initialization.
- Timestamp frequency (in Hertz) defined in xxTimer.c.
  - For template timestamp:

    #define TIMESTAMP_HZ 1000000
- Timestamp period expressed in system clock ticks.

# Frequency and Period Routines

- Template routine to get timestamp frequency:

```
UINT32 sysTimestampFreq (void)
    {
    /* When possible read timer register(s). */
    return (TIMESTAMP_HZ);
    }
```

- Template routine to get timestamp period:

```
UINT32 sysTimestampPeriod (void)
    {
    /* When possible read timer registers(s). */
    sysTimestampPeriodValue = TIMESTAMP_HZ /
                                    sysClkTicksPerSecond;
    return (sysTimestampPeriodValue);
    }
```

# Timestamp

- The timestamp routine reads the timestamp timer tick register. May also need to:
  - Preload counter.
  - Invalidate data cache.
  - Convert to seconds by dividing count by sysTimestampFreq().
- Template timestamp routine:

```
UINT32 sysTimestamp (void)
    {
    UINT32 count = 0;
    /* Read the timestamp timer value */
    return (count);
    }
```

# Timestamp - cont.

- Template routine to get timestamp timer tick count for timer which cannot be read while enabled:

```
UINT32 sysTimestampLock (void)
    {
    UINT32 result;
    int oldLevel;
    oldLevel = intLock ();
    result = sysTimestamp ();
    intUnlock (oldLevel);
    return (result);
    }
```

- May also need to correct for time skew.

# Timestamps and the System Clock

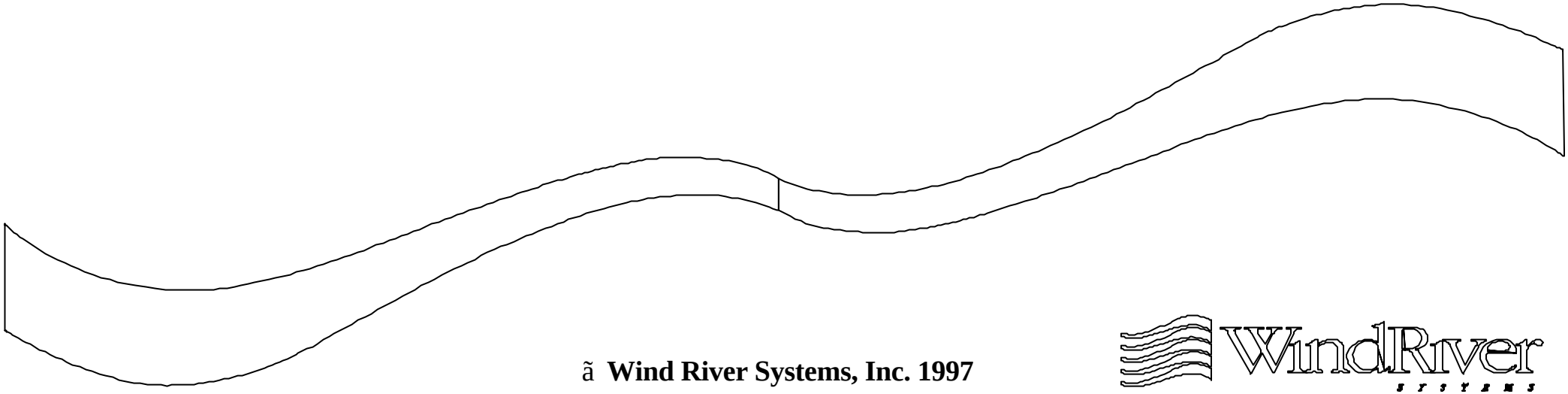
- Timestamps can also be obtained from the system clock with some caveats:
  - Timer must support read while enabled.
  - Timestamp driver must not have rollover ISR which will interfere with system clock ISR. Timer must be monitored for counter rollovers.
  - Timestamp driver should not reset counter.
  - System clock should set timer period.
  - sysClkRateSet() should not be callable while using timestamp driver.
- In general timestamp driver and system clock will be needed in different frequency regimes and will require separate timers.

# Summary

- BSP support for VxWorks timers - use template code:
  - System clock.
  - Auxiliary clock.
  - Timestamp.
- VxWorks requires the presence of a single dedicated timer for the system clock.
  - Installation routine for system clock also calls sysHwInt2() which completes initialization of system devices started in sysHwInit().
- Optional auxiliary clock used for high speed polling.
- BSP support routines details for timestamp depend on nature of timer used and cache configuration.

