WIND RIVER WHITE PAPER

Implementing Basic Memory Protection in VxWorks: A Best Practices Guide

Paul Chen
Product Manager, Wind River

Contents

Implementing Basic Memory Protection in VxWorks: A Best Practices Guide

1. Introduction

Wind River's **VxWorks**® is the most widely deployed real-time operating system (RTOS) in the embedded device market. With over 46 percent share of the embedded RTOS market[1], **VxWorks** has enabled thousands of customers worldwide to deliver millions of reliable, high-performance products to market — rapidly, efficiently and cost-effectively. To achieve the guaranteed responsiveness and determinism that are crucial characteristics of mission- or life-critical applications, **VxWorks** supports application development exclusively in kernel (or "supervisor") mode. Executing in a flat, non-protected memory space enables applications to obtain the utmost performance, determinism, and application flexibility. Kernel mode execution, however, precludes the ability to take advantage of some types of memory protection enforced by a Memory Management Unit (MMU). Use of an MMU can protect application code, kernel code, and critical data. This memory protection is usually accompanied by a cost in reduced system performance, due to the implementation of a process model-based or a message-passing architecture.

As the sophistication of embedded applications has increased, however, issues of system reliability and protection have become ever more important. If an embedded system can withstand some performance degradation, its applications can benefit from memory protection mechanisms to improve reliability. Errant or poorly designed applications can be prevented from having harmful or disastrous effects on the embedded system. Memory protection between applications supports a "fault containment" concept, and can limit the extent of the effect of certain types of software faults. Memory protection also assists development, catching memory trashing bugs during integration or debug cycles that are often difficult to detect and diagnose.

While **VxWorks** does not provide full memory protection and application isolation, Wind River does offer both basic MMU support that is bundled with **VxWorks**, as well as **VxVMI**™, an unbundled Operating System (OS) Extension for **VxWorks** that provides additional memory protection features and programmatic access to the MMU. The basic memory protection features in **VxWorks** utilize the MMU that is present in many of today's processors to enhance productivity during the development and debugging cycles. Use of memory protection can also improve the reliability and quality of an embedded system by reducing security risks in its applications, and can improve the device's time to market by making development and debugging cycles both more efficient and effective. To enable developers to take advantage of these benefits, this paper describes some "best practice" examples for implementing memory protection in **VxWorks** when designing reliable embedded applications.

2. Embedded Device Development Issues

Over the past few years, Wind River has been tracking key trends in the embedded device market. The trends relevant to this paper involve issues facing the developers of embedded devices. These issues have a direct impact on system design requirements, as described below.

Increasing software complexity

Embedded applications comprising several megabytes of code are becoming commonplace. Larger development teams, from different divisions or business units or even from different companies, collaborate on designs and face an increasing challenge to create robust applications. System reliability must be ensured inherently by a system's software design, including OS-provided memory protection, rather than by relying on extensive system-level integration testing.

Compressed development times

To remain competitive, developers of embedded devices face increasing pressure to shorten project development cycles and to bring devices to market more quickly. Use of memory protection can greatly enhance the efficiency of both development and debugging phases of projects, thereby improving the device's time to market.

Expanding device functionality

Embedded devices are becoming open, multifunction, managed appliances. Embedded applications will become increasingly more dynamic as vendors allow customization (personalization), extensions, and software updates on such devices. Again, memory protection becomes critical to device reliability.

Demanding service-based economy

As the economy becomes progressively more service-based, it grows more dependent on the timeliness and availability of those services. Downtime for the provider means a loss of service revenue and customer loyalty. Downtime for the consumer means a denial of service access and user frustration. These consumers, unlike desktop users, are extremely intolerant of failures. Systems providing these services require protection from common memory faults to improve their overall robustness.

Increasingly connected devices

When embedded devices are opened up to the outside world, they must preserve system reliability and robustness through protective measures against inadequately debugged or poorly designed third-party applications.

Due to these and other concerns, memory protection has assumed paramount importance for developers of embedded devices. Using **VxVMI**, developers can meet this requirement while obtaining the small footprint, high performance, reliability, and determinism that **VxWorks** is known throughout the industry for providing.

