Porting VxWorks Applications to Linux Pthreads

A Brief Tutorial on

RTOS Legacy Code and

Embedded Linux

BETA Version

MONTAVISTA
SOFTWARE
1237 E. Arques Ave.
Sunnyvale, CA 94085

Copyright

Revision History

1.0	November 17, 2000	Beta Release

This document was written for MontaVista by Gary Robertson.

You are welcome to contact us: **sales@mvista.com**

Contents

Chapter 1: Introduction

Reasons for Porting VxWorks Applications to Linux

For many years now, VxWorks has been a mainstay of the embedded systems marketplace. It remains today one of the most popular embedded systems environments available. However, in recent years many embedded systems developers are becoming interested in using the Linux operating system as a platform for their embedded systems applications. Linux is typically desired as a platform for the embedded systems environment based upon:

- price
- stability
- open-source architecture
- standard interfaces
- networking support
- the wide range of supported peripheral devices

Unfortunately, in contemplating a move to this new open-source platform, the established embedded systems developer must often contend with large bodies of existing ‘legacy’ code developed for the VxWorks platform. To simplify the job of porting existing VxWorks applications to the Linux environment, MontaVista Software has created a VxWorks ‘virtual machine’ which implements most of the core features of the VxWorks kernel on a standard Linux platform.

The virtual machine presented here is emphatically not suitable in its present form as a direct replacement for VxWorks.

Goals and Scope of This Paper

This paper is intended to serve as a brief tutorial regarding the issues and solutions involved in porting VxWorks applications code to a POSIX threads environment as implemented in the latest stable version of Linux. We will assume the reader has a general knowledge of real-time applications design, but is perhaps not familiar with Linux or with POSIX threads (pthreads).

We will attempt to detail the differences between the VxWorks and pthreads programming environments. We will then explain the methods chosen to resolve those differences in implementing the VxWorks virtual machine, and to explain why those methods were chosen and how they work.

More information about the scope of this paper can be found in “System Calls Not Addressed and Other Caveats” on page 31.

Chapter 2: Porting from VxWorks to Linux and Improving Performance of Ported Code

Overview

To begin porting VxWorks applications to Linux, you will need:

- A Linux-based Development System (recommended)
- The Hard Hat Cross Development Kit from MontaVista (Available for download from ftp://ftp.mvista.com/pub)
- The VxWorks example header files and demo programs from MontaVista (Available for download from ftp://ftp.mvista.com/pub/VxWorks/)
- Target hardware
- VxWorks code to be ported

There are two main steps to port VxWorks code to Linux pthreads:

- Port the VxWorks code to Linux pthreads with minimal source changes.
- Modify the port for performance

The following sections provide additional information about the above steps.

Steps in Porting from VxWorks to Linux

Porting code from VxWorks (or any other RTOS) to Linux can be most efficiently accomplished via the following steps:

1. Download the Example Header Files and Demo Programs

Example header files and demonstration programs are available on the MontaVista FTP site at ftp.mvista.com/pub/VxWorks/ in the tar file, `VxWorks2Linux.tgz`. While not required, you may want to download this file before you begin, so you can refer to the example files as you develop your own.

2. Establishing a Linux Cross-Development Environment

The first step in porting code to a Linux environment is to establish a cross-development tool set. Unless the desired target hardware is one of the common desktop systems for which off-the-shelf Linux distributions are available, this step will usually consist of generating a GNU C and/or C++ compiler, assembler, and linker for the desired target CPU. Documentation on the procedures for doing this are available from the Free Software Foundation. MontaVista Software provides tools and expertise for facilitating this process, and may be of assistance in foreshortening this step of the porting process. For more information see http://www.mvista.com.

3. Establishing a Linux Environment on the Target Hardware

Once a cross-development tool set is available, the Linux kernel and an application-dependent subset of available daemons and system utilities must be ported to the target hardware. A requisite set of device drivers must also be generated. In many cases drivers will already be available and simple recompilation may suffice, but occasionally new device drivers will have to be written. This is especially true if the target system contains vendor-specific hardware. MontaVista Software specializes in providing tools, components, and expertise in porting the Linux operating system to new target hardware. Their experts can provide any assistance needed in establishing a viable Linux environment on custom target hardware. For more information see http://www.mvista.com.

4. Re-compiling the Target Code for the New Environment

With a viable tool set and a target-resident Linux environment established, the next step is to eliminate any compile-time problems in the target applications code. This will involve (at a minimum) the resolving of differences in header file contents and location, as well as function prototypes and word-length issues. Because VxWorks implements many POSIX, UNIX, and standard C runtime calls in its native environment, it supplies an extensive list of header files to support these libraries. Unfortunately, a number of these header files conflict with their Linux equivalents. The VxWorks virtual machine addresses this by creating local header files to satisfy requisite VxWorks definitions and prototypes, and then dispensing with the use of the normal VxWorks include file directory tree. An alternative would be to identify the conflicting VxWorks headers and to replace them in the VxWorks header directories by overwriting them with their Linux counterpart headers. In some cases, conflicts may arise between Linux system header files and the ‘local’ header files, or between

the function prototypes in the Linux system headers and the legacy code being ported. In this case, either the source code or the local header files should be modified as necessary to resolve the conflicts. In extreme cases it may be necessary to eliminate inclusion of the conflicting headers and then copy in only those definitions from the offending header file which are actually referenced in the source code. In any case, the standard Linux headers should never be modified to resolve such conflicts.

5. Eliminating Undefined Linker References

Once a clean compile has been achieved for all source code modules, the build process will probably begin to fail due to unresolved external references. These unresolved references will generally be due to functions present in the legacy RTOS environment which have not been duplicated in the Linux virtual machine. The options for resolving these are:

- to extend the virtual machine environment by implementing the referenced RTOS functions,
- to substitute a native Linux function whose functionality approximates that of the missing RTOS function, or
- to re-code the application in such a way as to eliminate the need for the missing RTOS function.

In the case of missing VxWorks kernel calls, extending the virtual machine environment may be the preferred alternative. For most VxWorks-specific calls used in initializing and manipulating devices, filesystems, or networking stacks, the application can be re-coded to eliminate the call and/or equivalent native Linux manipulation functions may be used instead. Most POSIX or UNIX-style calls implemented in VxWorks have native Linux equivalents which are functionally identical.

6. Debugging the Ported Code

Most C code unfortunately contains what might be considered abuses of the language, and these create problems when porting the code. This is a consequence of the freedom to ‘bend the rules’ which makes C an attractive language for systems programming in the first place. Programming tricks which enhance performance or facilitate coding in one environment often ‘break the code’ when transported to another compiler or CPU architecture.

The most common problems which arise in porting have to do with differences in integer fetch width and the resulting possible problems with integer ranges. Code which validates integer ranges or handles integer overflow is likely to need attention if the integer size changes between targets or compilers.

Dependencies based upon ‘endian-ness’ are the next most frequent problem. Bit field variables and enumerated variables are often broken by some combination of word length and ‘endian-ness’ differences. (‘Endian-ness’ refers to the order of byte significance in multi-byte scalar data. ‘Little-endian’ systems place the least-significant bytes of a multi-byte scalar at the lowest numeric addresses, while ‘big-endian’ systems place the most significant bytes at the lowest numeric addresses.)

Another common porting problem results from assumptions regarding the internal layout of structures. The C language allows the compiler vendor great liberties in aligning members within structures. Depending upon the compiler and the CPU

architecture, there may be ‘holes’ or ‘padding’ placed strategically between structure members in order to optimize data alignment. These alignments often differ from one compiler and CPU family to another. This means code which assumes knowledge of internal structure layout is likely to break when ported. All ‘union’ data types should be closely scrutinized when porting to ensure that they will still work in the new environment.

Source-level debugging may not be available on the target hardware, or in some cases may cause the code to function differently during debugging than when ‘free-running’. On the 80x86 Linux platform it is often necessary to insert `sleep()` statements at the start of a newly created VxWorks task when using `xxgdb`. Furthermore, in many real-time designs, suspending one task for debugging prevents that task from the timely performance of activities vital to the operation of the rest of the system, causing unpredictable and/or undesirable results. For this reason it is often preferable to fall back to the tried and true method of instrumenting the code with ‘diagnostic `printf()`’ trace statements which will report the activities of the code at critical junctures. These trace statements can be controlled via conditional compilation or even through runtime diagnostic option settings. Even when using these trace statements, discretion is advised. The `printf()` call entails some significant overhead in itself, and overdoing these can in some cases slow real-time performance to less than tolerable levels.