# Chapter - 11

## Completing The BSP





WindRiver SYSTEMS

# Completing The BSP

11.1 Overview

Remaining BSP Routines

Device Driver Issues

Final BSP Files

Validation Test Suite

# Finishing the BSP

- Once initial VxWorks image kernel is successfully activated and system clock is enabled, developer becomes involved in BSP completion activities.
- Primary BSP completion activities:
  - Ensuring all required BSP routines are present.
  - Providing support for appropriate optional BSP routines (e.g. auxiliary clock routines).
  - Obtaining and integrating generic device drivers.
  - Clean-up of code content and location.
  - Final configuration of system features.
  - Documentation.
  - Validation testing.

# BSP Development Cycle

- Milestones for BSP development:
  - Prepare development environment, obtain reference BSP, BSP template files, and choose appropriate initial VxWorks image type.
  - Make necessary modifications to reference romInit.s.
  - Test romInit.s performance and romStart() configuration parameters using development tools or/and LEDs. Need to get to usrInit().
  - Write and test sysHwInit() and necessary pre-kernel BSP support libraries. Need to activate kernel.
  - Optionally develop polled mode serial driver to provide Tornado access.
  - Develop and install system clock and system ISRs.
  - Perform BSP completion activities.

# Final System Configuration

- Portions of the BSP completion phase may involve interfacing with non-BSP developers suppling driver code for target devices:
  - Driver groups will require integration support.
  - BSP developer(s) will require driver information for documentation.
- BSP developer should provide all code and documentation required by generic device driver developers to successfully:
  - Integrate code.
  - Initialize generic device.
  - Support runtime driver operation.

# Final System Configuration - cont.

- BSP developer should be familiar with generic device driver issues:
  - Location of driver code.
  - Location of driver configuration and control parameters.
  - Interrupt assignments for both generic and BSP specific devices.
  - Bus access issues for generic driver.
- BSP developer(s) should obtain device and driver documentation from driver developer and add to BSP documentation files.
  - Generic driver developer is responsible for documentation in driver code files.

# System Feature Configuration

- If there are non-standard hardware configurations which were used during development (e.g. jumper controlling boot from socketed ROM or Flash):
  - Hardware should be reconfigured.
  - Documentation should clearly state which configurations are and are not supported.
- Build ROM(s) if appropriate.
- Provide support for additional timers (if appropriate):
  - Auxiliary clock.
  - Timestamp.
- Provide support for external buses (if appropriate).

# VxWorks Images

- Should be able to build all VxWorks images supported in ../h/make/rules.bsp.
  - Test that images can be built and booted.
- Conversion from development image should be almost transparent:
  - Will need to convert romInit.s to sysALib.s for loadable images.
  - Configure bootrom images after required driver(s) (network, serial, SCSI) are available.
  - Provide NVRAM support for loadable images, and place default BSP boot parameters in config.h.
  - Makefile in BSP directory appropriately modified (will be discussed in upcoming section).

