WIND RIVER

VxWorks®
6.0

HARDWARE CONSIDERATIONS GUIDE

Corporate Headquarters
Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): (800) 545-WIND
telephone: (510) 748-4100
facsimile: (510) 749-2010

For additional contact information, please visit the Wind River URL:

http://www.windriver.com

For information on how to contact Customer Support, please visit the following URL:

http://www.windriver.com/support

VxWorks Hardware Considerations Guide, 6.0

10 Nov 04
Part #: DOC-15304-ZD-01

Contents

VxWorks

Hardware Considerations Guide

6.0

1. Introduction

VxWorks runs on many architectures and targets from a wide variety of vendors, including custom and embedded hardware, VMEbus, Multibus, and PCIbus single-board computers, and workstation and PC mother boards. VxWorks can also boot from many different UNIX and Windows hosts using a variety of communication media.

With the number of combinations that have been configured, Wind River has gathered sufficient experience to make recommendations in board design to best suit the VxWorks run-time environment. However, this document should not be used to determine the desirability of potential or existing ports. Many considerations essential to such determinations, such as cost analysis, time to market, and power and cooling requirements, are beyond the scope of this document.

This document enumerates run-time functional requirements, distills features of important run-time components, and warns against potential pitfalls. The primary objective of this document is to assist developers in selecting or designing appropriate boards and components for VxWorks operation. The following issues are discussed in this document:

- minimum system requirements
- processor and architectural considerations
- memory
- bus
- devices

The particulars of how an individual architecture implements these considerations are discussed in the *VxWorks Architecture Supplement* document for each architecture. This document is a general discussion of the issues and not an implementation-specific guide.

NOTE: All file and directory paths in this manual are relative to the VxWorks installation directory. For installations based on VxWorks 5.x, this corresponds to *installDir*. For installations based on VxWorks 6.0, this corresponds to *installDir***/vxworks-6.0/**.

2. Minimum System Requirements

It is recommended that the minimum system include the following features:

- 32-bit processor
- JTAG processor debug capability, or equivalent
- interrupt controller
- RAM and RAM controller (see below for minimum RAM size)
- ROM (see below for minimum ROM size)
- MMU
- software controlled LEDs
- system clock
- high resolution timestamp counter
- two serial ports
- Ethernet controller

Although the above is the recommended minimum, VxWorks will run with considerably less than specified above. Only a minimum set of features is absolutely required for a given board to run VxWorks. This list is short: a 32-bit processor, an interrupt controller of some kind, some kind of timer or clock, RAM and RAM controller, and enough non-volatile memory to hold a boot image. Although this list specifies a 32-bit processor, some 64-bit processors can be used instead.

Note that it is not absolutely necessary to have a serial port, Ethernet device, or other means of communication between the target board and the development host. However, to avoid serious application development delays, it is best to include one or two serial ports, at least one Ethernet port, and, if the processor supports such a capability, some kind of processor debug port such as JTAG.

A bank of software controlled LEDs is also a desirable feature for the board. During development, they can indicate the state of the system relevant to the application being developed. And for product deployment, they can indicate system status.

For more effective debugging, it is helpful to have a high resolution timer other than the system clock, to be configured as a timestamp device. It is also usually appropriate to include support for an auxiliary clock.

Note that it is also not absolutely required that the system have an MMU. However, in cases where no MMU is present, some system capabilities are not available, such as write protection of the text segment and the use of sentinel pages to help protect the task stacks and other important data structures. When an MMU is available on a processor, it must be used.

3. Processor and Architecture Support

VxWorks supports a variety of processor architectures and many specific processors. A list of standard board support packages (BSPs) is available from the Wind River Web site at:

http://www.windriver.com/products/bsp_web/bsp_architecture.html

One of these BSPs can be used as a reference when developing your own BSP. For VxWorks to be used as the OS for your project, it is helpful to design the hardware similar to the hardware used by one of the BSPs from these pages.

In addition, it is possible to get a list of processors supported by VxWorks from the Wind River Online Support Web site.

Care should be taken to insure that the processor has the functionality that is required and that VxWorks libraries support that functionality on that processor. Many chips are designed without certain features, or with reduced capabilities for some features. Even if the processor is supported, this can make them less desirable with VxWorks libraries in some respects. For more information, see *10. Partial Compliance to Standards*, p.31.

4. Architecture Considerations

At the core of any VxWorks run-time environment is the target architecture. This section is dedicated to the capabilities and run-time ramifications of architecture selection. Some general observations follow, but most details are covered in documents devoted to a particular architecture, the architecture supplements.

For additional documentation that pertains to VxWorks architecture support, refer to the following:

- *VxWorks Architecture Supplement* (for the appropriate target architecture)
- *VxWorks BSP Developer's Guide*
- *VxWorks Device Driver Developer's Guide*
- Specific BSP Documentation (for a BSP closely related to your target)
- *Tornado User's Guide* (for VxWorks 5.x users)
- *Wind River Workbench User's Guide* (for VxWorks 6.0 users)
- Wind River Technical Notes, available online through the Wind River Online Support Web site

Interrupt Handling

Interrupts asynchronously connect the external world to the system, and are typically the most important aspect of real-time systems. VxWorks adopts a vectored interrupt strategy where applications "connect" ISRs (Interrupt Service Routines) to a unique vector generated by the interrupting component. VxWorks provides functions to dynamically program these vectors to contain the address of an extremely small and fast code stub that calls an application's C-language ISR, and then returns control to the kernel.

A frustrating complication to ordinary interrupt servicing is interrupt acknowledgment (IACK). Most system architectures provide for automatic interrupt acknowledgment. For the relatively few that do not address this issue, ISRs must manually acknowledge an interrupt through a register access or by some other awkward mechanism.

Finally, interrupt latency may vary from architecture to architecture. Interrupt latency is the maximum amount of time from the initial processor interrupt request to the start of interrupt service processing. Both hardware and software contribute to interrupt latency. The hardware may prioritize external interrupts, thereby introducing an intrinsic latency to lower-priority interrupts.

Architectures often have indivisible instructions whose execution times are surprisingly long. Especially problematic are cache push operations, which may

take tens of uninterruptable microseconds. The operating system and application also contribute to interrupt latency by inhibiting the processor's ability to receive interrupts. While each VxWorks architecture port has optimized these interrupt locks to an absolute minimum, be aware that some variation in performance exists when comparing one architecture to another.