3. VxWorks and Memory Protection

VxWorks is a task-based RTOS in which applications run in kernel (or "supervisor") mode. In this mode of operation, all of physical memory can be accessible to developers in real-addressing mode. Applications that execute in kernel mode can achieve the highest performance and

[1] "2002 Embedded Market Study," CMP Media LLC, November 2002.

determinism, and have the utmost in flexibility. One risk, however, of executing in kernel mode is that any area of memory — e.g., the application code, data, or even the kernel itself — can potentially be overwritten by software bugs or misbehaving applications.

Many processors today provide an on-chip or onboard Memory Management Unit (MMU) that can be used to protect memory against unintentional or undesired access. To enable developers to take advantage of memory protection in their **VxWorks**-powered embedded devices, Wind River provides bundled, basic MMU support and an unbundled OS Extension called **VxVMI**. Some examples of how memory protection can assist the development of **VxWorks** applications include:

- Detection of buffer overruns or underruns
  Using either the bundled MMU support or the APIs provided by **VxVMI**, pages allocated around local buffers can be write-protected so that attempts to write beyond buffers can be detected both during debugging and in commercially deployed products and systems.
- Protection of private data
  Using either the bundled MMU support or the APIs provided by **VxVMI**, private data areas can be created for tasks that are protected from access by other tasks.
- Protection of kernel and application code
  VxVMI can automatically write-protect text segment pages in memory, so that attempts to write to kernel or application code can be detected during debugging and in deployed products.
- Protection of the interrupt vector table
  VxVMI can automatically write-protect the interrupt vector table, so that attempts to overwrite this table (usually through corrupted memory references) can be detected during debugging and in deployed products.
- Detection of null pointer write exceptions
  Since the interrupt vector table is typically located at memory address `0x0`, the protection of this table by **VxVMI** also enables detection of writes to a null address.

By utilizing **VxWorks** memory protection, developers of software for embedded systems can realize the following benefits:

- Improved efficiency of development and debugging
  Memory protection enables faster diagnosis of incorrect memory accesses.
- Improved system reliability and robustness
  Memory protection helps prevent bugs from crashing devices.
- Reduced development risk
  Memory protection helps reduce the possibility that latent bugs in the shipping code, or new bugs from add-on applications, will adversely affect the functionality or performance of devices.
- Improved time to market
  More efficient development using memory protection leads to faster development cycles.
- Improved device quality
  Memory protection enables safer code, and more graceful degradation of device behavior in the face of errors.

While there are certainly cases in which applications need to be developed in kernel mode because of their requirements for determinism or high performance, there are many cases where use of basic memory protection and **VxVMI** can greatly increase the reliability and quality of embedded devices that use **VxWorks**.

4. VxWorks Basic Memory Protection

The basic level of MMU support bundled with **VxWorks** primarily provides cache-safe buffers (see Chapter 12, "Virtual Memory Interface," in the *VxWorks Programmer's Guide*). Importantly, too, it provides programmatic modification of page status, which enables developers to implement a basic level of memory protection.

To enable the bundled, basic MMU support in VxWorks, define the configuration macro `INCLUDE_MMU_BASIC`. See figure 1.

The ability to modify page status dynamically enables developers to:

- Protect private memory
- Protect shared memory
- Detect buffer overruns and underruns

Each of these benefits will be discussed below. For a summary of basic MMU routines, see `vmBaseLib` in the *VxWorks OS Libraries API Reference*.

Protecting private data

There are many instances when tasks need to protect data from being modified by other tasks. Without memory protection, such data would be stored in a global area where any task could read or overwrite the data, causing potentially undesirable side effects that may be difficult to diagnose and debug.