# Boot ROM Images

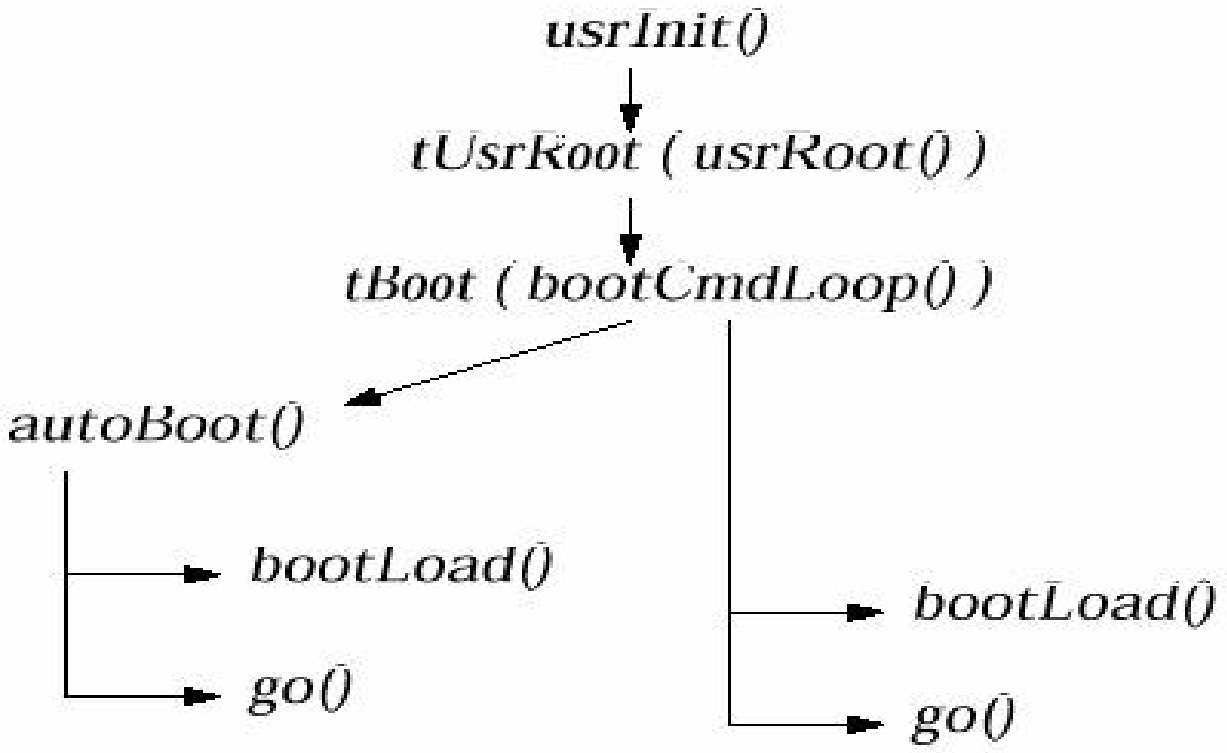
- Boot ROM images relocate a loadable VxWorks image using a network interface, serial line, SCSI disk, etc.
  - Boot parameters stored in NVRAM.
  - Boot parameters may be modified at boot time.
- For boot ROM images, bootConfig.c replaces usrConfig.c:
  - usrRoot() will spawn a task, tBoot, which will relocate loadable image.
  - File contains support routines to manage boot parameters, image relocations, and user interaction.
- BSP developer will not need to modify bootConfig.c unless custom boot path is to be supported.

# Boot ROM Images - cont.

- Entry point for task tBoot is bootCmdLoop():
  - Will autoboot image based on boot parameters in NVRAM after timeout expires with no user interaction by calling autoboot().
  - For interactive session, code loops in interactive session until boot continuation command is issued.
- Routine which relocates loadable image is bootLoad():
  - Boot parameters passed in through argument list.
  - Identifies appropriate device interface for load.
  - Performs any necessary initialization on load device.
  - Loads image.
- When bootLoad() returns processor is jumped to the entry point of the loaded image, sysInit().

# Boot ROM Images - cont.

- Sequence of events after romStart() jumps to usrInit():



# Completing The BSP

Overview

11.2 Remaining BSP Routines

Device Driver Issues

Final BSP Files

Validation Test Suite

# Required BSP Routines

- The following routines must be in the sysLib.o module. If not, an unresolved global error will be generated during linking of VxWorks:
  - sysBspRev() - Return BSP version and revision number, in sysLib.c.
  - sysClkConnect() - Connect a routine to the system clock ISR, in ../src/drv/timer/xxTimer.c.
  - sysClkDisable() - Disable system clock timer interrupts, in ../src/drv/timer/xxTimer.c.
  - sysClkEnable() - Enable system clock timer interrupts, in ../src/drv/timer/xxTimer.c.
  - sysClkInt() - Handler for system clock timer interrupt, int ../src/drv/timer/xxTimer.c.

# Required BSP Routines - cont.

- sysClkRateGet() - Get system clock rate, in ../src/drv/timer/xxTimer.c.
- sysClkRateSet() - Set system clock rate, in ../src/drv/timer/xxTimer.c.
- sysHwInit() - Initialization of system hardware before kernel activation, in sysLib.c.
- sysHwInit2() - Initialization of system hardware after kernel activation, in sysLib.c.
- sysMemTop() - Return the address of the top of the system memory pool, in sysLib.c.
- sysModel() - Return model name for target environment, in sysLib.c.
- sysNvRamGet() - Get the contents of NVRAM, in ../src/drv/mem/xxNvRam.c.

# Required BSP Routines - cont.

- sysNvRamSet() - Set the contents of NVRAM, in ../src/drv/mem/xxNvRam.c.
- sysSerialHwInit() - Initialized serial devices to quiet state prior to kernel activation, in sysSerial.c.
- sysSerialHwInit2() - Connect serial device ISRs after kernel activation, sysSerial.c
- sysSerialChanGet() - Get the address of a SIO_CHAN structure associated with a serial channel, in sysSerial.c.
- sysToMonitor() - Transfer control to the ROM/Flash monitor, in sysLib.c.

- Routines not discussed previously:
  - sysBspRev().
  - sysModel().