Non-maskable interrupts should not be used. On some architectures, they may be usable, provided that they do not call any VxWorks kernel routines as part of the service routine. However, on other architectures, they cannot be used for anything other than system reboot.

Cache Issues

Many architectures have instruction and data caching to increase processor performance and reduce CPU bus activity. The most difficult aspect of memory caching is that the technology has often addressed the cache coherency problem inadequately.

The cache coherency problem refers to cached information that is redundant with the information in memory. If another bus master or DMA device updates memory, the cached data no longer reflects the actual value in memory. Without sufficient hardware support, solving the coherency problem is left to the software.

Unfortunately, cache management varies greatly from architecture to architecture. In some cases, the architecture provides cache management instructions; in others, cache management is bundled together with functions for managing virtual memory.

VxWorks provides a cache library interface that is unified across disparate CPU architectures. This permits highly portable, high-performance device drivers to be implemented with VxWorks. For more information, see the reference entry for **cacheLib**.

When considering hardware snooping, only full cache snooping is of benefit. Some processors implement partial snooping, but partial snooping does not meet common memory coherency requirements. Only when the snoop hardware makes the memory fully coherent is it useful for VxWorks.

The combination of copyback cache without snooping is particularly dangerous, although the risk is reduced if all user buffers are positioned so that they do not share cache lines with any other buffer.

The user can protect the rear end of any buffer by increasing the size of the memory request by one cache line. This guarantees that no other buffer shares a cache line

with this buffer. Having done all this, memory buffers for DMA operations are relatively safe from the effect of memory coherency in a copyback situation.

Many processors implement write pipes that can buffer write operations when the bus controller is busy. Because of this, VxWorks device drivers make use of a macro, **CACHE_PIPE_FLUSH**, to flush cache contents. A **CACHE_PIPE_FLUSH** operation should be inserted between any I/O write operation and a I/O read operation. If a routine begins with an I/O read then drivers should assume that an I/O write operation precedes it.

MMU Support

VxWorks supports several different MMUs through a virtual memory library. Because virtual memory is inappropriate for some real-time applications, VxWorks can be configured to not include virtual memory support. In other applications, the MMU can provide memory protection for the text section and for other uses.

For more information on VxWorks virtual memory support, see the following:

- the reference entries for **vmBaseLib** and **vmLib**
- *VxWorks Programmer's Guide* (VxWorks 5.5)
- *VxWorks Kernel Programmer's Guide* (VxWorks 6.0)
- *VxWorks BSP Developer's Guide*

Floating-Point Support

Floating point is supported as a tasking extension to avoid increased context switch times for tasks that do not use floating-point operations. By default, the floating-point registers are not saved or restored during context switches. For tasks that are expected to use the floating-point hardware, each such task can be spawned with the floating-point option, **VX_FP_TASK**, which causes the floating-point registers to be saved and restored during context switches involving the specified task. Then, context switch callouts provide the mechanism to initialize, save, and restore the floating-point context. By switching floating-point data and control registers in and out, each task effectively shares the single floating-point unit.

For tasks that do not use the floating-point hardware, the VX_FP_TASK option is not specified. In this case, floating-point registers are not saved, and the performance cost incurred by context switch times for those tasks is reduced.

Higher-level transcendental functions are supported in VxWorks in one of these ways:

- A portable version that avoids using any floating-point instructions is standard, but can be replaced with an optimized (assembly language) version for certain architectures with floating-point capabilities. For more information on selecting optional features, see the configuration discussion in your VxWorks programmer's guide.
- For floating-point intensive applications, coprocessors offer significant performance advantages.

Other Issues

Other features worth consideration include the following:

- The endian byte order selection is transparent to full VxWorks functionality. However, some BSPs support only one byte order. For information on which byte orders are supported, refer to the BSP.
- An architecture with indivisible read-modify-write operation, such as test-and-set, is necessary for high-performance backplane network communication.
- It is recommended that NMIs be used only for system reboot. On some architectures, NMIs are completely precluded except to reboot the system. On other architectures, non-maskable interrupts must be restricted to events that require no operating system support.

5. Memory

This section discusses the following issues:

- RAM
- ROM
- Ethernet RAM
- NVRAM
- parity checking
- addressing

RAM

VxWorks CISC processors require 1 MB of RAM for a development system that includes the standard VxWorks features, such as the shell, network, file system, loader, and others. RISC processors typically require more RAM space: 2 MB of RAM is the minimum; 4 MB is encouraged. For a scaled-down production system, the amount of RAM required depends on the application size and the options selected.

ROM

VxWorks CISC processors require a minimum of 128 KB of ROM, which is just sufficient for stripped-down VxWorks compressed boot ROMs on most CISC architectures. For most CISC-based systems, 512 KB of ROM is preferred. RISC processors typically require greater ROM space; 512 KB of ROM is considered a minimum, because compressed boot images are typically greater than 256 KB in size. Many full-features **vxWorks** images can be larger than 1 MB therefore, 2 MB of ROM is preferred. If you are using TrueFFS, and the same flash bank will contain both a VxWorks image and a file system, more flash is required.

Applications running from ROM are usually slow because of 16-bit or (more commonly) 8-bit data width, slow access times, and so on. VxWorks avoids this problem by typically copying the contents of the boot ROMs into RAM. However, ROM-resident VxWorks images are supported on most BSPs. For information on which image types are supported, refer to the BSP.

Non-Volatile RAM

VxWorks can use 255 or 512 bytes of non-volatile RAM (NVRAM) for storing the boot line information. Without NVRAM, the correct boot line must either be burned into the boot ROMs or typed in after every reset/power-up during development.

NVRAM can be implemented with battery-backed static RAM, EEPROM, or other technology. A number of boards use the Mostek MK48T02 or Dallas Semiconductor 1643, both of which contain a time-of-day clock in addition to battery-backed static RAM. In this case, the time-of-day clock is not used by VxWorks.

Refer to the man page for the appropriate memory driver for more information on configuring a specific driver. Almost all drivers use the configuration macros

NV_RAM_SIZE, **NV_BOOT_OFFSET**, and **NV_RAM_ADRS**. These macros are usually defined in **config.h**, *bspname***.h**, or **configAll.h**.

