QNX SOFTWARE SYSTEMS

Porting Legacy Systems from Wind River's VxWorks® to the QNX® Neutrino® RTOS

Shiv Nagarajan, Field Consulting Engineer
Robert Craig, Kernel Developer
QNX Software Systems
Email contact: paull@qnx.com

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

VxWorks from Wind River Systems has been a very successful RTOS for the last decade or so. The relative simplicity of the VxWorks operating system – compared to the complexities of desktop/server operating systems – made it a good choice for embedded systems development. But with the increasing complexity of today's embedded systems software, coupled with the high expectations for their reliability and serviceability, developers need a more advanced and flexible operating system, one that offers a full range of capabilities and features designed to help create the embedded systems of the future.

The QNX® Neutrino® RTOS's third-generation architecture and leading-edge technologies offer developers the path to the future for platform software. Embedded systems developers can now build their products with the same high level of capabilities offered by advanced operating systems yet remain sensitive to the unique demands of the embedded systems environment. QNX Neutrino runs on ARM, MIPS, PowerPC, SH-4, and x86 platforms.

Traditionally, porting an application from a conventional RTOS like VxWorks to an advanced operating system like QNX Neutrino isn't easy. There are various issues to consider and choices to make that can dramatically affect the porting process. Realizing this situation, QNX has developed a comprehensive migration kit consisting of this document and a porting library to help customers port their VxWorks-based applications to QNX Neutrino in a more controlled fashion.

This document highlights the areas of impact and the decisions that you need to make in establishing a porting methodology. If the porting process is the journey, then this document is its roadmap.

Porting complex legacy systems from VxWorks to QNX Neutrino is a complex activity that an automated tool alone cannot do, although some of the porting can be automated. Understanding the differences between the two operating systems is essential in order to know what needs to be ported and how. While this document covers all the important topics related to the process of porting, it cannot completely describe this complex topic on its own. You may need to read other QNX documentation where necessary to gain deeper insight on a specific topic.

This paper touches on the various phases of application development/porting and draws comparisons between the two operating systems. It discusses porting strategies and presents the major similarities and differences between the two systems.

We use a "layered" approach to present porting requirements for each phase of the application port, starting from initializing and bringing up the target hardware and concluding with the high-level applications that are more or less hardware independent. In addition to outlining the porting requirements, this paper also compares the QNX Neutrino API with the VxWorks API, showing that the QNX Neutrino API is very much a superset of the VxWorks API.

Finally this document introduces the `vx2qnx` porting library that is designed to help developers port their VxWorks applications to QNX Neutrino with minimal amount of change to the original application code. Although in most cases, applications will be complex enough to necessitate modifications to the source code to port it completely over to QNX Neutrino, this porting library implements key VxWorks functions that enable developers to port their application in phases. And as this library is offered in source code form, it is also an excellent reference to demonstrate how customers can extend this porting library to cover additional functions that may be required.

Note**: In all cases within this document, "VxWorks" refers to versions 5.x in Wind River's portfolio,* ***not *to their AE product.*