# Release Numbers

char * sysBspRev (void)

- Routine returns pointer to BSP version/revision number string. Values defined as macros in config.h:
  - BSP_VERSION
  - BSP_REV
- Combination of version and revision numbers are the release numbers for the BSP.
  - Version number identifies BSP generation.
  - Revision number is an incrementing number identifying release within a generation, Should begin with zero for first release.

# BSP Model Name

char * sysModel (void)

- Returns pointer to model name for BSP:
- char *sysModel (void)

```
{
return (SYS_MODEL);
}
```

  - SYS_MODEL defined in <bsp>.h or sysLib.c.
- Model name string printed to standard output by tUsrRoot as part of WDB agent banner.
- Routine resides in sysLib.c.

# Optional BSP Routines

- The following routines are optional but are usually present as part of a BSP:
  - sysAuxClkConnect() - Connect a routine to the auxiliary clock ISR, in ../src/drv/timer/xxTimer.c.
  - sysAuxClkDisable() - Disable the auxiliary clock timer interrupt, in ../src/drv/timer/xxTimer.c.
  - sysAuxClkEnable() - Enable the auxiliary clock timer interrupt, in ../src/drv/timer/xxTimer.c.
  - sysAuxClkInt() - Handler for auxiliary clock timer interrupt, in ../src/drv/timer/xxTimer.c.
  - sysAuxClkRateGet() - Get auxiliary clock rate, in ../src/drv/timer/xxTimer.c.
  - sysAuxClkRateSet() - Set auxiliary clock rate, in ../src/drv/timer/xxTimer.c.

# Optional BSP Routines - cont.

- sysPhysMemTop() - Get the top of physical memory for the target, in sysLib.c.

- These routines have all been discussed previously.
- There is also a set of optional routines for managing system busses if present. Some of these routines have been discussed previously, a complete list appears in the Tornado BSP Developer's Kit for VxWorks User's Guide.
  - There is also a macro, BUS_TYPE defined in ../h/ vxWorks.h to identify system bus(es).
- Any other custom BSP specific routines should placed in sysLib.c or the appropriate file in the BSP directory or sub-directory.

# Completing The BSP

Overview

Remaining BSP Routines

11.3 Device Driver Issues

Final BSP Files

Validation Test Suite

# Device Drivers and the BSP

- BSP developer is not responsible for writing BSP independent device drivers, however, BSP developer is responsible for integration of these device drivers.
- BSP developer should be aware of driver issues:
  - Location of files.
  - Configuration parameters.
  - Design strategies.
  - Driver structure.
- In addition BSP developer should provide sufficient documentation for device driver writers as well as end-users.

# Location of Device Driver Code

- All device driver code not developed by WRS should be placed in:
  - BSP directory.
  - A sub-directory of the BSP directory.
  - Developer created directory directly below ../config.
- Popular configuration - BSP drivers in BSP directory, generic driver code in subdirectory of BSP directory.
- WRS places it's generic device driver code in ../src/drv/ xxxx and ../h/drv/xxxx.
  - These device drivers can be used with any BSP.
  - WRS reserves all rights with respect to these directories.

# config.h

- End-user modifiable BSP configuration file will contain some generic driver configuration and control variables in addition to specific BSP parameters:
- General BSP control parameters in config.h:
  - Default boot parameters.
  - Local memory configuration.
  - Off-target memory (e.g. VME bus window sizes).
  - Interrupt controller configuration bit patterns.
  - Cache and MMU options (e.g. L2 cache support).
  - Bus configuration (e.g PCI bus numbers).
  - Shared memory network definitions.
- config.h contains #include of <bsp>.h.

# config.h - cont.

- Generic driver related parameters in config.h:
  - Device support macros.
  - Number of supported serial channels.
  - Auxiliary clock timer device ID (if supported).
  - Network device support (e.g. link buffer pool size).
  - SCSI device support (e.g. fast and wide).
- Device specific, target independent configuration and control parameters should be placed in the xxDrv.h header file.
- Generic device parameters appearing in config.h associated with device features which would change if device was installed in different target environment.

# <bsp>.h

- <bsp>.h contains parameters which are not configurable by the end-user:
  - Device base addresses (bridges, controllers, etc.).
  - Register offsets system devices.
  - BSP device control bit patterns.
  - System and auxiliary clock minimum and maximum rates.
  - Local bus maps and bus speeds.
  - External bus maps.
  - System interrupt vectors and levels.
- After completion of BSP, parameters in <bsp>.h should only be changed as part of modification of BSP design.