Parity Checking

VxWorks makes no direct use of memory parity checking on the RAM. If parity checking is desired or needed, it is usually left to the BSP or the user to enable parity and to implement a parity error handling routine. Some architectures may specify an interrupt or exception vector to be used for parity error handling.

Addressing

The address map for VxWorks itself is not important; however, a complex distribution of code and data within memory might not be supported by the toolchain.

The normal configuration methods use only three variables to determine address mapping: **RAM_HIGH_ADRS**, **RAM_LOW_ADRS**, and **ROM_TEXT_ADRS**. More complex address maps are not recommended.

6. Bus Support

This section describes issues of concern when considering the following bus types:

- PCI, cPCI, and PMC
- VMEbus
- busless

PCI, cPCI, and PMC

Wind River provides drivers compliant with revision 2.2 of the PCI Specifications. These drivers provide support for manual and automatic configuration and enumeration of devices. Also provided is an interrupt library module that handles the overloading of PCI interrupt sources. Refer to the reference entries for

pciConfigLib, **pciConfigShow**, **pciAutoCfg**, and **pciIntLib** for more information. *Wind River Technical Note #49* also contains information regarding PCI support.

Addressing

The standard Wind River PCI configuration causes all memory space on the PCI bus to be accessed through the MMU, which requires system memory to hold the page table entries (PTEs).

Dual-Port Memory

Most CPU boards have local (on-board) RAM on the processor bus. PCI is normally configured so that all of the on-board RAM is accessible to PCI devices for data buffers. This is also called dual porting. Dual porting is required for off board devices that DMA into target memory.

Using processor memory for device data buffers can cause a slight amount of increased interrupt latency. When the device is transferring data, the processor is unable to fetch instructions or data required by ISRs immediately due to bus contention. For most applications, this is not a problem. However, for applications with extreme low latency requirements, this can be ameliorated somewhat by locating device buffers in PCI memory

VMEbus

This section discusses issues of concern to the BSP developer considering the VMEbus.

VME Specification C.1

VME interoperability is crucial for the success of the standard. Special-purpose extensions to the bus should be confined to the user pins on rows A and C of the P2, and they should be clearly documented. Under no circumstance is it acceptable to deviate from the timings presented in the specification.

The VME-64 Specification is a superset of earlier specifications. At this time Wind River does not provide support for the plug and play features provided in the VME-64 specification or any of its extensions.

Addressing

The choice of address map is not critical in VxWorks. Local addresses can obscure parts of the VME bus address space. Some boards cannot address low memory on the bus because their local address starts at 0. This is not a problem for VxWorks,

because all VME device drivers are configurable. However, conflicting devices may be a system issue.

Dynamic Bus Sizing on VMEbus Accesses

There are three address types defined in the specification:

- A16 short
- A24 standard
- A32 extended

In addition, there are often data width restrictions to off-board devices.

Many implementers offer different windows with different data widths (D16 or D32) to the same VME address.

Especially useful are windows that break the wider D32 accesses into two D16 accesses automatically. This can be achieved with the dynamic bus-sizing capability of some architectures (for example, 68K).

Some boards require that a register be initialized to indicate data "direction" (read/write) in addition to the AM (Address Modifier). This is inconvenient.

Wind River does not provide direct support of 64-bit addressing or data. However, this does not preclude board specific routines from providing such support to the user.

Dual-Port Memory

Most CPU boards have local (on-board) RAM. Creating a slave access window on the VMEbus makes the local memory accessible by other CPUs and DMA devices on the bus. This is also called dual porting. It is required by systems that want backplane-shared memory to be on the local processor's RAM. Such dual-ported memory should fully support RMW cycles as described below.

Dual porting is also required for off board devices that DMA into target memory, such as the Excelan EXOS-302.

It is useful if the dual-ported memory can be seen by the local processor at the memory's external address, although this is often not provided (and is not used by VxWorks).

Dual-port memory is also very useful during porting; it facilitates the use of backplane debugging.

Using processor memory for device data buffers can cause a slight amount of increased interrupt latency. When the device is transferring data, the processor is unable to fetch instructions or data required by ISRs because of bus contention. For

most applications, this is not a problem. However, for applications with extremely low latency requirements, this can be ameliorated somewhat by locating device buffers in VME memory.

Read-Modify-Write (RMW)

Read-modify-write (RMW) must be provided in an indivisible manner.

The board must adhere to the RMW mechanism defined in the VME specification; namely, a master must keep the address strobe low between the read and the write portions of the cycle. A correctly functioning dual-ported slave board keeps the RMW indivisible across both ports by detecting an address strobe that remains low.

Unfortunately, keeping the address strobe low is only viable if you are reading and writing the same single address. A more complicated indivisible instruction, such as CAS, that involves multiple addresses cannot use this, and thus has no correct mechanism for dual-ported memory. Because VxWorks uses only TAS, this is not an issue. Some vendors have added a LOCK pin to the P2 bus for this reason. However, the pin is not part of the standard and is therefore insufficient support for this mechanism. For most boards, this is not an issue.

Caching and/or snooping can be an issue for VME RMW cycles. The shared memory master board must not introduce any cache coherency problems. It must be non-cached, or protected by full snooping capabilities, in order to handle proper VME slave accesses.

For some PowerPC implementations, it has been necessary to use bus arbitration as a global semaphore for VME RMW operations. When a board cannot generate, nor respond, to RMW cycles, using the bus as a global semaphore works. Any board that cannot use RMW, arbitrates for and holds the VME bus while a read and write cycle is completed. In addition, the bus master board, where the shared objects are stored, must implement the same bus lockup protection, even if the master board can do RMW cycles correctly. This scheme is implemented in the BSP **sysBusTas( )** and **sysBusTasClear( )** functions.

Arbitration

The board should default to bus request level 3 and provide a jumper mechanism if alternative arbitration levels are supported.

It is often convenient to be able to select the manner of bus-release that can be RWD (release when done), ROR (release on request), or RAT (release after timeout).

Multiple bus request/grant levels may be critical for systems with many masters in the backplane; with round-robin arbitration it can guarantee timely access to the bus for each bus master.

If masters on the same level are daisy chained, the masters far down the bus may become “starved.”