7. Refining the Ported Code to Improve Performance

Once the application has been ported and its basic functionality has been verified in the new Linux environment, it may be desirable to enhance the real-time performance of the ported code. If the application was a good candidate for porting to Linux in the first place, then it should not require massive changes in order to bring it to acceptable performance levels. Techniques for improving the performance of the ported application are discussed in the next section.

Examples

Included in the tar file, `VxWorks2Linux.tgz`, that you downloaded in step one above are two examples, `demo` and `validate`.

Example: `Demo`

The first example, `demo`, demonstrates how to use the emulation code to convert the following:

- semaphores
- tasks to threads
- memory partitioning
- queuing

The `demo` example illustrates a producer/consumer problem. There is one producer and two consumers. The producer waits until one or both of the consumers is ready to receive a message (indicated by event flags) and then posts a message on the mailbox of that consumer. Messages consist of memory blocks allocated from two partitions (one per consumer). The producer allocates a block, writes data into it and then sends the block to the consumer. The consumer reads the data and then releases the block.

Example: `validate`

The second example, `validate`, tests and validates the following:

- events
- standard queues
- variable length queues
- semaphores
- partitions
- tasks

The `validate` example tests the above calls by emulating the VxWorks environment. It then tests the functionality of each of the features to be sure that they work.

Techniques for Improving Performance of Ported Code

If it is determined that the ported application code does not perform adequately well to meet the requirements of the product design, a number of techniques may be employed to enhance performance. Often significant performance gains may be realized by analyzing the design to determine what code is executed most frequently and then concentrating on optimization of that code. This analysis of what code is used most often, and what functions are the most time-consuming is called 'profiling'. The GNU compiler tool set and run-time libraries may be compiled to support profiling. Alternatively, the profiling can be done 'by hand' by instrumenting key functions with profiling code. This code would count the number of times the functions are called and make the counts available for later analysis.

1. Enhancing Efficiency of the VxWorks Virtual Machine Code

Much of the functionality of the VxWorks virtual machine layer is fairly efficient for the functionality it provides. The principal source of overhead in the VxWorks virtual machine layer lies in the processing of linked lists associated with the control blocks for various objects. If the number of tasks, queues, etc. in the application design is small, the inefficiency of this list processing may be negligible. However, if there are many tasks, etc. in the application design, linked list overhead could pose a serious performance bottleneck. With the exception of tasks, all control blocks are typically referenced by identifiers containing the address of the control block. However, linked lists are used to ensure that 'stale' control block pointers do not attempt to reference objects which have been dynamically deleted from the VxWorks virtual machine environment. If no objects are dynamically deleted in the applications software, this validity checking can be eliminated.

2. Eliminating 'Bulletproofing' Code After Application Testing

Scattered throughout the VxWorks simulation code are strategically placed `pthread_cleanup_push()` and `pthread_cleanup_pop()` pairs. These macros are intended to guarantee that pthreads will not be terminated while holding exclusive use of critical shared resources. They cause functions to be called upon thread termination which would release such resources. If the developer studies the functionality of these calls carefully and is confident that none of the pthreads in the ported application will be prematurely terminated or aborted, some performance gains may be realized by eliminating these safeguards. A distinction should be made between those cleanup macros which insure against premature termination as opposed to those which facilitate deliberate task deletion. The 'premature termination' safeguards could be surrounded by conditional compilation constructs which would eliminate them once the application is sufficiently validated.

3. Adapting VxWorks Code to Native Linux Code

Another option for performance enhancement is to re-design VxWorks code to eliminate the overhead of the simulation layers. This could be done only for those portions of the application identified by profiling, or for the entire application design.

Aside from the overhead associated with locating the control blocks for tasks and other objects, most of the overhead in the VxWorks simulation code results from the requirements to allow multiple tasks to wait on a single object, and to allow tasks to be awakened either in FIFO or priority order. Each VxWorks task, queue, and semaphore maintains a list of tasks which are waiting on the object. The maintenance and use of this list requires a fair amount of overhead and complexity. To eliminate this overhead, it would first be necessary to ensure that only a single pthread would wait on a given VxWorks 'lightweight object' at any one time. Such objects could then employ much simpler logic for awakening waiting pthreads by simply broadcasting a signal for their associated condition variables. Any VxWorks task waiting on such objects could then assume that any signal to the condition variable associated with that object was for the current task. If the modified 'send' and 'wait' calls supporting these 'lightweight objects' use the address if the object control block directly, then the calling tasks would need no lookup scheme to locate these objects.

A VxWorks task can be converted to a native pthread in order to eliminate overhead associated with task control block lookups if:

- the task waits only on 'lightweight objects' as described above or on native pthreads synchronization and communication primitives, and
- other tasks do not interact with the task in a manner requiring their access to the task's control block (for example, VxWorks task events or priority changes), and
- the task does not call VxWorks functions which would reference its own task control block (for example, timer calls or task mode changes).

4. Re-coding Frequently-called Functions in Assembly Language

Compilers by nature must produce assembly language code which is fairly generic in nature. Late in the compilation process, optimization routines may execute which trim the generated instructions with varying degrees of efficiency. Normally, however, an accomplished assembly language programmer can produce more efficient code than even a well-optimized compiler. Dramatic performance gains can sometimes be accomplished by re-writing frequently executed code directly in assembly language. This should be approached by first identifying those frequently executed routines via profiling as described earlier. Each of the candidate routines identified by profiling should then be compiled (with all compiler optimizations enabled) to assembly language source. The resulting assembly language source code should then be analyzed for efficiency. If it is determined that significant improvements could be made in the efficiency of the code, it is then 'hand-optimized' by an assembly language programmer. The original C or C++ sources are then removed and replaced by the optimized assembly language source code files. The makefiles and build process are altered as required to employ the assembly language sources in lieu of the C or C++ sources, and the application is rebuilt.

Chapter 3: Comparing the Linux Pthreads Environment to the VxWorks RTOS Environment

Overview

The following section highlights the major differences between the VxWorks and Linux pthreads environments. These sections cover:

- Task Scheduler Characteristics
- Applications Memory Map and Memory Utilization
- Inter-Task Communication and Synchronization Facilities
- Input/Output Device Access

Task Scheduler Characteristics

The VxWorks scheduler was designed to support hard real-time applications. As such it was engineered to provide brief and predictable worst-case response times to external ‘real-world’ events. The Linux scheduler, on the other hand, was designed to support optimized average-case performance in order to maximize overall system throughput. Brief and predictable worst-case response times were not a primary design constraint. As mentioned above, this means the standard Linux kernel may not be a good choice for systems where a missed response deadline might result in a catastrophic system failure. Having said that, however, it should be added that a properly trimmed Linux kernel running on a capable CPU architecture might well prove to be reliably fast to meet most real-time application requirements.

In VxWorks the basic unit of scheduling is called a task. In Linux the basic unit of scheduling is called a process. In the Linux implementation of POSIX threads (pthreads), each pthread competes for CPU time as though it were an independent process. In discussing scheduling, all of the above entities may be considered equivalent and will be referred to as ‘processes’.

The default scheduling algorithm in Linux is a ‘fairness’ algorithm in which all processes have the same basic priority level (0). The effective priority level of each process is then adjusted dynamically to ensure each process a ‘fair share’ of CPU resources. However, Linux does offer POSIX ‘real-time’ scheduling as well in its standard scheduler. This paper will ignore the ‘default’ scheduling and will deal only with the real-time scheduler characteristics.

Both VxWorks and the Linux real-time scheduler are based upon ‘static’ priority levels which are assigned when a process is initially started. These ‘static’ priorities may be adjusted ‘on the fly’ by explicit system calls, but are not automatically adjusted by the scheduler itself. The scheduler maintains a list at each priority level of processes which are currently ready to run. Each time the scheduler is entered, it searches these ‘ready lists’ in order from highest to lowest priority level. The first (highest priority) runnable process found in this search is the process to be given CPU control upon exit from the scheduler.