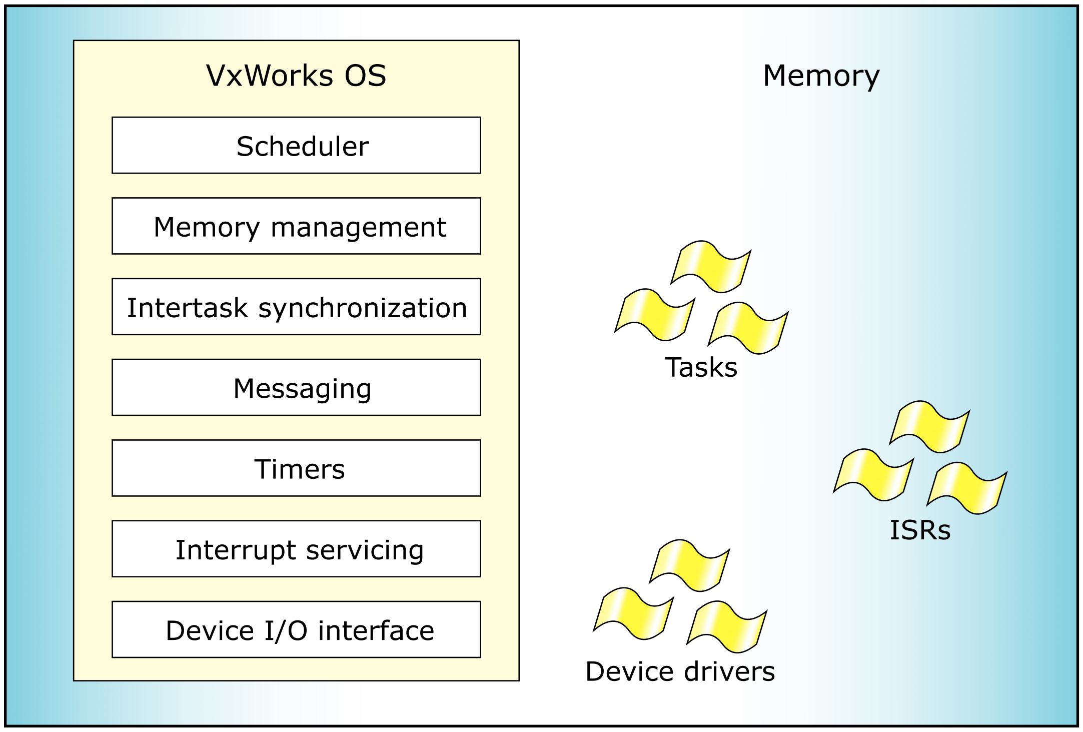
1.1 Architectural overview

This section discusses the major architectural differences between the two operating systems and introduces the major issues that have to be addressed when moving an application to QNX Neutrino. A more in-depth analysis is provided in Section 2.

1.1.1 Architectural comparison

1.1.1.1 VxWorks

Figure 1: The VxWorks architecture.



An architectural overview diagram is shown in Figure 1. The VxWorks kernel itself is small, offering a scheduler, memory management, intertask synchronization, messaging, timers, interrupt servicing, and device I/O interface. The smallest (and only) schedulable unit in the operating system is the *task*.

The OS supervises the execution of tasks through the use of messaging and synchronization techniques. The kernel is preemptable, and uses a priority-based scheduler to determine when tasks will run. The OS and all tasks reside in the same, common address space; all tasks run in supervisor mode, allowing them full access to all processor instructions and physical memory (including kernel memory).

While the OS doesn't require a memory management unit on the processor, the MMU may be used (if it exists) to implement the caching of instructions and data, if desired. The MMU is set up in such a manner that tasks use physical memory addresses when accessing memory. That is, there's no memory address translation required to obtain a physical address from a memory address used within a task. It's important to note that, while the MMU may be enabled with VxWorks, it is *not* used to offer memory-protection of any sort.

Communication into the OS kernel from a task is carried out via a simple function call interface that may or may not include rigorous error-checking for parameters. Although fast, this mechanism can result in the corruption of internal kernel parameters if care isn't taken to ensure parameter correctness.

Interrupt Service Routines (ISRs) and device drivers (provided either by Wind River or the user) are tied into the OS through the appropriate API calls. The device-driver API is implemented such that the standard I/O calls (open, read, write, etc.) can be used to access the devices. Again, device drivers also have full access to the available memory space.

1.1.1.2 QNX Neutrino architecture

QNX Neutrino is a *microkernel*-based OS structured as a tiny kernel that provides the minimal services used by a team of cooperating processes, which in turn provide the higher-level OS functionality. QNX Neutrino is also a process-based, multithreaded operating system that implements OS services such as memory-protected user processes that communicate with each other and with applications via message-passing primitives and synchronization primitives managed by the QNX Neutrino microkernel.