To enable basic MMU support:

- In **TORNADO**,® the project must be a "bootable application"
  1. Go to the VxWorks view in the project's Workspace window
  2. Navigate the component tree:
     [Project name]→hardware→memory→MMU→MMU Mode→basic MMU support
  3. Right-click on basic MMU support and select Include basic MMU support

- To define the macro manually in the configuration header files, define `INCLUDE_MMU_BASIC` in config.h

Figure 1

To implement protected private memory:

- Add a routine using `taskSwitchHookAdd()` that is called with every task switch, to dynamically change the state of "private memory" pages
  - Call the macro `VM_STATE_SET()` (which calls `vmBaseStateSet()` for bundled MMU support)
  - Set the memory pages to `VM_STATE_VALID` upon entry into the task, to enable this task to manipulate its "private memory"
  - Set the memory pages to `VM_STATE_VALID_NOT` upon leaving the task, to protect that memory from access by other tasks

Tasks can thus create their own protected private memory. If any other task attempts to access an address within the space of that context, a translation look-aside buffer (TLB) exception will occur, leading to detection during debugging.

Protecting shared data

Often, tasks need to share data in a protected manner. For example, in a producer-consumer model, one task will produce data and store them in a particular data structure while consumer tasks will read the data to accomplish certain functionality. Protecting the data from being overwritten by unauthorized consumers can increase the reliability of the system.

To implement protected shared data:

- Allocate memory for the shared data
  - E.g., use `valloc()`
- Create a semaphore to control access to the shared data
  - E.g., use `semBCreate()`
- Write-protect the shared data area
  - Use the macro `VM_STATE_SET()` (which calls `vmBaseStateSet()` for bundled MMU support) and `VM_STATE_WRITABLE_NOT`
- Create an access routine that can write to the shared memory
  - Take the semaphore while temporarily write-enabling the shared memory using `VM_STATE_SET()` and `VM_STATE_WRITABLE`

Any task requiring only read-access to the shared memory area can access it directly; however, any task requiring write-access to the shared memory area must do so using the access routine. Any write-access attempted outside the access routine will cause a TLB exception, leading to detection during debugging.

Detecting buffer overruns/underruns

Memory protection can be used to help detect programming errors related to allocated memory buffers or to application-defined arrays. Common programming errors include attempting to access before the beginning of, or beyond the end of, a buffer or data structure and using a data structure that grows larger unexpectedly.

By allocating "guard" pages on either side of a buffer or data structure, and disabling access to those pages, incorrect overrun or underrun accesses to buffers or data structures can be detected.

To implement guard pages:

- Allocate memory for the buffer or data structure
  - Increase the size requested by two extra pages' worth of bytes
  - The buffer needs to be page aligned
  - The actual data structure begins one page length into the allocated memory and ends one page length before the end of the allocated memory
- Read- and write-protect the "guard" pages
  - Disable access to the first and last pages of the allocated memory using the macro `VM_STATE_SET()` (which calls `vmBaseStateSet()` for bundled MMU support) and `VM_STATE_VALID_NOT`

5. VxWorks Memory Protection using VxVMI

The OS Extension **VxVMI** provides basic memory protection features, as well as a programmatic interface to the MMU. The memory protection features are described below, and section 6 describes some best practices that make use of the MMU APIs.

To enable **VxVMI** and memory protection, make sure that **VxVMI** has been installed. Then define the configuration macro `INCLUDE_MMU_FULL`. All **VxVMI** routines are then available for use in applications. See figure 2.

For a summary of **VxVMI** routines, see `vmLib` in the *VxWorks OS Libraries API Reference*. Also see Chapter 12, "Virtual Memory Interface," in the *VxWorks Programmer's Guide*.

To enable full MMU support:

- In **TORNADO**, the project must be a "bootable application"
  4. Go to the VxWorks view in the project's Workspace window
  5. Navigate the component tree:
     [Project name]→hardware→memory→MMU→MMU Mode→full MMU support
  6. Right-click on full MMU support and select Include full MMU support

NOTE: The **TORNADO**® *IDE* user interface will provide a warning if both basic MMU support and full MMU support have been selected, as these are mutually exclusive options.

- To define the macro manually in the configuration header files, define `INCLUDE_MMU_FULL` in config.h

Figure 2

Protecting program code

VxVMI can protect program code from unintentional overwriting — a common hazard when applications run in kernel mode and bugs cause pointers to reference incorrect memory locations. To enable program code (text segment) protection, simply define the configuration macro `INCLUDE_PROTECT_TEXT`. No additional code is necessary to benefit from **VxVMI** text segment protection (see figure 3). When the rebuilt image is downloaded to the target, all text segment pages will automatically be write-protected in memory. Additionally, the text segments of any object modules that are subsequently loaded using `ld()` will also automatically be marked as read-only. When object modules are loaded, memory to be write-protected is allocated in page-size increments. Any attempt to write to a memory location that is write-protected will cause a TLB exception.