Arbitration/System Controller

The system controller functionality should be optional and selected by a jumper.

The system controller should not be enabled through a register that can be set by software. The ability for software to read the system controller state (on/off) is useful.

The bus system controller should assert the bus RESET signal when a “local” reset is initiated.

The system controller need not arbitrate all bus levels, but if it only arbitrates one level, it is usually level 3.

It is the responsibility of the system controller to time out and assert the BERR signal for slave processors. Better implementations allow this timeout to be selected from as fast as 16 microseconds to as slow as forever. A system controller LED is useful for alerting the user to the state of the arbitration logic.

Mailbox Interrupts

Mailbox interrupts and location monitors are similar mechanisms that may be used as inter-processor synchronization methods.

A mailbox allows the interrupter to pass some information, usually a short or long word, to the receiver, while a location monitor only causes an interrupt and has no capability to pass data.

VxWorks uses these mechanisms for synchronization when transmitting network packets between nodes in the backplane. Without them, VxWorks must rely on an inefficient polling mechanism that degrades backplane network throughput. One mailbox or location monitor is used currently, but two or more would be better.

VxWorks can use VME interrupts to drive the backplane driver; this is preferable to polling but not as good as mailbox interrupts.

No information is actually passed by VxWorks when using mailbox interrupts; only their interrupt capability is used.

VMEbus Interrupts

Although VxWorks does not require VMEbus interrupts, it is a good idea to support all VMEbus interrupts, especially if off-board Ethernet capability is required. It may be possible to jumper the enabling and disabling of these interrupts, but software control is preferable. Allowing software to read the interrupt state is valuable.

VMEbus Interrupt Acknowledge

VMEbus interrupt requests must be acknowledged by the receiver. While some implementers have chosen to force the ISR to acknowledge the interrupt, the more elegant and preferred solution is to have the hardware automatically acknowledge the request and present the CPU with the correct vector.

Software interrupt acknowledgment is not recommended because it carries a significant performance penalty.

VME interrupt generation capability is also desirable. This is especially true if the board is used in a backplane network with boards that do not have mailbox interrupts. The most important example of this is on a backplane with a Sun board running SunOS; if VME interrupt generation capability is not provided by the slave, the backplane driver on the SunOS side needs to poll.

Power Usage

The VMEbus standard specifies the maximum power draw per voltage level. Designers must adhere to these restrictions or clearly document additional power requirements.

The typical maximum power consumption per board is about 7 watts. Boards that have requirements in excess of this should emphasize it.

Extractors

VME boards should have card extractors mounted on the front panel.

VLSI Chips

A number of VLSI chips exist that offer complete VMEbus interface functionality.

Busless

When confronted with the task of porting VxWorks to a single-board computer, Wind River strongly recommends the use of a visionICE or other JTAG debugger,

or an In-Circuit Emulator. Emulator object file formats and operations are as varied as the vendors that sell them. Contact Wind River for information on appropriate combinations.

The initial goal should be to get serial line activity. Start small. Build a simple polled serial driver and build up from there. Take advantage of any LEDs that can be blinked as checkpoints in the code are passed. Use an oscilloscope to see that interrupts are occurring and being acknowledged correctly.

See the *VxWorks BSP Developer's Guide* for more information on strategies for dealing with the initial startup of a new BSP.

7. Devices

Devices should support both read and write access; it is both expensive and error prone for software to keep copies of registers or other data written to a device.

Devices that have mandatory access timing requirements should not expect software to delay between accesses. This is both error prone and non-portable. Instead, they should automatically suspend the next access.

The rest of this section discusses more specific areas of concern for devices that may be used in your real-time system.

Interrupts

Interrupts are a major consideration in any real-time system. The issue of interrupt latency can have major influence on system design. The interrupt handler mechanism in VxWorks is designed to provide minimum interference between the interrupt event and the execution of the interrupt handler routine. A minimum response time is published in the benchmarks.

All of this effort could go to waste if the hardware interrupt circuits are designed in a way that makes interrupt handling cumbersome or inefficient. This is one reason why certain hardware is not suited to real-time systems design.

Another important fact is that the device drivers are often intentionally written to disable interrupts for a brief period of time. This is done to guard data variables that are shared (between task level code and the interrupt handler) from

corruption. Ideally, the driver can disable only the interrupt from the device the driver controls. However, because of some hardware with limiting designs, the driver must sometimes disable all interrupts on the board. This is not desirable, because no interrupts can be serviced during this lock-out period.

Some devices are capable of supplying the interrupt vector during the interrupt acknowledge bus cycle. In addition, some of these devices can provide a modified vector, where certain bits are changed to indicate the exact cause of the interrupt within the device. This is a desirable feature because it allows the proper handler routine within the driver to be executed faster than if only a single shared handler routine is provided. (A single handler routine needs to poll the device to determine which of several possible events caused the interrupt).

Many hardware designs include a special class of device: the interrupt controller. These devices "field" all of the interrupt sources of the board and provide programmable selection of interrupt parameters for each source. These parameters may be: the priority level the interrupt generates, the vector the interrupt generates, and whether the interrupt source is enabled or disabled. Some interrupt controllers can be programmed to provide round-robin arbitration of interrupt sources that are set to the same priority level.

The following are guidelines for interrupt circuitry:

1. Choose devices that can provide the interrupt vector and modify the vector according to individual events. If this is not possible, then choose devices that provide a single vector.
2. Interrupt controller devices are considered good features. They can replace or augment the requirement in (1) above. Wind River has support routines for several widely available interrupt controller devices. If implemented in-house, make sure the device is well documented.
3. Each interrupt source on the board should have the capability to individually enable/disable the interrupt from reaching the processor. This is so the driver never needs to alter the CPU interrupt level. An interrupt controller device usually provides this feature. A simple control register can also be used to block interrupt signals from a device. Most devices contain an interrupt enable/disable bit in one of the programmable registers. However, this is not always true: Wind River has experience with devices that have no mechanism for disabling their interrupt pin from being asserted.
4. All sources of interrupts must be in the de-asserted state after a hardware reset or power up. This requirement is generally wise practice, but is especially important in VxWorks. This is because of the way VxWorks is layered into modules of different functionality. If a device is asserting its interrupt signal

after reset, some programming of the device or other circuits must be done at kernel initialization time, to cause the device to de-assert its signal. The sort of device-specific knowledge needed to do this is usually contained only in the device's driver. It is not appropriate to duplicate this sequence within the kernel initialization routines.