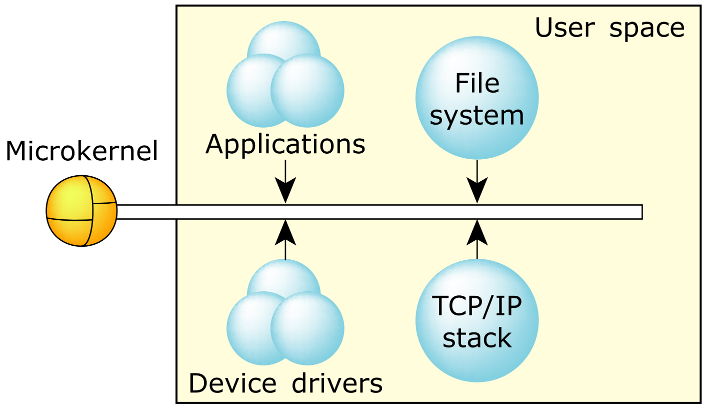
This architecture is fundamentally different from that of realtime executives and of traditional, monolithic Unix-style operating systems. The QNX Neutrino microkernel differs from a standard realtime executive in how the IPC services are used to extend the functionality of the kernel with additional, service-providing processes. Since the OS is implemented as a team of cooperating processes managed by the microkernel, user-written processes can serve both as applications and as processes that extend the underlying OS functionality for user-specific applications. Thus, the OS itself becomes "open" and easily extensible.

Moreover, user-written extensions to the OS won't affect the fundamental reliability of the core OS. QNX Neutrino fully utilizes the processor's MMU to deliver the complete POSIX process model in a protected environment. The microkernel offers complete memory protection, not only for user applications, but also for OS components (device drivers, filesystems, etc.).

The key advantage gained by adding memory protection to embedded applications, especially for mission-critical systems, is improved robustness. With memory protection, if one of the processes executing in a multitasking environment attempts to access memory that hasn't been explicitly declared or allocated for the type of access attempted, the MMU hardware can notify the OS, which can then abort the thread (at the failing/offending instruction). This "protects" process address spaces from each other, preventing coding errors in a thread on one process from "damaging" memory used by threads in other processes or even in the OS. This protection is useful both for development and for the installed runtime system, because it makes postmortem analysis possible.

During development, common coding errors (e.g. stray pointers and indexing beyond array bounds) can cause one process/thread to accidentally overwrite the data space of another process. If the overwritten memory isn't referenced again until much later, debugging can prove quite difficult. With an MMU enabled, the OS can abort the process the instant the memory-access violation occurs, providing immediate feedback to the programmer instead of mysteriously crashing the system some time later. The OS can then provide the location of the errant instruction in the failed process, or position a symbolic debugger directly to this instruction.

Figure 2: The QNX Neutrino architecture gives full memory protection.



In monolithic architectures, the "kernel" comprises so many functions that it is equivalent to the entire operating system, whereas QNX Neutrino is truly a microkernel. It is not only small, but also dedicated to only a few fundamental services for handling threads, signals, message passing, synchronization, scheduling, timers, and processes. Unlike threads, QNX Neutrino itself is never scheduled for execution. The processor executes code in the kernel only as the result of a thread's making an explicit kernel call or in response to a hardware interrupt. The entire OS is built upon these calls. QNX Neutrino is fully preemptable even while passing messages between processes; it resumes the message-pass where it left off before preemption.

The minimal complexity of the QNX Neutrino microkernel helps place an upper bound on the longest nonpreemptable code path through the kernel, while the small code size makes addressing complex multiprocessor issues a tractable problem. Services were chosen for inclusion in the microkernel on the basis of having a short execution path. Operations requiring significant work (e.g. process loading) were assigned to external processes/threads, where the effort to enter the context of that thread would be insignificant compared to the work done within the thread to service the request.

System processes are essentially indistinguishable from any user-written program – they use the same public API and kernel services available to any (suitably privileged) user process. Since most operating system services are provided by standard system processes, it's very simple to augment the QNX Neutrino OS itself: just write new programs to provide new OS services.

1.1.1.3 OS comparison

The basic runtime unit in VxWorks is a *task*. The basic runtime unit in QNX Neutrino is a *thread*. Threads in QNX Neutrino are grouped into containers called "*processes*" and are controlled and scheduled via the operating system. A task in VxWorks and a thread in QNX Neutrino can be considered equivalent. Task control is implemented using *taskLib* functions for VxWorks and via the *pthread_* family of functions for QNX Neutrino.