When a process performs an operation which causes it to wait on some event (called ‘blocking’), that process is removed from the ready list for that priority level. When the blocking operation completes, the process is placed at the tail end of the ready list for processes at its respective priority level. This allows any other processes at that priority level to move up to the head of the ready list for that level and to get CPU time eventually. This scheduling policy is called FIFO (First-In, First-Out) scheduling.

FIFO scheduling may result in some processes being starved for CPU time if other processes at their priority level fail to block often enough. To overcome this problem, a second type of scheduling exists in both VxWorks and the Linux real-time scheduler. In this policy, each process is given a finite amount of time (called a quantum) to control the CPU. At the end of this quantum the process will be preempted and moved to the tail of the ready list for its priority level. At this time the next runnable process at the same priority level is given CPU control. This guarantees each process within a given priority level equal access to the CPU. This scheduling policy is called ‘Time Slicing’ or Round-Robin scheduling. Except for the quantum-based preemption within priority level, Round-Robin scheduling is identical to FIFO scheduling.

If an event occurs which causes a higher priority process to become runnable, the currently-executing process will be immediately preempted. When all higher priority processes are blocked, a preempted FIFO-scheduled process resumes execution at the head of the ready list for its priority level. A preempted Round-Robin-scheduled process will resume execution until the remainder of its CPU quantum is consumed.

In VxWorks, facilities are provided whereby an asynchronous service request from an external device (an interrupt) can result in the scheduler being entered. This in turn can result in an a new process being made runnable and preempting the process which was executing when the interrupt occurred. This capability for an external event to trigger a process context switch via an interrupt service routine is missing from Linux. Interrupt service routines are completely ‘transparent’ to the Linux scheduler and do not directly trigger a process context switch.

In general, the characteristics of the VxWorks task scheduler are qualitatively comparable to those of the Linux real-time process scheduler. Quantitatively, however, the standard Linux scheduler may not provide optimal performance, and may be improved upon by adding a new real-time scheduler available from MontaVista Software. MontaVista Software also has kernel enhancements available for uni-processor kernels which add preemptability to some portions of the kernel itself, further improving potential real-time responsiveness.

Applications Memory Map and Memory Utilization

Earlier it was noted that for purposes of scheduler activities, processes, tasks, and Linux pthreads could be considered as synonymous. This viewpoint will now be amended for the remainder of this document. Processes, tasks, and pthreads each represent independent threads of program execution by the CPU. In Linux, each process obtains its own unique address space. Linux pthreads and VxWorks tasks share a common address space for instructions and for global data, but each pthread or VxWorks task has its own stack space and local variable space. The memory model for a Linux pthread is therefore very similar to that for a VxWorks task, but that of a Linux process is very different.

In the VxWorks environment, memory addresses are normally statically bound to fixed locations at link/load time. In the Linux environment, all addresses are dynamically determined when the programs are loaded. This means that constants pointing to objects or data at fixed addresses generally cannot be used.

In a VxWorks environment, there are explicitly defined pools of memory which are used for both static and dynamic memory allocation by the system. The various system control structures, as well as message buffers, etc. are allocated from these memory pools. VxWorks provides a suite of system calls which are used to manipulate these memory pools in a reasonably efficient and deterministic manner. The Linux VxWorks virtual machine dispenses with explicit memory partitioning and instead uses thread-safe `malloc()` and `free()` functions for managing memory. This use is mostly confined to the system calls used to create and delete various VxWorks objects, so it would be of little concern if the VxWorks application does not dynamically create and delete tasks, queues, partitions, and the like. However, the timing of `malloc()` and `free()` calls tends to be highly non-deterministic, so applications which do engage in frequent creation and deletion of VxWorks objects might want to consider an alternative technique.

Inter-Task Communication and Synchronization Facilities

In a multi-tasking environment, the various concurrently executing tasks must be carefully coordinated to avoid deadlocks, race conditions, and conflicts in utilization of shared resources. In order to facilitate this coordination, two generic classes of inter-task communication are employed. They are:

- synchronization mechanisms
- mutual exclusion mechanisms

Synchronization mechanisms allow one thread of execution to wait until another thread of execution satisfies some mutually agreed-upon condition before the first thread proceeds. Mutual exclusion mechanisms safely secure the exclusive use of a shared resource by a single thread of execution, excluding all other threads from accessing the resource until the current exclusive user releases it for re-use.

VxWorks provides ‘binary semaphores’ and ‘counting semaphores’, which can serve as either synchronization or mutual exclusion primitives. It also provides special ‘mutex semaphores’ which are optimized for mutual exclusion. Finally, it provides ‘message queues’, which combine implicit mutual exclusion and synchronization features to pass shared data between tasks. Tasks can query these primitives in a non-blocking call (i.e. without waiting), a timeout-bounded blocking call, or a call which blocks indefinitely until the requested resource state is satisfied.

The POSIX threads environment provides mutual exclusion primitives called ‘mutexes’. It also provides counting ‘semaphores’ for use in synchronization or mutual exclusion. In addition, it provides a synchronization mechanism called a ‘condition variable’. For each of these primitives, pthreads can query the resources in either a non-blocking call or a call which blocks indefinitely until the requested resource state is satisfied. However, only condition variables offer the option of a timeout-bounded wait on the requested resource state.

VxWorks provides support for the above POSIX features as well as POSIX.1b named semaphores, message queues, and timers. Linux currently does not provide support for these features, partly because they overlap SYSV IPC features to be described next. However, efforts are underway to add POSIX1.b timers and message queues to Linux.

The Linux environment offers System V inter-process communications primitives (SYSV IPC) which include ‘named pipes’ or ‘FIFOs’, a more robust flavor of counting ‘semaphores’, and inter-process ‘message queues’. Use of these resources involves a number of complexities, however, and they were not used in the current implementation of the VxWorks virtual machine.

Both Linux and VxWorks support some form of ‘asynchronous signal’ mechanisms which can provide ‘software interrupts’ to running tasks; however, the details of these facilities will not be addressed in this paper.

Input/Output Device Access

Both VxWorks and Linux generally follow the UNIX model of treating peripheral devices as special types of files, and using device drivers and I/O subsystems which support this abstraction. From a task, process, or pthread perspective all I/O is performed using files.

Because of its embedded systems heritage, VxWorks requires a substantial amount of explicit definition and initialization for its I/O subsystem. Consequently, VxWorks supplies a copious suite of configuration and initialization support routines which are intended to be used during system startup. Some of these routines are specific to a given type of peripheral hardware, while others are specific to filesystem or protocol types. The embedded systems developer must specify which devices, filesystems, and protocols are to be included in the system.

This is accomplished using special configuration header files, and must ultimately be reflected in the system linker command files which specify which VxWorks library components are to be included in the system. During the initialization process, the programmer may have to make explicit initialization and configuration calls to prepare the system hardware for operation. Furthermore, for devices which are not already supported by VxWorks, the programmer may be required to write new device drivers. VxWorks does provide example code and reference material to assist with this effort when a new driver is required.

Linux, on the other hand, was designed to configure devices, filesystems, and protocols more or less transparently to the end user. In an embedded systems environment, the user would employ kernel configuration scripts to include support only for those devices, filesystems, and protocols desired. Beyond that, the system's device initialization is largely automatic. As with VxWorks, if Linux does not provide support for a specific device, the programmer may be required to write a new device driver. Fortunately, Linux has perhaps the widest array of device support of any Unix-style operating system. Furthermore, the source code for the drivers is readily available, as is documentation and/or advice and assistance.

Chapter 4: Supporting VxWorks System Calls in the Linux Pthreads Environment

Overview

As mentioned earlier, it was necessary to implement a layer of adapter functions and data structures on top of the native pthreads environment in order to emulate VxWorks functionality in a Linux environment. The sections below detail the methods and rationale used in the current implementation of a VxWorks virtual machine.

It was felt that the insertion of large bodies of source code into the midst of this document would render the rest of the document to be largely incomprehensible. The source code referenced here is available for download from MontaVista Software, Inc. (www.mvista.com), and it is recommended that the source code and this document should both be examined together. The source code is profusely commented, and it is hoped that this paper satisfactorily explains the overall rationale for the design which is missing from the source code itself.

It should be emphasized once more that the initial design focus was functionality-oriented rather than performance-oriented, and that alternative techniques would likely offer better performance. It should also be noted that the source code submitted with this paper is a work in progress. It is made available in the hope that it may prove useful to other developers who are considering a port of their legacy VxWorks applications to a Linux platform.