Protecting the interrupt vector table

VxVMI can protect the interrupt vector table from unintentional overwriting — another common hazard when applications run in kernel mode and null pointer errors occur while writing to memory. To enable interrupt vector table protection, simply define the configuration macro `INCLUDE_PROTECT_VEC_TABLE`. No additional code is necessary to benefit from **VxVMI** interrupt vector table protection (see figure 4).

Interrupt vector table protection sets the table to be nonwritable. Any attribute changes to the table required by the kernel are made automatically; however, if the application needs to modify the interrupt vector table, it can use the routine `intConnect()`, which write-enables the interrupt vector table for the duration of the call.

6. VxWorks Memory Protection using VxVMI: Advanced Best Practices

Protecting text segments and the interrupt vector table are the basic, automatic benefits of using **VxVMI**. In addition, **VxVMI** provides an API that allows developers to manipulate the MMU programmatically using low-level routines. Using these routines, developers can make data private to a task or to a code segment. And as with the basic level of MMU support bundled with **VxWorks**, developers can also use **VxVMI** routines to make portions of memory non-cacheable or can write-protect portions of memory.

Protecting private data

As described earlier, private data can be protected using the MMU support bundled with **VxWorks**. An alternative implementation can be achieved using **VxVMI**, as described below.

To implement protected private memory using **VxVMI**:

- For each task that requires protected private memory, create a private virtual memory (VM) context using `vmContextCreate()`
  - Disable access to the corresponding addresses in global virtual memory using the macro `VM_STATE_SET()` (which calls `vmStateSet()` if **VxVMI** is installed and full MMU support is enabled) and `VM_STATE_VALID_NOT` to ensure that these pages are inaccessible from the default global virtual memory map
- Add a routine using `taskSwitchHookAdd()` that is called with every task switch
  - This routine should swap virtual memory contexts, saving any prior private context and installing any new private context

Tasks with their own protected private memory will allocate and use memory from this context. If any other task attempts to access an address within the space of that context, a TLB exception will occur, leading to detection during debugging.

The advantage of protecting private data using **VxVMI** is that it will often be faster to switch a VM context rather than to dynamically change the state of a number of private data pages upon task switching (the bundled support technique).

For more details on programming protected private memory, see "Example 12-1: Private Virtual Memory Contexts" in Chapter 12, "Virtual Memory Interface," of the *VxWorks Programmer's Guide*.

To protect text segments:

- In **TORNADO**, the project must be a "bootable application"
  1. Go to the VxWorks view in the project's Workspace window
  2. Navigate the component tree:
     [Project name]→hardware→memory→MMU→write-protect program text
  3. Right-click on write-protect program text and select Include write-protect program text
- To define the macro manually in the configuration header files, define `INCLUDE_PROTECT_TEXT` in config.h

Figure 3

To protect the interrupt vector table:

- In **TORNADO**, the project must be a "bootable application"
  1. Go to the VxWorks view in the project's Workspace window
  2. Navigate the component tree:
     [Project name]→hardware→memory→MMU→write-protect program text
  3. Right-click on write-protect vector table and select Include write-protect vector table
- To define the macro manually in the configuration header files, define `INCLUDE_PROTECT_VEC_TABLE` in config.h

Figure 4

Protecting shared data

As described earlier, shared data can be protected using bundled MMU support. **VxVMI** also enables developers to protect shared memory, and the implementation thereof is essentially the same as when using the basic MMU support.

For more details on programming protected shared data using **VxVMI**, see "Example 12-2: Nonwritable Memory" in Chapter 12, "Virtual Memory Interface" of the *VxWorks Programmer's Guide*.

Detecting buffer overruns/ underruns

Also as described earlier, buffer overruns and underruns can be detected using the bundled MMU support APIs. **VxVMI** can also be used to detect these errors, again using the same technique as for the bun-dled case detailed earlier. By using the macro `VM_STATE_SET()`, the system will automatically call the proper state-setting routine, `vmBaseStateSet()` or `vmStateSet()`, based on whether the bundled MMU support or full MMU support with **VxVMI** is available.