In VxWorks, there is only one "*task container*" and that is the operating system itself. In QNX Neutrino, multiple processes containing one or more threads may execute. Each process has its own private, protected memory space that isn't accessible by other processes. In this context, a full VxWorks system can be equated to a single process with many threads operating in QNX Neutrino. In VxWorks, every task has full access to the entire physical memory space provided by the processor. This means that any task can access (and potentially corrupt) memory space belonging to another task or to the OS. Processes may share memory regions, but the processes must specifically indicate which memory areas they wish to share using the *shm_** functions.

There is a very clear delineation between the operating system and the application in QNX Neutrino. Calls into the OS are implemented via a system call interface that completely isolates user applications from the OS. This is in sharp contrast to the function-based interface into the OS provided by VxWorks in which corruption of internal OS structures may occur due to incorrect parameters or invalid memory pointers.

This separation of application and operating system is also evident in building application loads for a target system. In VxWorks, the OS components must be linked into the application to produce a single final image. In QNX Neutrino, application builds are accomplished completely independently from the kernel and are stored separately in a filesystem for subsequent execution. The QNX Neutrino kernel itself isn't user-modifiable – it has no need to be, given its extensible architecture. In QNX Neutrino, you extend the kernel's functionality by messaging to other cooperating processes.

Memory access in VxWorks is almost always based on a "one-to-one" mapping, whereby physical memory addresses are the same as the application addresses. QNX Neutrino implements *virtual memory*; the MMU translates application addresses into corresponding physical memory addresses while applying the appropriate memory-protection criteria to the memory being accessed. If access to a particular physical memory location is required, the memory space must be "mapped into" the process memory space using the *mmap_** functions calls. If the physical address of an application address needs to be known (e.g. for programming a buffer location into a hardware device), then the address needs to be translated using functions such as *mem_offset()*, *mmap_device_memory()*, *mmap_device_io()*, and *mmap()*.

QNX Neutrino incorporates the concept of "privilege" and "permissions" into threads. This essentially provides a mechanism for restricting the functionality that a thread can use. For example, in the x86 architecture, a thread must have I/O privileges (obtained by using the *ThreadCtl()* function) and **root** permissions (obtained by setting the appropriate permission bits on the application image) before it can access input/output opcodes. Similar privilege levels are provided on other processor families.

The middleware associated with integrating device drivers into each OS is very different given the fundamental architectural differences between the two operating systems. This is true for all types of devices (block, character, network, etc.).

1.1.2 Top porting issues

Given the differences between the two systems, you'll need to address the following key issues:

1) The library APIs for VxWorks and QNX Neutrino are quite different. While both operating systems provide POSIX compliance, there are many library routines that are unique to each OS. Therefore, a "mapping" of each VxWorks routine to the equivalent QNX Neutrino routine must be made.

2) Application addresses must be translated into *physical memory addresses*. Physical memory needs to be explicitly mapped into a process's memory space.

3) In QNX Neutrino, memory isn't automatically shared between threads in different processes. Shared memory must be explicitly enabled between threads in different processes. Initially, a process can access only the memory assigned to it from the heap/operating system.

4) Given that the device-driver infrastructure is so different, reuse of device-driver code may not be feasible; drivers will, for the most part, have to be rewritten. Device drivers must handle the appropriate privilege, permission, and address-translation issues.

1.2 Porting strategies

From the point of view of porting code, there are two possible strategies you could follow:

1) Use the `vx2qnx` porting library that emulates VxWorks APIs on QNX Neutrino to migrate most of the application code from VxWorks to QNX Neutrino with minimal amount of modifications.

2) Replace the native VxWorks functions with appropriate native QNX Neutrino calls. This replacement can be carried out manually or possibly automatically through the use of code-parsing tools. Since there won't be a direct mapping between some VxWorks functions and native QNX Neutrino APIs, some portion of the application code may need to be rearchitected and modified to fit the architecture and services offered by QNX Neutrino.