The only other mandatory requirement for interrupt design is to provide a well diagramed and documented interrupt vector scheme. Even in a flexible open-ended design, documentation should mention default vectors and priorities (if selectable) for the devices on the board.

Some people are confused about the terminology used in discussing interrupts. Interrupt level refers to an interrupt's input priority. Priority levels are the means by which devices interrupt the processor. Interrupt vectors are the ID numbers used to identify the correct interrupt service routine in response to an interrupt. Drivers attach service routines to interrupt vectors. Drivers enable and disable interrupt levels. Also note that while it is normal for a driver to enable the interrupt level when it is initializing a device, it should not disable the interrupt level when disconnecting from the device. This is because other devices may share the same interrupt level.

Wind River Technical Note #46 discusses the creation of standard interrupt controller devices for use in certain architectures. Refer to this technical note for further information.

System Clock

A system clock interrupt is mandatory. The system clock for VxWorks requires an interrupt between 30 Hz and ~2 kHz.

The default is 60 Hz and it is desirable to generate this frequency exactly. Relying on baud-rate generators often makes this precision unattainable.

Auxiliary Clock

VxWorks uses an auxiliary clock from 30 Hz to ~2 kHz to profile CPU utilization. This is required by the spy utility. It is useful to have a 24-bit (or greater) counter driven at a high frequency (~10 MHz) for profiling performance.

Timestamp Clock

Many of the generic timers in **target/src/drv/timer** include timestamp functionality. A timestamp driver provides a high-resolution time measurement facility, typically used by the WindView product. See the *VxWorks Device Driver Developer's Guide* for more information on the API.

Serial Ports

The Tornado tools currently require that the target have at least one RS-232 serial port or an Ethernet port, which is needed during the development and debug phases. The serial device should be configurable to operate at a number of "standard" baud rates; 9600 is preferred for console output, and higher rates are preferred if the serial port is used for Tornado communication. Standard communications parameters are 8 data bits, 1 stop bit, no parity, and no modem control lines. It is possible to boot VxWorks without a serial interface, and in production, it is conceivable that embedded systems will not have a serial interface.

VxWorks supports software flow control; hardware flow control signals are usually ignored. Therefore, only three lines (transmit, receive, and signal ground) are required for operation.

The API for SIO type serial drivers has been expanded to include support for hardware flow control and for changing the operational parameters for each serial device. The user can alter the hardware options flag for each device in order to change the number of data bits, data parity, stop bits, and to enable or disable hardware flow control. The default settings are always to use 8 data bits, 1 stop bit, no parity, and no hardware flow control. The ioctl code for changing hardware options is **SIO_HW_OPTS_SET** (0x1005). The code **SIO_HW_OPTS_GET** (0x1006) can be used to read the current hardware options. See **target/h/sioLib.h** for the values assigned to the options flag bits. Not all drivers have been updated to respond to these new ioctl commands yet.

Additional serial ports may be provided by the hardware and required by the application, they are not required by VxWorks. If multiple serial ports are present, you should be able to set their baud rates independently.

Ethernet Controllers

VxWorks is very "network oriented," at least during the development phase; thus, it is highly desirable to have a networking interface available. The interface can be used for booting and downloading application code as well as application-specific inter-processor communication. VxWorks provides a device-independent interface to Ethernet controllers through **netLib**, which permits the use of RPC and NFS and other Internet protocols.

There are two basic classes of Ethernet devices on the market: those that share memory with the CPU and other bus devices, and those that maintain a private packet buffer hidden from the CPU.

The devices that share memory with the CPU do so through DMA bus cycles. This implies that the device is sharing cycles on the bus while it is transmitting and receiving data. This can have a non-deterministic effect on the application.

The devices that hide the packet buffer from the CPU typically require CPU processing to move the packet data in and out of the private buffer. This is commonly done with byte or word moves to or from a register on the device. This model may provide better deterministic behavior because the CPU is in complete control of the movement of packet data. However, if the CPU is responsible for transferring data, the typical result is lower network throughput and increased processor load. The system designer needs to determine the trade-off between higher performance and more deterministic system behavior.

Within the shared memory class of devices is another classification: those devices that only deal with contiguous memory, and those devices that deal with fragmented memory. Devices that can deal with fragmented memory are generally preferred for VxWorks, because the network interface driver exchanges **mbufs** with the protocol modules, and mbufs are basically memory fragments. If a device can only deal with contiguous memory, the driver must copy the packet data between mbufs and this contiguous memory, which can affect performance.

If a device from the shared memory class is used, it is advantageous to select one that does not have any addressing or memory segment restrictions. This keeps the driver simpler, more efficient, and more generic.

If data caching is an issue with the selected processor, it is advantageous if the Ethernet controller/DMA device and the CPU have hardware cache coherency support such as the snoop lines on the MC68040 and the Intel 82596, or if the Ethernet device and the memory it uses can be "marked" as non-cacheable. If hardware support is not available, the driver must take into consideration cache coherency, performing cache-invalidate or cache-flush operations at appropriate times. This makes the driver more complex and less generic.

Designing a CPU board to include an Ethernet chip saves the expense of additional off-board networking hardware (in a "bus-full" environment) and, potentially, has a higher performance.

A detailed description of writing a VxWorks END network device driver is included in the *VxWorks Device Driver Developer's Guide*.

USB Controllers

Wind River USB host and peripheral stacks are provided for VxWorks starting with VxWorks 6.0. For releases prior to VxWorks 6.0, support for the Wind River USB host or peripheral stacks may be available as an optional product. For more information, see the Wind River Online Support Web site.

NOTE: Wind River USB, 2.1 does not currently support the on-the-go specification.

Wind River provides drivers for a limited number of USB devices. For support of other devices, you must provide your own custom drivers.

SCSI Controllers

SCSI (Small Computer Systems Interface) controllers can be used to control hard disks, floppy disks, and tape drives. These devices can be used for local booting and data storage as required by the application. VxWorks provides a device-independent interface to SCSI controllers through **scsiLib**, which also permits the use of the MS-DOS and RT-11 compatible file systems and the "low-level" raw file system.