In the startup code for the VxWorks virtual machine, the parent process establishes characteristics for all its child pthreads, locking their address spaces into memory to prevent delays due to their memory being swapped to and from disk. It then creates the VxWorks 'root task', which executes at priority level 99. The root task first creates the system 'exception task, which executes at priority level 98 and is responsible for handling VxWorks watchdog timer processing. The root task next calls a user-defined function named `user_sysinit()`. `user_sysinit()` is roughly analogous to the VxWorks `usrInit()` routine, and is expected to contain the logic for initial startup of the VxWorks applications. However, `user_sysinit()` differs from `usrInit()` in that the multitasking environment is already active when it is called, and thus `user_sysinit()` may freely employ any VxWorks facility as desired. Typical uses for `user_sysinit()` would include creation and startup of other VxWorks tasks and creation of queues, and semaphores. After `user_sysinit()` returns, the root task enters an infinite dormant state. Meanwhile, after having created the root task, the parent process calls a user-defined

function named `user_syskill()`. When and if `user_syskill()` returns, the entire VxWorks system 'shuts down', so normally `user_syskill()` will contain an infinite delay loop which optionally contains an exit test for the virtual machine environment. All Linux resources used by the VxWorks virtual machine are cleaned up by the operating system when the virtual machine exits.

It may be worth noting at this point that the real-time scheduling used in the VxWorks virtual machine requires that the virtual machine runs with an effective user ID of 'root'. That is, it must either be run by the root user or must created to run as 'setuid root'. Furthermore, since these tasks run at priority levels above that of all 'normal' user-space Linux processes (including X-windows and standard terminal logins) it may be difficult to obtain a command line prompt in the event that a real-time process under test gets into a runaway loop. It is advisable to have at least one terminal window running at priority level 99 in order to guarantee the ability to abort a runaway VxWorks virtual machine. MontaVista Software has freely downloadable tools which will enable the pthreads developer to set up such a terminal.

Task Management Calls

VxWorks tasks in the virtual machine are mapped onto 'real-time' POSIX threads. A wrapper function is used for the task code which provides several benefits:

- It converts between the function type expected by pthreads and the function type expected for a VxWorks task.
- It provides a means for passing the parameters which can be passed to a VxWorks task on startup.
- It provides a cleanup of the scheduler-locking mutex if the VxWorks task should exit and return unexpectedly.
- It deletes the task control block if the VxWorks task should return unexpectedly.

Task Creation and Activation

The VxWorks virtual machine provides the `taskInit()` call to initialize the data structures associated with a task without actually starting task execution. The `taskInit()` call also allows the caller to specify an explicit address for the task's stack and control block structure; however the virtual machine currently ignores the stack address specification. The caller's VxWorks priority level (between 0 and 255) is translated as cleanly as is possible into a Linux real-time priority between 1 and 97. The scheduling policy (FIFO or Round-Robin) is established for the task's pthread. Finally, the new control block is linked into a list of task control blocks for all VxWorks tasks in the virtual machine environment.

A second function, `taskSpawn()`, is normally used to both initialize and activate a new VxWorks task. The virtual machine's implementation of `taskSpawn()` allocates space for a task control block structure as defined in `vxwk2pthread.h`, using a thread-safe `malloc()` call. If no task name was supplied by the caller, a name is fabricated automatically. Next `taskSpawn()` calls `taskInit()` to initialize the various members of the control block structure. The stack size specification and option flags arguments are meaningless in the virtual machine and are ignored. After the task control block is initialized, `taskSpawn()` calls `taskActivate()` to begin task execution.

The `taskActivate()` function activates a pthread for the specified VxWorks task. It activates the task wrapper function as a new pthread, passing it the caller's parameter block. The wrapper function then converts the format of the parameter block and calls the VxWorks task entry point (which normally does not return to the wrapper function).

Task Suspension, Resumption, and Restarting

Linux pthreads are treated by the kernel as separately schedulable processes with some special characteristics which distinguish them from normal processes. A consequence of this is that pthread signal handling under Linux differs somewhat from what might be expected in other POSIX pthreads environments. This difference was exploited in the virtual machine's implementation of the VxWorks `taskSuspend()` and `taskResume()` calls. The `taskSuspend()` call sends a SIGSTOP signal to the pthread running the specified VxWorks task, causing the execution of the pthread to pause indefinitely. The `taskResume()` call sends a

SIGCONT signal, causing execution of the paused pthread to resume at the point where it paused earlier.

The VxWorks virtual machine also provides a means of suspending a task for a specific delay interval. The `taskDelay()` call takes an argument which defines an interval in terms of VxWorks scheduler clock 'ticks'. The interval width of this 'clock tick' is a constant defined in the VxWorks setup and initialization logic on a native VxWorks system. In the virtual machine the interval is defined by a symbolic constant called `VXWK_TICK`. This constant is defined in the `vxwk2pthread.h` header file, and is currently set to 10 milliseconds. `taskDelay()` in the virtual machine is implemented as a loop which invokes the `usleep()` system call for a specified number of microseconds equivalent to the length of the interval specified by the caller. The loop is necessary, as mentioned before, because the system call may return prematurely due to an unexpected signal.

Prior to starting the loop, an absolute time value is calculated at which the delay interval should expire. Each time the `usleep()` system call returns, a new delta in microseconds is computed between the time of the return from `usleep()` and the predetermined absolute time. `usleep()` is then called once more with the new remainder of the original interval.

A call to `pthread_testcancel` at the beginning of the loop provides an exit point for the thread if it is killed during the delay interval. **On 'x86' Linux, the minimum possible delay appears to be 20 milliseconds, with increments of ten milliseconds thereafter. Thus, a delay of 1 'tick' would equal approximately 20 milliseconds, while 2 'ticks' would equal 30 milliseconds and 10 'ticks' would equal 110 milliseconds.**

Two status querying functions indicate the current scheduling condition of a task. `taskIsSuspended()` returns a nonzero value if the specified task has been explicitly suspended by a `taskSuspend()` call, and a zero otherwise. `taskIsReady()` returns a nonzero value if the specified task is ready to run (i.e. not suspended, delayed, or blocked) and a zero value if the task is not currently runnable.

The VxWorks virtual machine also provides a `taskRestart()` call, which terminates task execution and then restarts it from the task entry point, using the task's operating characteristics and parameters as they existed at the time the restart call was invoked. In the VxWorks virtual machine, this is accomplished by terminating the pthread currently running the VxWorks task and then starting a new pthread using the same task control block. If the task is restarting itself, a watchdog timer function is used to start the new pthread after the calling pthread terminates. Note that after a task restarts itself, there may be some delay between the start of code execution in the new pthread and the updating of the pthread identifier in the task control block. During this interval, any code called by the VxWorks application which attempts to manipulate the characteristics of the new pthread may fail. On Linux 'x86' systems, a call to `usleep( 10000L )` at the beginning of the task which restarts itself appears to safeguard against problems associated with the task restart. This results in a 20 millisecond (not ten!) delay to allow the pthread identifier to be initialized. Meanwhile, no other tasks should attempt to reference a task which restarts itself, since such access would be asynchronous to the restart operation and might subsequently fail.

Task Priority and Preemption Management

The VxWorks virtual machine provides the calls `taskUnlock()` and `taskLock()`, respectively, to make the task preemptable or non-preemptable. In Linux, priority level 99 is the maximum permissible priority for real-time processes or pthreads. In the virtual machine, this priority level is reserved for 'non-preemptable' sections of code. Thus, preemption by other tasks is avoided by temporarily setting the current pthread's priority level above that of all other VxWorks task pthreads. This same 'scheduler locking' mechanism is used during many of the virtual machine's VxWorks system calls to preclude their preemption temporarily.

A mutex and condition variable are used to guarantee exclusive access to the 'scheduler locking' mechanism by only one pthread at a time. A lock ownership identifier variable and a lock nesting level variable, protected by the mutex, are then used to allow recursive 'locking' and 'unlocking' of the scheduler by a given pthread. This allows a pthread which has already become non-preemptable to make additional calls to routines which contain non-preemptable sections, and prevents the pthread from being made prematurely preemptable on exit from one of these routines. At the same time, it prevents other pthreads from simultaneously acquiring non-preemptable status.