The goal of the first option is to migrate an application from VxWorks to QNX Neutrino with as little modification as possible. Ideally, this should be as simple as "rebuild and run." But in reality, most applications are complex, and the operating systems are architecturally different enough to make the idea of full emulation almost impossible to implement in practice. Also, there are hundreds of API functions provided in a typical operating system. Emulating *all* API functions correctly is a difficult and error-prone task.

The second option requires a port of the application from VxWorks to QNX Neutrino, which will involve modifying some portion of the code to reflect the new architecture and services offered by QNX Neutrino. This would move you away from coding with VxWorks and allow you to progress towards developing code that uses more of the features available in QNX Neutrino. There may also be a performance benefit in "hand-coding" the changes. The obvious disadvantage is that the translation is a time-consuming and tedious process. Also, in order to fully utilize QNX Neutrino's feature set, you may need to rearchitect the application.

In both cases, an appropriate mapping of native VxWorks to native QNX Neutrino function calls is required. Each function will fit into one of four broad categories:

- no changes necessary (e.g. stdio functions and POSIX functions)
- function syntax change (one-to-one mapping)
- function call must be emulated in QNX Neutrino (several QNX Neutrino functions must be used to emulate the VxWorks function)
- no equivalent functionality exists.

Care also has to be taken with the mapping to ensure that mapped routines either provide the same behavior, or that the behavioral differences aren't significant from the application's point of view.

The `vx2qnx` porting library emulates the commonly used VxWorks functions (e.g. the task library functions, semaphores, message queues, etc.). As has been indicated by the open-source porting libraries and also validated by other third-party offerings, it's a relatively straightforward task to develop an emulation of the basic VxWorks APIs on a POSIX operating system such as QNX Neutrino. This porting library is provided in source code form so developers can easily extend it to increase the API coverage. However, we must also take into account the degree of compatibility that a porting library can provide. In particular, a porting library probably won't be able to deal with code that requires direct access to hardware (since this code will depend on physical and virtual addresses being the same and it may need to service interrupts). For the most part, code that is "close to the hardware" will have to be hand-ported so that it integrates correctly into the QNX Neutrino architecture.

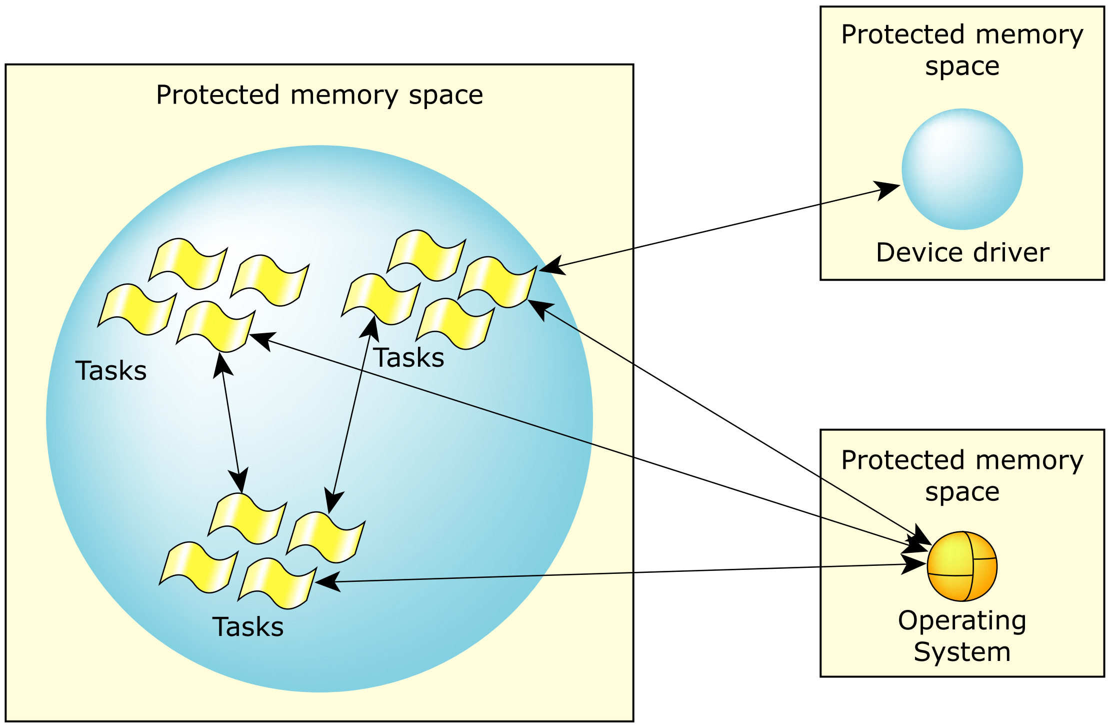
The **VxSim** product from VxWorks identifies a convenient delineation between portions of an application that can be ported in this manner versus portions that require more in-depth work. Ideally, given an application designed to run on both target hardware and on VxSim, you should be able to "rebuild and run" the application on QNX Neutrino using the `vx2qnx` porting layer.