# Driver Design Strategies

- These guidelines are valid for both BSP dependent and independent device drivers.
- Primary design problems involve:
  - Device hardware.
  - Performance enhancement.
  - Code re-entrancy.
  - Configuration and portability.
- There will be trade-offs associated with attempting to resolve these design problems.
- Some design issues will require an understanding of the VxWorks OS, and how driver code is accessed by VxWorks.

# Design and Device Hardware

- For multi-function chips providing more than one device on a single ASIC the primary goal should be scalability:
  - Do not write one driver for ASIC.
  - Write separate driver for each device.
  - If devices must share ASIC resources or one driver needs to support an other, clearly document dependences in dependent driver code.
- Chip may be accessed as memory mapped device or I/O mapped device:
  - Access memory mapped devices via access macros.
  - For I/O mapped devices assembly routines will be required to reach I/O, direct C expression will not.

# Driver Performance

- Optimizing device driver performance will require some knowledge of the VxWorks real-time kernel operation.
  - Tornado Training Workshop or Tornado Device Driver Workshop provide more relevant information.
- Designing for performance will involve:
  - Using device DMA capabilities if present.
  - Minimizing interrupt latency.
  - Minimizing subroutine nesting.
  - Developing an appropriate cache strategy.
  - Assignment of priorities for tasks accessing driver code.
  - Providing mutual exclusion and synchronization.

# Driver Code Re-entrancy

- Device driver code will be called in the context of tasks in VxWorks.
  - Re-entrancy necessary to prevent race conditions.
- Driver code may manage multiple instances of a device (a serial driver may control several channels), must be reentrant to prevent device “cross-talk”:
  - Each device should have a separate control structure for the device and pass it to driver routines for initialization and device control.
  - Device control structure obtained using malloc() when device is initialized or as an array of device control structures with a separate element for each device instance.

# Device Configuration and Code Portability

- Driver should not inhibit device feature access:
  - Driver code should provide end user with flexible user-friendly configuration interface.
  - If all features of device will not be supported, design should not preclude support at a later time.
- Portability supported through use of configuration and hardware access macros:
  - Register addresses.
  - Control and status inquiry parameters.
  - Hardware management access macros.
- Portability also improved by standard driver interface.

# Overview Of Driver Structure

- WRS has established general guidelines for device driver structure:
  - For both standard and non-standard drivers.
  - For BSP independent and BSP dependent devices.
- General guidelines for the structure of a device driver to be used with WRS products:
  - Object-oriented design for device control structures.
  - Device register access through macros.
  - Driver does not connect ISRs.
  - Comprehensive driver documentation.
- More information in Tornado Device Driver Workshop.

# Device Control Structures

- Each device should be represented by a single control structure containing all state information associated with the device:
  - Structure created when device is created with xxDevCreate() routine.
  - xxDevCreate() routine should check that device is present and that appropriate driver code to manage device is present.
- Device control structure should contain methods (routines) to manage the device structure.
- If device has multiple channels, each channel should have its own control structure.

# Example Device Control Structure

- This example (template) device is serial chip with two channels (ports A and B). So there will be control structures for:
  - SCC chip.
  - Each channel for chip.
- Control structure for template serial chip (device):

```
typedef struct
        {
        TEMPLATE_CHAN portA;
        TEMPLATE_CHAN portB;
        volatile char * masterCr;
        } TEMPLATE_DUSART;
```

# Example Device Control Structure - cont.

- Per-channel control structure:

```
typedef struct
    {
    SIO_CHAN        sio; /* SIO_CHAN element */

    /* callbacks */

    STATUS          (*getTxChar) ();
    STATUS          (*putRcvChar) ();
    void *          getTxArg;
    void *          putRcvArg;
```

# Example Device Control Structure - cont.

```
    /* register addresses */

    volatile char *         cr;     /* control register */
    volatile char *         dr;     /* data register */
    volatile char *         sr;     /* status register */
    volatile char *         ms;     /* modem status */
    volatile char *         mc;     /* modem control */
    volatile short *        br;     /* baud constant */

    /* misc */

    int                     mode;       /* current mode */
    int                     baudFreq;   /* clock frequency */
    int                     options;    /* Hardware options */
    }   TEMPLATE_CHAN;
```

# Access Macros

- Access to hardware registers should be made through macros:
  - Code to read and modify chip registers.
  - Chip register address definitions.
- Usually access methods will require read, write, and perhaps bit modification macros:

  WIDGET_READ (adrs, pData)

  WIDGET_WRITE (adrs, pData)

  WIDGET_CLR_SET (adrs, clear-bits, set_bits)
- Return values by reference through macro routine argument list, not as a routine return value.