The use of a SCSI controller with internal or external DMA capability is not required, but use of DMA greatly enhances the performance of the driver. The same cache coherency and addressing issues that apply to Ethernet controllers also apply to SCSI controllers.

It is important that the SCSI controller selected support the type of SCSI target required by the application. If advanced SCSI features are important, the SCSI controller must be able to provide the required features.

A detailed description of writing a VxWorks SCSI device driver is included in the *VxWorks Device Driver Developer's Guide*.

DMA Controllers

DMA controllers can free the processor of lengthy copies to I/O devices or off-board memory and may optionally be used by SCSI device drivers.

Reset Button

A reset button should be provided that is functionally equivalent to a power-on reset. If operating in a bus-full environment, the reset signal may need to be propagated to the bus.

Abort Button

The ability to generate a non-maskable-interrupt (NMI) allows the user to retake control from a wayward interrupt-bound processor, without resetting the whole board.

DIP Switches

Selection of addressing and interrupts is more convenient with DIP or rotary switches than with jumpers. Software-readable DIP switches (jumpers) are an advantage during configuration.

User LEDs

LEDs are useful debugging tools, especially on single-board computers. They allow for quick visual inspection of a board to verify its functionality. With a row of LEDs, information such as error codes can be displayed.

Physically, the LEDs should be in an easily visible place such as a front panel.

Programmatically, the LEDs should be in an easily visible place so that not much board initialization needs to be done before the LEDs can be used. For example, putting the LED registers on a subordinate bus is a bad idea since it prevents the LEDs from being used until the bus is configured.

Parallel Ports

VxWorks provides simple support of parallel ports. Refer to directory **target/src/drv/parallel** for supported devices and capabilities.

8. Flash Devices and Flash File System Support

This section discusses TrueFFS features and functionality. In addition, this section provides generic information on flash devices and a discussion of TrueFFS usage.

8.1 Flash Devices

There are two types of flash technology—NOR and NAND—that are named after the types of logic devices that comprise each bit in the flash device. A third type of flash technology, SSFDC, is a special case of NAND. TrueFFS for VxWorks supports NOR devices and SSFDC devices. NAND devices that do not comply with the SSFDC specification are not supported.

NOR Technology

NOR technology offers XIP (execute in place) capabilities and high read performance. Write and erase performance is very low.

NAND Technology

NAND technology allows very high cell densities and high capacity. Write, read, and erase performance is very high.

NOTE: In general, Wind River no longer supports NAND technology. Only NAND flash that is SSFDC compliant is supported.

SSFDC Technology

SSFDC (solid state floppy device card) flash is generally known as smart media. For more information about this type of flash, see the SSFDC forum at:

http://www.ssfdc.or.jp/english

8.2 Choosing TrueFFS as a Medium

TrueFFS applications can read and write from flash memory just as they would from an MS-DOS file system resident on a magnetic-medium mechanical disk drive. However, the underlying storage media are radically different. While these differences are generally transparent to the high-level developer, it is critical that you be aware of them when designing an embedded system.

Flash memory stores data indefinitely, even while power is removed. The physical components of flash memory are solid-state devices—that is, devices with no moving parts—that consume little energy and leave a small foot print. Thus, flash memory is ideal for mobile devices, for hand-held devices, and for embedded systems that require reliable, non-volatile environments that are too harsh for mechanical disks.

Flash has a limited life due to the finite number of erase cycles. TrueFFS only supports flash devices that are symmetrically blocked. In addition, some features common to block device drivers for magnetic media are not available with flash memory. Unequal read and write time is a typical characteristic of flash memory, in which reads are always faster than writes. For more information, see *Wear-Leveling, p.27*.

The unique qualities that distinguish flash memory from magnetic-media disk drives make it an ideal choice for some types of applications, yet impractical for others.

NOTE: Although you can write in any size chunks of memory, ranging from bytes to words and double words, you can only erase in blocks. The best approach to extending the life of the flash is to ensure that all blocks wear evenly. For more information, see *Wear-Leveling, p.27*.

NOTE: The TrueFFS optional product does have not support for partition tables.

8.3 TrueFFS Features

This section discusses flash memory functionality, and the ways in which it protects data integrity, extends the lifetime of the medium, and supports fault recovery.

Block Allocation and Data Clusters

As is required of a block device driver, TrueFFS maps flash memory into what appears to be (from the users perspective) a contiguous array of storage blocks, upon which a file system can read and write data. These blocks are numbered from zero to one less than the total number of blocks. Currently, the only supported file system is dosFs (for more information, see the appropriate VxWorks programmer's guide for your release).

Block Allocation Algorithm

To promote more efficient data retrieval, TrueFFS uses a flexible allocation strategy, which clusters related data into a contiguous area in a single erase unit. These clusters might be, for example, the blocks that comprise the sectors of a file. TrueFFS follows a prioritized procedure for attempting to cluster related data in the following order:

a. First, it tries to maintain a pool of physically consecutive free blocks that are resident in the same erase unit.

b. If that fails, it then tries to assure that all the blocks in the pool reside in the same erase unit.

c. If that fails, it finally tries to allocate a pool of blocks in the erase unit that has the most space available.

Benefits of Clustering

Clustering related data in this manner has several benefits, listed and described below.

- **Allows Faster Retrieval Times**

For situations that require TrueFFS to access flash through a small memory window, clustering related data minimizes the number of calls needed to map physical blocks into the window. This allow faster retrieval times for files accessed sequentially.

- **Minimizes Fragmentation.**

Clustering related data cuts down on fragmentation because deleting a file tends to free up complete blocks that can be easily reclaimed.

- **Speeds Garbage Collection**

Minimizing fragmentation leads to faster garbage collection.

- **Localizes Static File Blocks.**

Localizing blocks that belong to static files significantly facilitates transferring these blocks when the wear-leveling algorithm decides to move static areas.

Read and Write Operations

One of the characteristics of flash memory that differs considerably from the more common magnetic-medium mechanical disks is the way in which it writes new data to the medium. When using traditional magnetic storage media, writing new data to a previously written storage area simply overwrites the existing data, essentially obliterating it; whereas flash does not. This section describes how flash reads from, and writes to, memory.