For some applications, a combination of the two strategies could be used. The `vx2qnx` porting library can be used to provide the bulk of the initial port, with some hand-coding to cover areas not supported. Once the initial port is completed, a planned strategy should be introduced to move towards using native QNX Neutrino code as quickly as possible so that the full feature set of QNX Neutrino can be brought into play and optimizations introduced where required.

From an execution point of view, there are also two possible strategies:

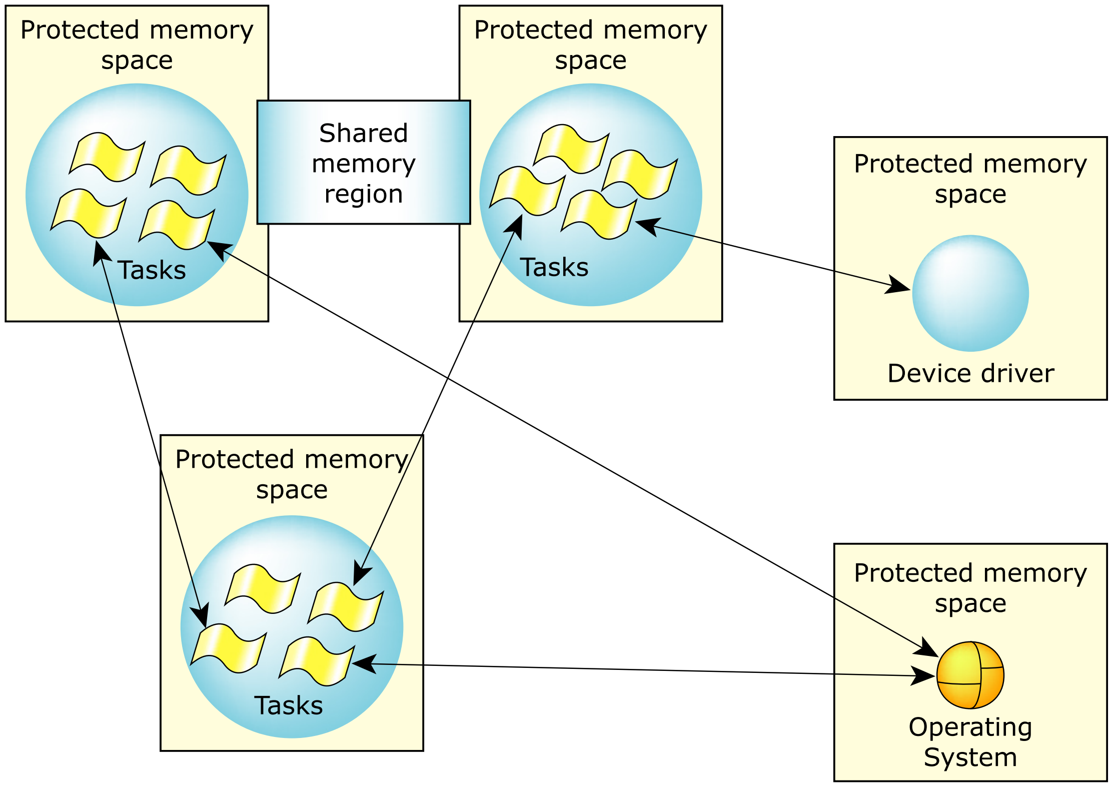
1. Run the application as a single, multithreaded process in QNX Neutrino as in Figure 3.