Examining virtual memory contexts

A virtual memory context can be examined in the debugger that is included with the **TORNADO**® or **WIND**®**POWER** *IDE* toolsets. The display appears on the standard output device and provides data about each memory block: its validity, its ability to be written, and its ability to be cached. This information can be invaluable when debugging the embedded system software.

To enable **VxVMI** virtual memory context display, simply define the configuration macros `INCLUDE_MMU_FULL` and `INCLUDE_SHOW_ROUTINES`. See figure 5.

Once virtual context display is enabled, call `vmContextShow()` from the shell as necessary to examine a virtual memory context.

To enable virtual context display:

- In **TORNADO**, the project must be a "bootable application"
  1. Go to the VxWorks view in the project's Workspace window
  2. Navigate the component tree:
     [Project name]→development tool components→show routines→MMU show routine
  3. Right-click on write-protect program text and select Include write-protect program text

- To define the macro manually in the configuration header files, define `INCLUDE_MMU_FULL` and `INCLUDE_SHOW_ROUTINES` in config.h

Figure 5

VxVMI® Availability

Architecture	Family	Processor(s)
Motorola® 68K	68000, 68030, 68040, 68060	MC68030, MC68040, MC68LC040, MC68060, MC68302, MC68EN302
Motorola PowerPC™	Motorola MPC8xx	MPC823, MPC823e, MPC850, MPC850DSL, MPC850SAR, MPC855T, MPC857T, MPC860, MPC860DP, MPC860P, MPC860SAR, MPC860T, MPC862DT/DP/P/SR/T
ARM®	ARM7™	ARM7TDMI®, ARM720, ARM720T™, ARM740, ARM740T™
	ARM9	ARM920T™, ARM940T™, ARM966E-S™
	ARM10	ARM10
Intel® StrongARM® and XScale™	StrongARM and XScale	SA-110®, SA-1110®, IXP1200, 80200
Hitachi SH	SH3 (Big and Little Endian) / DSP	SH7729, SH7709A, SH7727
	SH4 (Big and Little Endian) / FPU	SH7750, SH7751
Intel Architecture	i486® and Pentium®	i486, Pentium, Pentium with MMX™
	Pentium II	Pentium Pro, Celeron®, Pentium II, Pentium II Xeon™
	Pentium III	Pentium III, Pentium III Xeon
	Pentium 4	Pentium 4

Figure 6

7. Getting VxVMI

VxVMI is an OS Extension for **VxWorks** or the **VxWorks**® *DEVELOPER TOOL KIT* (VDT). The product description and data sheet for **VxVMI** are available on the Wind River Website: http://www.windriver.com/products/vxvmi/index.html.

VxVMI availability

VxVMI is currently available on the architecture families and processors in figure 6.

In addition, WIND RIVER® Service Teams have ported **VxVMI** to several other Motorola PowerPC processors, including MPC603, MPC604, MPC750, and MPC8260. WIND RIVER Service Teams would be able to provide these ports or to discuss porting **VxVMI** to a currently unsupported processor.

8. Summary

VxWorks is a kernel mode addressing real-time operating system, but use of features bundled with the operating system or provided by the OS Extension **VxVMI** affords developers important basic memory protection features. These features, including write-protection of text segments and of the interrupt vector table, can prevent errant or poorly designed applications from having harmful or disastrous effects on the embedded system. In addition, developers can detect buffer overruns and underruns, protect private data from corruption, and provide secure shared data.

Use of basic MMU support and **VxVMI** can help developers using **VxWorks** for embedded system software to improve the efficiency and productivity of the debugging phases of development, reducing both development risk and time to market, and to improve system reliability and robustness, improving overall device quality.

WIND RIVER

Wind River Worldwide Headquarters
500 Wind River Way
Alameda, CA 94501 USA
Toll free 1-800-545-WIND
Phone 1-510-748-4100
Fax 1-510-749-2010
Inquiries@windriver.com
Nasdaq: WIND

For additional contact information, please see our Web site at www.windriver.com.