The virtual machine provides the calls `taskPriorityGet()` and `taskPrioritySet()`, respectively, to examine and modify the execution priority level of a task. The `taskPrioritySet()` function maps the specified VxWorks priority level to a Linux pthreads priority level according to the same rules used during the taskInit call. Then it modifies the scheduling parameters in the attributes structure for the pthread executing the specified VxWorks task.

Task Identification and Verification

VxWorks gives each task a character-string name upon creation. Each task also obtains a task identifier. If the task is created using `taskSpawn()`, the current virtual machine uses a simple index number as an identifier; if taskInit was called explicitly to create the task control block, the address of the task control block is used as the identifier. The task verification and identification functions traverse the linked list of task control blocks until they find a TCB with a member matching that specified by the caller. Information implied by the actual function called is then returned to the caller for the matching TCB.

- `taskNameToId()` returns the task identifier number corresponding to the task name specified by the caller.
- `taskName()` returns the task name corresponding to the task identifier specified by the caller.
- `taskIdVerify()` indicates whether a task currently exists with the specified identifier.
- `taskIdSelf()` returns the task identifier for the calling task.
- `taskIdListget()` returns an array of task identifiers to the caller.
- `taskTcb()` returns a pointer to the task control block for the task identifier specified by the caller.

Task Deletion

The VxWorks virtual machine allows tasks to enter a ‘deletion-safe’ condition during which they may not be deleted by other tasks. A task can call `taskSafe()` to enter a ‘deletion-safe’ condition while it holds some critical resource. Later, the task can call `taskUnsafe()` to become deletable again. The `taskDeleteForce()` call ignores the deletion-safe condition and deletes the targeted task at once. The `taskDelete()` call respects the deletion-safe condition of its targeted task, and will block its calling task when attempting to delete a task which is in a deletion-safe condition. When the task targeted for deletion re-enters a deletable condition, the deletion operation will complete and any blocked tasks will be awakened.

Actual deletion of a VxWorks task in the Linux pthreads virtual machine begins by awakening any tasks which are blocked waiting on the deletion. Next it is determined whether the current task is deleting itself or is deleting another task. Self-deletion begins by ‘detaching’ the pthread which is running the VxWorks task to be deleted. POSIX threads can exist in either the ‘joinable’ state or the ‘detached’ state. A given pthread or process can wait for a joinable pthread to terminate. At this time, the process or pthread waiting on the terminating pthread obtains an exit status from the pthread and then the terminating pthread’s resources are released. A ‘detached’ pthread has effectively been told that no other process or pthread cares when it terminates. This means that when the ‘detached’ pthread terminates, its resources are released immediately and no other process or pthread can receive termination notice or an exit status. If the pthread is deleting itself it must be ‘detached’ in order to free its Linux resources upon termination.

If the current task is deleting another task, the target task’s pthread is left in the ‘joinable’ state. In either case, the target pthread is terminated. If the current task is deleting itself, it executes a `pthread_exit()` call. If another task is being deleted, a `pthread_cancel()` call is executed for the pthread running that task. Either of these calls sets an internal flag so that the specified pthread will voluntarily exit when it reaches its next ‘cancellation point’. These ‘cancellation points’ have been built into each of the VxWorks system calls, so this means the task will terminate as soon as it executes its next VxWorks system call. This allows each pthread to perform a ‘clean’ shutdown, without leaving mutexes locked, etc. Special routines called ‘cleanup handlers’ were attached to each pthread at various strategic times during its execution to ensure that any locks owned by the pthread would be released if the pthread were killed before releasing the locks. These handlers are executed as the pthread terminates, cleaning up any locks it held. If the executing pthread was not deleting itself, then it calls `pthread_join()` and waits for the termination of the target pthread to complete. After its pthread is terminated, the task being deleted is removed from the waiting lists of any queues or semaphores and its task control block (TCB) is deleted.

Semaphore Management Calls

Semaphore Creation

The Linux VxWorks virtual machine defines three separate types of semaphores. They are:

- binary semaphores
- counting semaphores
- mutex semaphores

Each of these semaphore types has its own creation function; all other semaphore functions are common to all semaphore types. Semaphore creation in the virtual machine begins by using a thread-safe `malloc()` call to allocate space for a new semaphore control block. Next, it initializes the mutexes and condition variables associated with semaphore access and deletion, and the semaphore token count is initialized with the number of available tokens (zero or more) specified.

`semBCreate()` creates a simple binary semaphore. The initial token count can be either 0 (locked) or 1 (available). A flag specified by the caller indicates whether tasks waiting on this semaphore will be awakened in FIFO order or in priority order. Binary semaphores do not allow recursive `semGet()` and `semGive()` operations, and do not test for ownership when `semGive()` is called. Thus it is incumbent on the user to guarantee that a task only calls `semGive()` for binary semaphores it currently has 'locked'.

`semCCreate()` creates a counting semaphore. The initial token count can be 0 to 32767. A flag specified by the caller indicates whether tasks waiting on this semaphore will be awakened in FIFO order or in priority order. Counting semaphores do not test for ownership when `semGive()` is called, and do not return an error if `semGive()` is called more times than `semTake()`. Thus it is incumbent on the user to guarantee that a task only calls `semGive()` for counting semaphores from which it currently has acquired one or more tokens.

Semaphore Identification

Semaphores, message queues, and watchdog timers as implemented in the Linux VxWorks virtual machine all have identifiers which are pointers to the control blocks for the objects. This should enable efficient access to the associated objects; however, the fact that these objects can be dynamically deleted from the system creates a possibility that an identifier might point to an object which no longer exists. Consequently, a linked list of pointers to valid control blocks is searched each time a semaphore or other object is accessed. If the specified control block is found in the list, the POSIX mutex associated with access to the semaphore or other object is locked to guarantee exclusive access to the object, and a nonzero result is returned to indicate the object identifier is current and valid.

Acquiring a Semaphore Token

The same basic mechanism is employed for all inter-task communication in the Linux VxWorks virtual machine. It involves a POSIX.1b mutex and related condition variable. This mechanism was chosen because it is the only POSIX.1b mechanism which offers support for timeout-bounded waits. The mutex guarantees exclusive access to the object in question. The receiving task is waiting for some set of

conditions (called a ‘predicate’) related to the object in question to be satisfied. The condition variable is the mechanism for signaling the receiving task(s) of a change in the state of the predicate. The receiving task first waits until it can acquire and lock the mutex. This implicitly guarantees that the receiving task will not test the predicate conditions while they are being modified by the sending task. The receiving task then enters a loop in which it first tests the state of the predicate and then (if the predicate conditions are not yet met) waits on the condition variable until either a sending task signals a change or a timeout expires. (If the wait is indefinite then timeout expiration is not used as an exit condition for the loop.) The `pthread_cond_timedwait()` system call unlocks the mutex, enabling the sending task to lock it and modify the predicate conditions.

If a change is signaled on the condition variable, the `pthread_cond_timedwait()` system call first locks the mutex and then returns to the loop to test the predicate once more. This loop is required since it is not guaranteed that the change made to the predicate state meets the requirements of the receiving task, and also because some other type of Unix signal may interrupt the `pthread_cond_timedwait()` system call, causing a premature return. Eventually either the predicate condition is met or the timeout expires, and the receiving task exits the loop with the mutex locked. At this time the receiving task makes whatever response is appropriate for the specified predicate conditions and then releases the mutex again.

In the case of semaphores, the logic described above occurs in the `semTake()` call. Multiple tasks may acquire tokens from a given semaphore, and at any given time there may be multiple tasks waiting (and contending) for a token to become available on a semaphore. In this scenario either the first task to begin waiting for a token or the highest-priority task waiting for a token will be given the new token when it becomes available. Which task is selected to receive the token depends on a queuing order option specified during semaphore creation. A timeout argument to the semTake() call indicates whether the receiving task is to wait for a token, and if so, for how long.

The waiting list for a semaphore is organized as a linked list of pointers to the task control blocks for each task waiting on the semaphore. The list is organized in FIFO order as each task begins waiting on the semaphore, and is terminated by a NULL TCB pointer. The list head is contained in the semaphore control block, and points to the task control block of the first waiting task. The task control blocks themselves each contain a link to the next task waiting on the semaphore. Each task control block also contains a `suspend_list` pointer to the semaphore control block for the semaphore it is waiting on. This `suspend_list` pointer is used to remove the task from the semaphore’s waiting list if the task should be deleted or killed while waiting on the semaphore, or if the semaphore itself should be deleted while the task is waiting on it.