Figure 3: A single, multithreaded process in QNX Neutrino.



2. Break the application into many multithreaded processes in QNX Neutrino as in Figure 4. Native QNX Neutrino IPC calls or shared-memory regions can be used to provide communications between the threads in the processes.

Figure 4: Many multithreaded processes in QNX Neutrino.



Option one has the advantage of being quicker to implement, since it maps far more closely into the VxWorks runtime model. However, it has a significant disadvantage: none of the protection and isolation mechanisms offered by QNX Neutrino are used to prevent tasks from corrupting each other inadvertently. While not ideal, this mechanism still provides considerably more protection than the VxWorks implementation, because QNX Neutrino and its native device drivers are fully protected.

Option two is more desirable, but significantly more work. Here the full protection and isolation capabilities provided by QNX Neutrino are brought into play. However, it's quite likely that rearchitecting the code will be required in addition to a rewrite in order to support the native command set.

In practice, it would make sense to perform an initial port based on option one, and then migrate gradually towards option two. Loosely coupled VxWorks tasks can be readily "pulled out" of the main application process and into separate processes using either native QNX Neutrino or `vx2qnx` porting-library calls for interprocess communication (IPC). Loosely coupled tasks are those for which a clean IPC interface has been designed (e.g. message queues or a defined shared-memory block protected with semaphores). Strongly coupled tasks would include tasks using shared function callbacks or global variables for IPC/synchronization. The additional modularity and isolation provided by this separation can greatly aid in troubleshooting such problems as "memory tramplers" where one task writes over memory belonging to another. As the modularity of the application increases, or as new features need to be added, you can begin to rearchitect portions of the application to fully take advantage of the native QNX Neutrino function calls.

Care should be taken to plan a migration strategy that eventually results in native calls being used wherever possible, in order to optimize the application and to fully realize the enhanced capabilities of QNX Neutrino.

1.2.1 Recommended porting path

In summary, the following porting path is recommended:

- Use the `vx2qnx` porting library that enables the porting of the application code with minimal code modification. Where necessary, hand-coding parts of the application code to fit the new architecture would be required. This would create a single process that contains all of the VxWorks threads. This gets the application up and running in the shortest period of time.

- Identify any areas with performance issues and rewrite those sections of the code using native QNX Neutrino calls.

- Analyze the code base and introduce separate processes (with one or more threads) that communicate using native QNX Neutrino IPC mechanisms for blocks of functionality that are loosely coupled.

- When new features need to be added, implement them using the native QNX Neutrino functions, rather than extending the code in the porting layer.

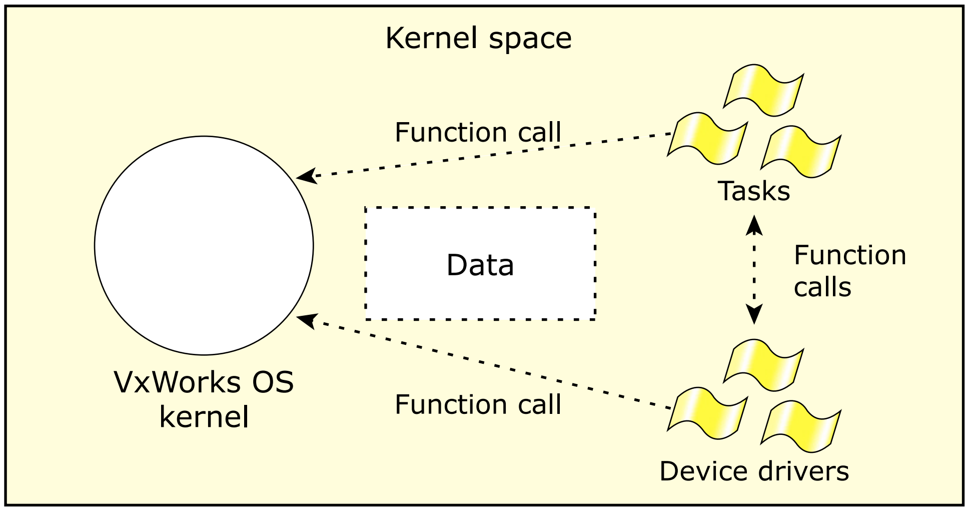
2 DETAILED COMPARISON

In this section we take a closer look at how the two operating systems compare with respect to certain feature sets.