# Access Macros - cont.

- Minimize use of conditional statements within code blocks. Example:

```
#ifdef INCLUDE_MY_WIDGET
    widgetReset (&myWidget,
#else
    widgetReset (&stdWidget,
#endif
                        arg1, arg2)
```

- A better approach:

```
#ifdef INCLUDE_MY_WIDGET
# define THE_WIDGET myWidget;
#else
# define THE_WIDGET stdWidget;
#endif
....
widgetReset(&THE_WIDGET, arg1, arg2);
```

# Access Macros - cont.

- Do not use structures which map hardware registers, code will not be portable (even using the same toolchain across architectures):

```
typedef struct {
        char WIDGET_CR;
        char WIDGET_SR;
        char WIDGET_DATA:
        }MY_WIDGET
```

- Use macros to convert a base address to a register address, and to specify offsets for registers:

```
#define WIDGET_ADRS(reg) (WIDGET_BASE_ADRS + reg);
....
#define WIDGET_CR WIDGET_ADRS(WIDGET_CR_OFFSET);
....
#define WIDGET_CR_OFFSET 0x020447
```

# Device Driver ISRs

- ISR code will be in file with other driver code but driver should not call intConnect() for ISR:
  - Interrupts connected by BSP in sysHwInit2().
  - Use INCLUDE_XXX macros to conditionally connectISRs if device is being supported.
- Driver will enable hardware interrupt when ready.
- Device driver ISRs should exit immediately if device hardware is not asserting and interrupt:
  - Do not assume one-to-one mapping of interrupt vectors and handlers. Interrupt lines may be multiplexed.
  - Driver ISR should check device to determine if it is generating request for interrupt service.

# Device Driver Documentation

- Documentation should provide comprehensive description of hardware features:
  - Clearly identify which features are and are not supported.
  - Should be an introduction to both the chip and the device driver.
- Documentation placed in file with driver code, and can be included in Tornado man pages for users.
- Device driver writer and BSP developer will need to:
  - Develop documentation detailing how driver is integrated into BSP (in xxDrv.c).
  - Develop documentation for BSP description in target.txt.

# Device Driver Development

- General strategy is perform-top down design, and bottom-up implementation and testing.
- Top-down design sequence:
  - Start with template file, if there is no template file use existing driver.
  - Obtain hardware manual and write documentation describing device and driver support.
  - Define per-device (and driver) control structures.
  - Define and document access and configuration macros for driver.
  - Declare all routines which BSP developer and users will need to use this driver.
  - Create declarations for driver routines and comment.

# Device Driver Development - cont.

- Bottom-up implementation and testing sequence:
  - Write device initialization code. Configures hardware registers, enables/disables device interrupts, and installs routines on driver table for standard drivers.
  - Write device control structure initialization routine which accepts a device control structure and initializes it.
  - Complete remainder of driver routines, perform compile checks, and verify that only desired external routines are unresolved.
  - Run-time test, debug, and re-compile as dictated by project requirements.
  - Performance test. Benchmark driver using industry standards if available.

# Completing The BSP

Overview

Remaining BSP Routines

Device Driver Issues

11.4 Final BSP Files

Validation Test Suite

# Final BSP Files

- BSP files were listed and described earlier. Part of completing a BSP is ensuring the BSP files which the BSP developer modifies are in their proper final form:
  - Source files.
  - Include files.
  - Makefiles.
  - Derived files.
  - Document files.
- All files which developer modifies (or creates) are in BSP directory, BSP subdirectory, or directory which developer has created directly below ../config.

# Source Files

- Source files relevant to BSP development:
  - Developer created files.
  - Modified reference BSP files.
  - Modified WRS supplied generic driver routines.
  - Modified template files.
  - Third party supplied files.
- All these files belong in one of the following directories:
  - BSP directory (../config/<bspName>).
  - Sub-directory of BSP directory.
  - Developer created sub-directory under config.
- Popular configuration - Non-WRS generic device drivers in BSP subdirectory.

# sysALib.s

- Will need to provide sysALib.s for loadable VxWorks images (if not done previously):
  - Cut and paste romInit.s into sysALib.s.
  - Change routine name from romInit() to sysInit().
  - Remove any memory initialization code.
  - Modify routine to jump to usrInit() (not romStart()).
- Default sysALib.s may contain code which will be lost if romInit.s is copied into (overwrites) sysALib.s:
  - Example - system bus access routines written in assembler.
  - sysALib.o (and sysLib.o) included in all VxWorks builds through macro MACH_DEP defined in ../h/ defs.bsp.