Multiple tokens may be released back to a semaphore before the task or tasks waiting on the semaphore get their turn for CPU time again. In this case, the sending tasks will have signaled the condition variable for the semaphore many times; however, each task waiting on the semaphore will receive only one guaranteed notification that tokens have become available on the semaphore. This means that the predicate condition test must consider all tokens in the semaphore on each pass through the query loop.

As long as available tokens remain in the semaphore and the timeout (if any) has not expired, a nested inner query loop checks to see if the current task is selected to

acquire the next token from the semaphore. If the next token is not for the current task, the inner loop temporarily unlocks the mutex and suspends the current task for a short delay. This allows other receiving tasks of equal or lesser priority an opportunity to obtain their tokens. After the short delay expires, the current task re-locks the mutex and returns to the top of the inner query loop. Upon timeout expiration or successful acquisition of a token, the inner query loop exits and the receiving task removes itself from the waiting list for the semaphore.

Releasing a Semaphore Token

Releasing a token back to a semaphore is done using a `semGive()` call. The Linux VxWorks virtual machine begins `semGive()` by locking the mutex for the semaphore to prevent other tasks from accessing it while it is being modified. Next the token count on the semaphore is incremented. After the token has been released to the semaphore, the condition variable for the semaphore is signaled with a `pthread_cond_broadcast()` call and the mutex for the semaphore is unlocked. This awakens any waiting tasks from their `pthread_cond_wait()` calls, and the tasks "arbitrate'' to determine which one ultimately obtains the token just released. For binary and counting semaphore types, no check is made for ownership or for excessive `semGive()` operations, so the caller must ensure that it legitimately 'owns' the token before releasing it. For binary and counting semaphores, the `semFlush()` call may also be used to release a semaphore token. The `semFlush()` call effectively grants a token from the 'flushed' semaphore to all tasks waiting on the semaphore simultaneously.

Semaphore Deletion

The semaphore delete operation is done via the `semDelete()` call. In the virtual machine environment, `semDelete()` begins by 'locking the scheduler' and checking the waiting list for the semaphore to determine if any tasks are currently waiting on the semaphore. If any tasks are waiting on the semaphore, the deletion mutex and also the main mutex for the semaphore are locked to prevent other tasks from accessing it during the deletion operation. Next the delete operation sets a special flag in the semaphore control block to indicate the semaphore is being deleted. After this, it signals the condition variable to alert the waiting tasks to the deletion. Having done this, it blocks the calling task with a `pthread_cond_wait()` on a `deletion_complete` condition variable for the semaphore. This allows the waiting tasks to receive the semaphore-deletion notification. The last waiting task to receive the deletion notification signals the `deletion_complete` condition variable for the semaphore and awakens the task performing the semaphore deletion. The delete operation then proceeds by first removing the semaphore control block from the list of semaphores and then destroying the underlying POSIX semaphore and freeing the control block memory allocated for the semaphore.

Finally, the delete operation 'unlocks the scheduler' and returns to the calling task.

Special Features of Mutex Semaphores

Mutex Semaphore Creation

semMCreate() creates a special mutual exclusion (mutex) semaphore. The initial token count can be either 0 (locked) or 1 (available). A flag specified by the caller indicates whether tasks waiting on this semaphore will be awakened in FIFO order or in priority order. Mutex semaphores also allow option flags specifying priority inversion safety and automatic task deletion safety characteristics.

Acquiring a Mutex Semaphore Token

The basic mechanism for acquiring a mutex semaphore token is similar to that described earlier for binary semaphores. However, mutex semaphores exhibit a number of special characteristics which are not shared by other semaphore types.

- Mutex semaphores are inherently 'locked' by only one task at a time.
- The task which currently owns the lock on a mutex semaphore may lock that semaphore recursively without blocking or returning an error.
- If the auto-deletion-safety option was specified during mutex semaphore creation, the task enters a deletion-safe condition when it first acquires the semaphore token and 'locks' the semaphore.
- If the priority inversion safety option was specified during mutex semaphore creation, and another task already has the mutex semaphore locked when `semGet()` is called, the priority inversion protection logic is executed. If the task which owns the semaphore lock has a lower priority than that of the task calling `semGet()`, its priority will be temporarily raised to equal that of the task calling `semGet()`. This insures that the higher-priority task isn't delayed unnecessarily because of the current mutex owner's lower priority.

Releasing a Mutex Semaphore Token

Releasing a mutex semaphore token and 'unlocking' the semaphore occurs in a similar fashion to the same operation on a binary semaphore. However, mutex semaphores exhibit a number of features lacking in other semaphore types.

- Because mutex semaphores support the concept of 'lock ownership', an error is returned if a task attempts to release a mutex semaphore lock which it does not currently own.
- If a task has recursively locked a semaphore, the recursion levels were tracked, and ownership of the semaphore lock will not be relinquished until the number of `semGive()` calls equals the number of `semTake()` calls.
- If the auto-deletion-safety option was specified during mutex semaphore creation, the task again becomes deletable after lock ownership of the semaphore is relinquished.
- If the priority inversion safety option was specified during mutex semaphore creation, and the priority of the task had been temporarily raised, its priority is restored to its previous level after lock ownership of the semaphore is relinquished.
- The `semFlush()` call is illegal for mutex semaphores, and immediately returns an error if called.

Queue Management Calls

Queues provide perhaps the most essential element of inter-task communication in VxWorks. They are a combination of mutual exclusion and synchronization primitives, and are fully supported in the Linux VxWorks virtual machine.

Queues always have a fixed number of dedicated message buffers, but the size of those buffers is specified by the caller at queue creation time, and the messages contained in those buffers may be of any size up to the maximum size specified during queue creation.

Queue Creation

Queue creation in the virtual machine is done using the `msgQCreate()` call. It begins by using a thread-safe `malloc()` call to allocate space for a new queue control block. Next, a 'data extent' size is computed based upon the size of each message buffer in the queue and the number of messages specified for the queue. A data buffer called an 'extent' is then allocated via thread-safe `malloc()` to contain the queue's message buffers. When the size of the extent is calculated, room is added to contain a single extra message in excess of the specified number of messages. This is used to support 'zero-length' queues and to enhance the chances for successfully sending an 'urgent' message to the queue without encountering a 'queue full' condition. A queuing order option indicates whether tasks waiting on this queue will be awakened in FIFO order or in priority order.

Receiving Queued Messages

Receiving a message from a VxWorks message queue is done by calling `msgQReceive()`. In the Linux VxWorks virtual machine, the basic mechanism for receiving queued messages is the same as that described earlier for acquiring a semaphore token. The same condition variable, POSIX mutex, and predicate query loop mechanism are used. The waiting and notification mechanisms are essentially like that for a counting semaphore, except that messages are exchanged rather than simple tokens. The task joins a list of tasks waiting on a message from the queue and then checks for available messages, blocking as needed until one or more messages are available.

As long as messages remain in the queue and the timeout (if any) has not expired, a nested inner query loop checks to see if the current task is selected to receive the next message in the queue. If the next message is not for the current task, the inner loop temporarily unlocks the mutex and suspends the current task for a short delay. This allows other receiving tasks of equal or lesser priority an opportunity to obtain their messages. After the short delay expires, the current task re-locks the mutex and returns to the top of the inner query loop. Upon timeout expiration or successful acquisition of a message, the inner query loop exits and the receiving task removes itself from the waiting list for the queue.

Upon successful acquisition of a message the task then fetches and deletes the message from the queue. Deleting the message from the queue is done by incrementing the `queue_head` pointer to the next message in the queue's data extent. The incremented value of the `queue_head` is tested to see if it is within the boundaries of the extent. If the increment operation placed the `queue_head` pointer outside the bounds of the extent, it is adjusted to point to the beginning of the extent. Because the message acquisition made space available in the message queue, the list

of tasks waiting for message space is checked. If any tasks are waiting to send a message to the queue, the condition variable associated with waiting for queue space is signaled. This awakens the waiting task(s) and a selected task obtains the newly freed message slot to send its message to the queue.