Reading from Blocks

Reading the data from a block is straightforward. The file system requests the contents of a particular block. In response, TrueFFS translates the block number into flash memory coordinates, retrieves the data at those coordinates, and returns the data to the file system.

Writing to Previously Unwritten Blocks

Writing data to a block is straightforward, a byte can only be written once before it is erased. Therefore, if the target block has remained unwritten since the last erase of the block, the write process is simple. TrueFFS translates the block number into flash memory coordinates and writes to the location. However, if the write request seeks to modify the contents of a previously written block, the situation is more complex.

If any write operation fails, TrueFFS attempts a second write. For more information, see *Recovering During a Write Operation*, p.29.

NOTE: Storing data in flash memory requires the use of a manufacturer-supplied programming algorithm, which is defined in the MTD. Consequently, writing to flash is often referred to as programming flash.

Writing to Previously Written Blocks

If the write request is to an area of flash that already contains data, TrueFFS finds a different, writable area of flash instead—one that is already erased and ready to receive data. TrueFFS then writes the new data to that free area. After the data is safely written, TrueFFS updates its block-to-flash mapping structures, so that the

block now maps to the area of flash that contains the modified data. This mapping information is protected, even during a fault. For more information on fault recovery and mapping information, see *Recovering Mapping Information*, p.29.

Erase Cycles and Garbage Collection

The write operation is intrinsically linked to the erase operation, since data cannot be "over-written." Data must be erased, and then those erased units must be reclaimed before they are made available to a write operation. This section describes that process, as well as the consequences of over-erasing sections of flash.

Erasing Units

Once data is written to an area of flash memory, modifying blocks leaves behind block-sized regions of flash memory that no longer contain valid data. These regions are also un-writable until erased. However, the erase cycle does not operate on individual bytes or even blocks. Erasure is limited to much larger regions called *erase units*. The size of these erase units depends on the specific flash technology, but a typical size is 64 KB. For more information on erase units, see the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS*.

Reclaiming Erased Blocks

To reclaim (erase) blocks that no longer contain valid data, TrueFFS uses a mechanism called *garbage collection*. This mechanism copies all the valid data blocks, from a source erase unit, into another erase unit known as a *transfer unit*. TrueFFS then updates the block-to-flash map and afterward erases the old (erase) unit. The virtual block presented to the outside world still appears to contain the same data even though that data now resides in a different part of flash.

For details on the algorithm used to trigger garbage collection, see *Garbage Collection*, p.28. For information about garbage collection and fault recovery, see *Recovering During Garbage Collection*, p.30.

Over-Programming

As a region of flash is constantly erased and rewritten, it enters an *over-programmed* state, in which it responds only very slowly to write requests. With rest, this condition eventually fixes itself, but the life of the flash memory is shortened. Eventually the flash begins to suffer from sporadic erase failures, which become more and more frequent until the medium is no longer erasable and thus no longer writable (data that is already resident on the flash is still readable). Consequently, flash limits how often you can erase and rewrite the same area. This number,

known as the *cycling limit*, depends on the specific flash technology, but it ranges from a hundred thousand to a million times per block.[1]

Optimization Methods

As mentioned, flash memory is not an infinitely reusable storage medium. The number of erase cycles per erase unit of flash memory is large but limited. Eventually, flash degrades to a read-only state. To delay this as long as possible, TrueFFS uses a garbage collection algorithm that works in conjunction with a technique called wear-leveling.

Wear-Leveling

One way to alleviate over-programming is to balance usage over the entire medium, so that the flash is evenly worn and no one part is over-used. This technique is known as *wear-leveling* and it can extend the lifetime of the flash significantly. To implement wear-leveling, TrueFFS uses a block-to-flash translation system that is based on a dynamically maintained map. This map is adjusted as blocks are modified, moved, or garbage collected.

Static File Locking

By remapping modified blocks to new flash memory coordinates, a certain degree of wear-leveling occurs. However, some of the data stored in flash may be essentially static, which means that if wear-leveling only occurs during modifications, the areas of flash that store static data are not cycled at all. This situation, known as *static file locking*, exacerbates the wear on other areas of flash, which must be recycled more frequently as a consequence. If not countered, this situation can significantly lower the flash medium's life-expectancy.

TrueFFS overcomes static file locking by forcing transfers of static areas. Because the block-to-flash map is dynamic, TrueFFS can manage these wear-leveling transfers in a way that is invisible to the file system.

Algorithm

Implementing absolute wear-leveling can have a negative impact on performance because of the many required data moves. To avoid performance degradation, TrueFFS implements a wear-leveling algorithm that is not quite absolute. This algorithm provides an approximately equal number of erase cycles per unit. Over time, you can expect the same number of erase cycles per erase unit without a

1. This number is statistical in nature and should not be taken as an exact figure.

degradation of performance. Given the large number of allowed erase cycles, this less-than-absolute approach to wear-leveling is sufficient.

Dead Locks

Finally, the TrueFFS wear-leveling algorithm is further enhanced to break a failure mode known as *dead locks*. Some simple wear-leveling algorithms have been shown to "flip-flop" transfers between only two or more units for very long periods, neglecting all other units. The wear-leveling mechanism used by TrueFFS ensures that such loops are kept to a minimum. For optimal wear-leveling performance, TrueFFS requires at least 12 erase units.

Garbage Collection

Garbage collection, described in *Reclaiming Erased Blocks*, p.26, is used by the erase cycle to reclaim erased blocks. However, if garbage collection is done too frequently, it defeats the wear-leveling algorithm and degrades the overall performance of the flash disk. Thus, garbage collection uses an algorithm that relies on the block allocation algorithm (*Block Allocation Algorithm*, p.24), and is triggered only as needed.

The block allocation algorithm maintains a pool of free consecutive blocks that are resident in the same erase unit. When this pool becomes too small, the block allocation algorithm launches the garbage collection algorithm, which then finds and reclaims an erase unit that best matches the following criteria:

- the largest number of garbage blocks
- the least number of erase cycles
- the most static areas

In addition to these measurable criteria, the garbage collection algorithm also factors in a random selection process. This helps guarantee that the reclamation process covers the entire medium evenly and is not biased due to the way applications use data.