# Include Files

- BSP include files, <bsp>.h and config.h, are modified during development.
  - May need to provide macro definitions and documentation for extendable hardware features which are not currently supported.
- Check that variables are in appropriate include file.
  - <bsp>.h contains target specific information which is not configurable by the end-user.
  - config.h is end-user configurable. Contains BSP specific parameters and (un)definitions of configAll.h macros.
- Include files for non-WRS drivers will appear in directory with driver source code (generic and BSP).

# Makefiles

- Makefiles should only need to be modified for:
  - Clean-up of development Makefile configurations.
  - Builds of non-WRS device driver code.
  - Custom VxWorks builds.
- Remove or redefine any BSP Makefile macros which have been configured specifically for BSP development.

  Examples:
  - If private copies of bootInit.c, usrConfig.c or/and configAll.h were used for development, reinstall generic versions by removing definitions of,

    BOOTINIT, USRCONFIG, and/or CONFIG_ALL.
  - USR_ENTRY set equal to usrInit.
  - Modify EXTRA_XXXX components if necessary.

# Makefiles - cont.6

- BSP independent driver code not supplied by WRS should reside in a BSP sub-directory.
  - Makefile in this sub-directory specifies rules to build driver object module(s) and place them in BSP directory if no #include in sysLib.c.
  - Driver target make is performed by Makefile in BSP directory. These make commands need to be added.
- BSP specific driver code should reside in BSP directory, and there should be a #include in sysLib.c.
- If non-supported VxWorks image type is required supply build rules and necessary definitions in Makefile residing in BSP directory.

# Derived and Documentation Files

- The only “non-standard” derived files will be object modules for device drivers (generic and BSP) not supplied by WRS and with no #include in sysLib.c.
- In addition to documentation provided in source files, BSP must supply documentation for:
  - target.txt - BSP developer is responsible for this file, may need contributions from BSP independent device driver writers or documentation
  - README - Provide detailed release record information, pay particular attention to any caveats which user must be aware of to use VxWorks with this BSP. If modifying an existing BSP, update this file, and provide any known information concerning SPRs filed against the BSP (if relevant).

# Completing The BSP

Overview

Remaining BSP Routines

Device Driver Issues

Final BSP Files

11.5 Validation Test Suite

# Validation Test Suite Overview

- Validation Test Suite (VTS) designed to provide report(s) allowing analysis of basic BSP functionality:
  - Auxiliary clock functionality.
  - System clock functionality.
  - Serial communication at supported baud rates.
  - Commands executable from VxWorks boot prompt.
  - RAM read operations.
  - ROM read operations
  - Local and external bus access.
  - Reboots and catastrophic error recovery.
  - NVRAM access.
  - Networking facilities.
  - SCSI read and write operations.

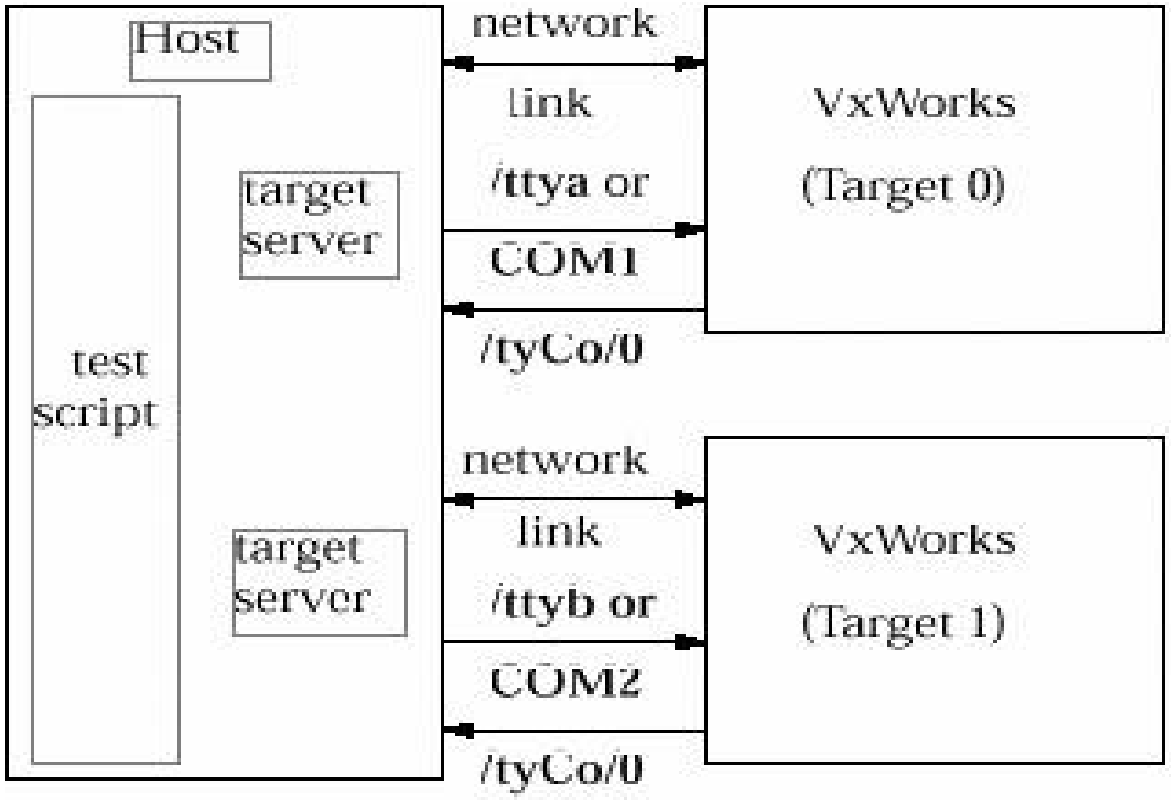
# Validation Test Suite Overview -cont.