One special case may occur while a task is waiting to receive a message from a queue; the queue may be deleted. In this case, all tasks currently waiting on the message queue will be re-awakened in priority order, and an error status will be returned to each task indicating the queue was deleted. In a queue deletion a second `deletion_complete` mutex and condition variable are used to allow the last waiting task awakened by the delete operation to inform the deleting task that no more tasks are waiting on the queue.

Sending Queued Messages

Sending a message to a queue is done by the `msgQSend()` call. In the Linux VxWorks virtual machine, `msgQSend()` begins by locking the mutex for the queue to prevent other tasks from accessing it while it is being modified. Next the queue is checked to determine if it has room to accept the new message. If no waiting was specified in the `msgQSend()` call, a queue-full condition results in an immediate error return. If a waiting period was specified in the `msgQSend()` call, the sending task joins a list of tasks waiting to send messages to the queue. Available space in the queue is released to tasks waiting to send messages in exactly the same way as available messages are released to tasks waiting to receive messages. If the waiting interval specified in the `msgQSend()` call was not an indefinite interval, and the interval expires before space becomes available for the calling task's message, an error return signals the queue-full condition.

If room is available in the queue for the new message, it must be decided where the message is to be placed into the queue. An option argument in the `msgQSend()` call specifies either a normal or urgent priority for the message being sent. For normal priority messages, the `queue_tail` pointer is incremented to point to the next available message buffer. If the increment operation placed the `queue_tail` pointer outside of the bounds of the queue's data extent, then the `queue_tail` pointer is adjusted to point at the first message in the extent. The message is then copied into the buffer addressed by the new `queue_tail` pointer.

For urgent priority messages, the `queue_head` pointer is decremented and then checked to see if the decremented location falls out side of the queue's data extent. If the decremented `queue_head` pointer falls outside of the extent, it is adjusted as needed to point to the last message in the extent. By decrementing the `queue_head` and copying the urgent message into the buffer at the new `queue_head`, it is assured that the new message will be fetched from the queue before any previously queued messages.

After the message has been copied into the queue, the condition variable for the queue is signaled with a `pthread_cond_broadcast()` call and the mutex for the queue is unlocked.

Because a VxWorks task can wait for space in a message queue when performing a `msgQSend()` call, it is possible that the message queue will be deleted while the task is waiting to send its message. This is handled identically to the scenario in which a task is waiting to receive a message when the queue is deleted. All waiting

tasks are awakened and immediately return with an error code indicating message queue deletion occurred.

Queue Deletion

Queue deletion is accomplished in the virtual machine via the `msgQDelete()` call. It can be fairly complex due to the requirement to awaken any and all tasks waiting on the queue when the deletion call is made. It begins by locking the mutex for the queue to prevent other tasks from accessing it during the delete operation. Next it checks the waiting lists for the queue to determine if any tasks are currently waiting on the queue. Tasks may either be waiting to send new messages to a full queue or to receive messages from an empty queue. If any tasks are waiting on the queue, a special flag is set in the queue control block to indicate the queue is being deleted. The task deleting the queue then 'locks the scheduler' and signals the condition variable(s) to alert the waiting tasks. Having done this, it blocks with a `pthread_cond_wait()` on a `deletion_complete` condition variable for the queue and allows the waiting tasks to receive their dummy messages and the queue-deletion notification. The last waiting task to receive the deletion indicator signals the `deletion_complete` condition variable for the queue and awakens the task performing the queue deletion. The deletion operation then proceeds by first removing the queue control block from the list of queues and then freeing all data extent and control block memory allocated for the queue. Finally the scheduler is 'unlocked' and the `msgQDelete()` call returns.

Watchdog Timer Management Calls

The Linux VxWorks virtual machine provides a suite of timer-related functions which allow caller-specified functions to be executed after a caller-specified delay interval. In the VxWorks native environment, these ‘watchdog timers’ operate from the context of the system tick timer interrupt service routine. In the virtual machine environment, they are executed from the context of the system exception task, which operates at pthreads priority 98. At each system timer ‘tick’ the exception task will service the watchdog timer list. Each timer with a tick delay count greater than zero will be decremented by one tick. When the tick delay count is decremented from one to zero, the timeout function specified when the timer was started will be called. The timer will subsequently be idle or inactive until another `wdStart()` call reactivates it.

Watchdog Timer Creation

The `wdCreate()` function creates a new watchdog timer control block and links it into the list of watchdog timers. The watchdog timer is initially inactive when created, and must be activated by using the `wdStart()` call. Once created, watchdog timers may be reused as desired until they are deleted.

Watchdog Timer Startup

The `wdStart()` function specifies a delay interval in terms of VxWorks system ‘ticks’, and a function to execute when the delay interval expires. This call will result in a single execution of the specified timeout function at the end of the specified delay interval. Note that, at least on the Linux ‘x86’ architecture, the minimum delay interval seems to be two clock ticks (20 milliseconds) rather than one tick (ten milliseconds).

Watchdog Timer Cancellation

The `wdCancel()` function immediately zeros the tick delay count for the specified timer, deactivating it without executing the associated timeout function. The watchdog timer, although inactive, remains available for subsequent reuse.

Watchdog Timer Deletion

The `wdDelete()` function removes a watchdog timer from the list of watchdog timers and releases the memory which was allocated for the timer control block. This makes the watchdog timer unavailable for further use.

Appendix A: System Calls Not Addressed and Other Caveats

System Calls Not Addressed

This paper deals only with the issues involved in porting VxWorks native kernel functions to Linux. The issues involved in porting filesystem and networking component features were left unresolved, although these may be addressed in a later paper. Likewise, the issues involved with multiple-processor architectures were not addressed here – although it is worth noting that Linux (like VxWorks) does provide multi-processor support.

Finally, no attempt was made to address target-level debuggers or board support packages. Linux has perhaps the largest number of supported peripheral devices of any Unix-like operating system, and source code, advice and assistance are readily available for the developer of new device drivers.

Anyone who has spent time poring over the volumes of information in the VxWorks user and reference manuals will understand why we choose not to list individual unsupported VxWorks calls in this paper; there are over a thousand calls which would have to be individually addressed. Instead, the unsupported libraries are listed below. Please note that in many cases, Linux does support features comparable to those implemented in the unsupported VxWorks libraries; Linux simply employs a different means of providing those features.

Device Support Libraries

VxWorks provides a series of libraries intended to provide support for specific peripheral hardware devices. In the VxWorks environment, these libraries provide 'driver level' support for those devices. They are explicitly included into the system build using configuration header files and linker command files. In the Linux environment, system configuration scripts used in building the kernel and/or dynamically loadable kernel modules result in the inclusion of equivalent driver level support for peripheral hardware. Thus, by and large, the devices are supported, while the VxWorks libraries and function calls are not. These device support library names are listed below:

- `ataDrv`, `ideDrv`
- `mb87030Lib`, `ncr*Lib*`, `wd33c93Lib*`
- `cd2400Sio`, `evbNs16550Sio`, `i8250Sio`, `m68*Sio`, `mb86940Sio`, `ns16550Sio`, `ppc*Sio`, `z8530Sio`

- `if_*` (network interface driver libraries)
- `ioMmuMicroSparcLib`
- `lptDrv`
- `mmu*Lib`
- `nec765Fd`
- `pcic`, `tcic`

Operating System Support Libraries

VxWorks provides a series of libraries intended to provide support for ‘transparent’ features of the VxWorks operating system. For the most part, Linux provides comparable features, but once again, these are configured into the system using a kernel configuration script. While most of the features are supported, the specific VxWorks libraries are not. In some cases, the VxWorks libraries implement ‘standard’ UNIX or POSIX features and calls which are directly equivalent to Linux features and calls. In other cases VxWorks employs non-standard calls to implement the features, and these calls are unsupported under Linux. The VxWorks system feature support libraries are listed below:

- `aioSysDrv` (Linux supports POSIX1.b asynchronous I/O)
- `bALib`, `bLib`
- `bootConfig`, `bootInit`, `bootLib`, `rebootLib`
- `cache*` - cache management libraries
- `cisLib`, `pccardLib`, `pcmciaLib`, `sramDrv` (Linux supports PCMCIA, card services, and numerous PCMCIA device types)
- `clockLib`
- `cplusLib` (standard C++ runtime library support is available with Linux)
- `dlpiLib`, `strmLib`, `strmSockLib` (Linux does not yet officially support STREAMS)
- `envLib` (Linux supports environment variables)
- `errnoLib` (Linux supports process and thread-specific errno variables)
- `exc*Lib` (Linux provides its own exception handling via signals)
- `fioLib`, `floatLib` (Linux supports ANSI C formatted I/O)
- `fppArchLib`, `fppLib`, `mathHardLib`, `mathSoftLib` (Linux supports hardware & emulation-based floating point math)
- `inflateLib` (Linux supports all widely-used compression and decompression methods)
- `int*Lib`
- `ioLib`, `iosLib`
- `kernelLib`
- `ledLib` (Linux supports the ‘ed’ line editor and command-line editing shells)
- `loadLib`, `unldLib` (Linux uses a dynamic linking loader called ld)
- `loginLib` (Linux supports a login process)
- `logLib` (Linux supports organized system and application logging through syslogd)

- `lstLib`
- `mathALib` (Linux supports all ANSI C math functions)
- `memDrv`
- `mem*Lib` (Linux provides standard POSIX/UNIX memory management)
- `mountLib` (Linux provides standard POSIX/UNIX file system mounting)
- `mqPxLib` (Linux does not yet officially support POSIX message queues)
- `msgQSmLib`, `semSmLib`, `sm*Lib` (the virtual machine does not support multi-processor architectures at this time)
- `pipeDrv` (Linux provides support for pipes and named pipes or FIFOs)
- `ptyDrv` (Linux supports pseudo TTY devices)
- `ramDrv` (Linux provides ram disk support)
- `rngLib`
- `scsi*Lib` (Linux provides disk, tape, and generic SCSI support over all standard variants of the SCSI bus)
- `selectLib` (Linux supports the 4.4BSD select() call)
- `semOLib` (The virtual machine currently does not support VxWorks 4.x semaphores)
- `shellLib` (Linux supports numerous standard UNIX shells, but does not support VxWorks-specific shell commands)
- `sigLib` (Linux supports standard POSIX signal behavior)
- `sysLib`
- `taskArchLib`, `taskHookLib`, `taskVarLib`
- `tickLib`
- `timerLib` (Linux does not yet officially support POSIX timers)
- `timexLib` (Linux provides execution profiling via other means)
- `ttyDrv`, `tyLib` (Linux provides standard POSIX TTY device support)
- `usrConfig` (The Linux VxWorks virtual machine uses `user_sysinit()` and `user_syskill()` instead)
- `usrLib` (Linux provides support for standard UNIX shells, but not for VxWorks-specific commands)
- `vm*Lib` (Linux provides virtual memory support)
- `vxLib`
- `vxw*Lib` (The Linux VxWorks virtual machine presently does not support a C++ calling interface)
- `zbuf*Lib`

Filesystem Support Libraries

VxWorks provides a series of libraries intended to provide support for specific standard file systems using a consistent applications interface. While the set of file systems supported by Linux differs somewhat from that supported by VxWorks, Linux offers a ‘virtual file system’ interface which supports numerous underlying file system formats in a manner largely transparent to the applications programming level.

Once again, file system support in Linux is specified using a kernel configuration script. A list of file system related VxWorks libraries follows:

- `dosFsLib` – MS/DOS file system support
- `rawFsLib` – raw block device file system support
- `rt11FsLib` – DEC RT-11 file system support
- `tapeFsLib` – Tape file system support

Networking Support Libraries

VxWorks provides a series of libraries intended to provide support for specific networking protocols and applications. Linux, for the most part, supports an even more complete suite of networking protocols and applications. However, once again, the features are supported but the libraries are not. Like the operating system features, some VxWorks networking calls follow POSIX and UNIX standards and are directly supported, while other non-standard calls are not. The list of VxWorks network-related libraries follows:

- `arpLib` (Linux supports ARP)
- `bootpLib` (Linux supports BOOTP)
- `etherLib` (Linux supports ethernet)
- `ftpdLib`, `ftpLib` (Linux supports FTP)
- `hostLib` (Linux supports hostname resolution)
- `inetLib` (Linux supports standard POSIX internet address manipulation)
- `m2*Lib`, `snmp*Lib` (Linux does support SNMP, and various MIB libraries are available)
- `netDrv`, `nfsDrv`, `nfsdLib`, `nsfLib` (Linux supports NFS clients and servers)
- `pingLib` (Linux supports PING clients and protocol)
- `ppp*Lib` (Linux supports PPP clients and servers)
- `proxy*Lib` (Linux supports proxy ARP)
- `remLib` (Linux supports 4.2BSD remote command functions)
- `rlogLib` (Linux supports 4.2BSD remote login)
- `routeLib` (Linux supports numerous network routing methods and protocols)
- `rpcLib` (Linux supports the SUN remote procedure call specification)
- `sockLib` (Linux supports POSIX standard sockets)
- `telnetLib` (Linux supports telnet clients and servers)
- `tftpdLib`, `tftpLib` (Linux supports TFTP clients and servers)

Debugging Support Libraries

VxWorks provides a series of libraries intended to provide support for VxWorks debugging and development tools. While Linux offers its own suite of debugging and development tools, the VxWorks-specific tools, libraries, and function calls are not supported. A list of debugging support libraries follows:

- `*Show` – object examination display functions
- `autopushLib`
- `connLib`
- `dbg*Lib`
- `evtBufferLib`
- `moduleLib`
- `passFsLib`
- `spyLib`
- `straceLib`, `strerrLib`
- `symLib`, `symSyncLib`
- `taskInfo`
- `unixDrv`
- `wvHostLib`, `wvLib`, `wvTmrLib`

C Runtime Library Call Issues

Linux provides equivalent calls for all ANSI C and POSIX.1 calls, as well as many SVR4 and BSD UNIX calls. Therefore, few if any C runtime library calls found in VxWorks are likely to be unsupported in Linux. However, some issues with thread-safe or reentrant behavior may exist. That is, some Linux/GNU versions of these library routines may prove not to be thread-safe, while their VxWorks equivalents may be inherently thread-safe. The GNU C runtime libraries may be compiled to provide reentrant operation. However at this time the author does not know:

- precisely which library routines this guarantees to be reentrant., or
- whether the shared C runtime libraries distributed with any current Linux distribution were compiled to be reentrant.

Nonetheless, the `makefile` used to build the VxWorks virtual machine components defines the `REENTRANT` flag. This is intended to ensure that reentrant versions of C runtime library routines are employed if available.

Caveats when Porting VxWorks Applications to Linux

Linux and most other Unix-type operating systems were never designed to support the requirements of ‘hard real-time’ systems, where deterministic and predictable response times for a real-world event are required in order to prevent catastrophic failures in the controlled hardware. Determinism was not a significant consideration in the design of the Linux kernel, nor was it a critical item in the current VxWorks virtual machine environment. However, if the intent is to use Linux as the operating system for a real-time target, the lack of determinism may become troublesome. Interrupt latency, critical paths through the kernel, non-preemptable system calls, and the behavior of various drivers and processes in the system may all contribute to significant ‘jitter’ in the timing of various real-time operations.

For the initial porting effort, the standard Linux kernel is adequate to support the VxWorks virtual machine, and there may initially be no concern over timing performance. If timing performance does become an issue at some point in the porting process, there are several avenues for performance enhancement:

- reduced-footprint kernel distributions which run only those processes essential to proper target operation,
- enhancements to the standard Linux kernel offered by MontaVista Software, Inc. which improve the real-time scheduling characteristics of the standard Linux scheduler, and
- redesign of the control system firmware to rely only upon the ‘native’ features of POSIX threads as implemented in Linux.

Redesign of the control system firmware to a native pthreads design will likely be required in order to maximize performance. VxWorks and other kernels specifically designed for hard real-time support offer numerous features for inter-process communication and synchronization which are not available in a POSIX environment. Support of these features requires an ‘adapter layer’ of code which adds functionality at the inevitable expense of raw performance.

Some VxWorks features, such as user hook routines at task-switch time, cannot be readily implemented in a Linux environment without resorting to kernel modification.

An additional caveat regarding the Linux environment regards the possibility of encountering library functions which are not thread-safe or reentrant. Many of the library functions used for memory management, string manipulation, mathematics, and network management are not inherently reentrant. To make these functions thread-safe they must be guarded by wrapper functions which employ mutexes in order to guarantee that the underlying library functions are accessed by only one thread at a time.