Fault Recovery in TrueFFS

A fault can occur whenever data is written to flash. For example, a fault can occur:

- in response to a write request from the file system
- during garbage collection
- during erase operations
- during formatting

TrueFFS can recover from the fault in all cases except when new data is being written to flash for the first time. In this case, the new data is lost. However, once data is safely written to flash, it is essentially immune to power failures. All data already resident in flash is recoverable, and the file and directory structures of the disk are retained. In fact, the only negative consequence of a power interruption or fault is the need to restart any incomplete garbage collection operations. The following sections describe fault occurrence and fault recovery in TrueFFS.

Recovering During a Write Operation

A write or erase operation can fail because of a hardware problem or a power failure. As mentioned in above, TrueFFS uses an "erase after write" algorithm, in which the previous data is not erased until after the update operation has successfully completed. To prevent the possible loss of data, TrueFFS monitors and verifies the success of each write operation, using a register in which the actual written data is read back and compared to the user data. Therefore, a data sector cannot be in a partially written state. If the operation completes, the new sector is valid; if the update fails, the old data is not lost or in any way corrupted.

TrueFFS verifies each write operation, and automatically attempts a second write to a different area of flash after any failure. This ensures the integrity of the data by making failure recovery automatic. This write-error recovery mechanism is especially valuable as the flash medium approaches its cycling limit (end-of-life). At that time, flash write/erase failures become more frequent, but the only user-observed effect is a gradual decline in performance (because of the need for write retries).

Recovering Mapping Information

TrueFFS stores critical mapping information in flash-resident memory, thus it is not lost during an interruption such as a power loss or the removal of the flash medium. TrueFFS does, however, use a RAM-resident mapping table to track the contents of flash memory. When power is restored or the medium reconnected, the RAM-resident version of the flash mapping table is reconstructed (or verified) from flash-resident information.

NOTE: Mapping information can reside anywhere on the medium.However, each erase unit in flash memory maintains header information at a predictable location. By carefully cross-checking the header information in each erase unit, TrueFFS is able to rebuild or verify a RAM copy of the flash mapping table.

Recovering During Garbage Collection

After a fault, any garbage collection in process at the time of the failure must be restarted. Garbage area is space that is occupied by sections that have been deleted by the host. TrueFFS reclaims garbage space by first moving data from one transfer unit to another, and then erasing the original unit.

If consistent flash failures prevent the necessary write operations to move data, or if it is not possible to erase the old unit, the garbage collection operation fails. To minimize a failure of the write part of the transfer, TrueFFS formats the flash medium to contain more than one transfer unit. Thus, if the write to one transfer unit fails, TrueFFS retries the write using a different transfer unit. If all transfer units fail, the medium no longer accepts new data and becomes a read-only device. This does not, however, have a direct effect on the user data, all of which is already safely stored.

Recovering During Formatting

In some cases, sections of the flash medium are found to be unusable when flash is first formatted. Typically, this occurs because those sections are not erasable. As long as the number of bad units does not exceed the number of transfer units, the medium is considered usable as a whole and can be formatted. The only noticeable adverse effect is the reduced capacity of the formatted flash medium.

9. Virtual Memory Library

The **vmBaseLib** (the base virtual memory support library) and the **vmLib** (the VxVMI option) have similar BSP requirements. Because **vmBaseLib** is bundled with VxWorks, BSPs are written to include it by default. If the VxVMI option is installed, the end user need only change **INCLUDE_MMU_BASIC** to **INCLUDE_MMU_FULL** in **target/config/***bspname***/config.h** to include the unbundled library in future builds.

For more information, refer to the **vmBaseLib** and **vmLib** reference entries and your VxWorks programmer's guide.

NOTE: The optional VxVMI product is not available forVxWorks 6.x. For information on optional products supported with your release, see your product release notes.

Note that some processors have special requirements related to virtual memory.

⚠ **CAUTION:** Due to a limitation in the current implementation of VxWorks, virtual addresses must equal physical addresses when using 68K MMUs (68K support is available for VxWorks 5.5 only).

10. Partial Compliance to Standards

Many devices of all types have claims that they support certain standards. However, when the device's errata are examined closely, there are significant exceptions. In general, VxWorks libraries are built to support the full standard sets. When significant exceptions are present, the devices may not work with the VxWorks libraries. For this reason, it is important to make sure that the devices used in the hardware design are fully compliant with the standards, or there may be serious redesign issues when the incompatibility issues are uncovered.

The general rule is this: if the chip has a feature or feature set with reduced capability, do not count on having access to that feature or feature set from VxWorks at all. VxWorks libraries are tested with devices which are known to have good adherence to the standards, not with devices which have poor adherence to the standards.

This affects all of the parts on a board design, from the processor right on down.

One of the processors which exhibits this problem is one specific variation of the MIPS processor. This processor has only single-precision, floating-point support as a cost savings measure, and can only be configured to use 32 single-precision (that is, 32-bit) floating-point registers. Wind River builds the floating-point library routines for MIPS64 with double-precision floating point, so that they support 32 double-precision registers. And the MIPS32 floating-point support is designed to accommodate sixteen double-precision registers. But neither configuration will work for that processor, because it can only be configured to use 32 single-precision floating-point registers. Because of this incompatibility, applications for boards using this chip cannot use hardware floating point without costly floating-point library revision. In this case, software floating point is the recommended resolution.

Another example might be a PCI host bridge. An existing PCI host bridge intended for high-end server applications was designed so that it allowed only 64-bit

transactions on the PCI bus. The result was that every device driver for devices which might be placed on the PCI bus would not work, because they use the full PCI capabilities and transactions of less than 64-bit size. So, no PCI devices would work for any system based on that host bridge with standard drivers, and all device drivers on such a system must be custom versions, resulting in greater software costs.

Another example of this problem on a device was an early version of a specific SCSI-2 device. In an earlier version of VxWorks, the Wind River SCSI driver legally asserted the attention line during the command phase. The microcode used in the device was incompatible with this, and caused the entire bus to become unusable. Although Wind River has modified the SCSI driver in a way which overcomes this particular difficulty, this example does illustrate the consequences of hardware which does not fully conform to industry standards.

There are many devices which exhibit this problem. In each case, the solution requires either additional expense in software development, more than making up for the savings of the reduced part, or reduced system capability and/or performance.

The resolution is to design using hardware which fully complies with the required standards.