- Test host (UNIX or Windows) controls execution of tests of BSP functionality on target.
  - All VTS facilities are host initiated.
- VTS can run with single or multiple VxWorks targets.
- Minimum requirements:
  - Tornado.
  - Appropriate host tools (host compilers, etc.)
  - Complete BSP Developers Kit (correct release level).
  - Complete source for BSP to be tested.
  - Network or serial target/host connection.

# VTS Structure

- VTS only runs in Tornado environment using Tcl/WTX scripts to perform BSP tests:
  - If basic tests are inadequate, source code may be modified to provide needed tests.
- Tests activated with single command:
  - Script attaches target server to first VxWorks target (Target 0) which uses network link.
  - If serial communication is required channel ttya (UNIX) or COM1 (Windows) opened to Target 0's / tyCo/0 serial port.
  - Connects subsequent target servers to ttyx (UNIX) or COMx to Target N's /tyCo/N serial port.
  - Host interacts with target using WTX and windSh commands.

# VTS Configuration



# VTS Files

- Configuration parameters controlling VTS for a single target reside in ../host/resource/test/bspVal/ <bspName>.T1.
  - Boot parameters.
  - Serial device and baud rate for console connection.
  - Timeout for loading VxWorks and other files.
  - Parameters for desired functional tests.
- Take default file <xxxx>.T1 in ../host/resource/test/ bspVal directory and copy it into <bspName>.T1.
  - Modify file as required. See documentation for test control parameters.
  - For additional targets files are <bspName>.TN (N = 1,2,...).

# VTS Files - cont.

- VTS script source files are located in directories:
  - ../host/src/test/bspVal/src/test
  - ../host/src/test/bspVal/src/lib
- Most test script files are located in ../host/src/test/ bspVal/src/tests, however if procedure is shared by multiple test the code is stored in ../host/src/test/ bspVal/src/lib.
  - Example - tests for system clock is stored in ../test/ sysClock.tcl.
  - Example - procedure to reboot VxWorks in ../host/ src/test/bspVal/src/lib.
- Complete source for Tcl available.

# Running the VTS

- BSP VTS is activated with a test script (see documentation for details for UNIX and Windows platforms).
- All output goes to standard output (host platform) by default, must use logfile for permanent storage:
  - Header with name of target server, BSP, and logfile.
  - As test begins, test name is displayed.
  - As sub-tests complete, name is displayed along with PASS/FAIL status information.
- Normally if test fails, other test will still run.
  - If fatal error is generated, VTS aborts and displays a FATAL ERROR message.

# BSP Validation

- Goal of validation to ensure integrity of BSP. WRS supplies a BSP validation checklist:
  - Product Packaging Test.
  - BSP VTS checklist.
  - Target Information test sheet.
- To have a BSP certified by WRS, documents associated with these test, along with complete BSP source, and two targets must be provided to WRS:
  - WRS will test all aspects of the BSP - installation, packaging (correct file organization and content), and functionality (see BSP VTS checklist).
  - WRS certified BSP products can be distributed displaying that they are WRS certified.

# Summary

- After development image activates kernel and system clock, developer enters final stage of BSP development:
  - Provide any missing required BSP routines.
  - Integrate generic driver code.
  - Add support for missing optional BSP routines.
  - Clean-up and complete BSP files.
  - Ensure all standard VxWorks images can be booted.
  - Test (use VTS if available) and document BSP.
- BSP development can be an on-going project as new drivers are added, and new features are supported.
  - Documentation is the key to maintaining continuity.
- Contact WRS if BSP certification is desired.