2.1 Kernel accessibility

2.1.1 VxWorks

Figure 5: In VxWorks, everything runs in "kernel space."



The kernel in VxWorks is fully accessible at a number of layers – including inadvertently – by tasks (See Figure 5). Essentially, everything in the application runs in "kernel space." Tasks gain access to the kernel through straight function calls provided by the VxWorks libraries. Communication between the OS and tasks and device drivers is implemented using a straight function-call interface with all data areas accessible by all executable components in the system.

Task management is handled through the VxWorks scheduler. It's possible to modify some of the characteristics of task execution through a series of *taskHook* routines in *taskLib*. These include such functions as *taskSwitchHookAdd()*, *taskCreateHookAdd()*, *taskDeleteHookAdd()*, etc. Tasks can also access OS information related to the task through direct access into the Task Control Block (TCB) structure. This access, while possible, isn't recommended given that the C-structure used for storing the information could change between releases. Corruption of these structures (detected when displaying all task information) is often a result of common user problems (e.g. stack overflows).

Task scheduling in VxWorks is priority-based and fully preemptive with two scheduling modes: First-In-First-Out (FIFO) and Round Robin. Task control is accomplished through the *taskLib* library. Calls are provided for delaying a task from scheduling for a period of time (*taskDelay*), preventing tasks from being preempted (*taskLock*/*intLock*), changing task priorities (*taskPrioritySet*), etc.

2.1.2 QNX Neutrino

The QNX Neutrino microkernel is paired with the Process Manager in a single module (**procnto**). This module is required for all runtime systems. The process manager is capable of creating multiple POSIX processes (each of which may contain multiple POSIX threads).

User processes access microkernel functions directly via kernel calls and Process Manager functions by sending messages to **procnto**. A user process sends a message by invoking the *MsgSendv()* kernel call. It's important to note that threads executing within **procnto** invoke the microkernel in exactly the same way as threads in other processes. The fact that the process manager code and the microkernel share the same process address space doesn't imply a "special" or "private" interface. All threads in the system share the same consistent kernel interface and all perform a privilege switch when invoking the microkernel.

2.2 POSIX support

VxWorks provides POSIX interfaces for several facilities including scheduling, semaphores, message queues, signals, and timers. QNX Neutrino includes all these interfaces and also provides a fully POSIX-conforming suite of synchronization mechanisms including semaphores, mutual-exclusion locks, message queues, condition variables, signals, timers, sleep-on locks, reader-writer locks, etc. All of the task/thread-based primitives conform to POSIX.

In VxWorks, task priorities operate such that lower-numbered priorities are higher in importance than higher-numbered priorities. Both POSIX and QNX Neutrino use the reverse: higher-numbered priorities are *higher* in importance. QNX Neutrino also provides 256 different priority levels (similar to VxWorks) for thread scheduling, so this provides a direct one-to-one mapping between VxWorks and QNX Neutrino priorities.

In terms of scheduling policies, in addition to the standard Round Robin and FIFO-based policies, QNX Neutrino provides access to a sporadic scheduling policy (which is also described in the POSIX interface specification) that allows application threads with sporadic CPU utilization requirements to allocate execution budgets for themselves.

The following table provides a side-by-side comparison of the POSIX-supported functions in both VxWorks and QNX Neutrino. Note that under VxWorks, most of the different functions are provided in separate libraries; in QNX Neutrino, most of the POSIX functionality is implemented by functions in the **libc** library.

Description (POSIX Section)	**VxWorks Lib**	**QNX Neutrino Lib**
Semaphores (POSIX 1003.1)	**semPxLib**	**libc**
Message queues (POSIX 1003.1)	**mqPxLib**	**libc**
Signals (POSIX 1003.1 and Unix)	**sigLib**	**libc**
Timers (POSIX 1003.1)	**timerLib**	**libc**
Scheduling policies	RR, FIFO	RR, FIFO, Sporadic Scheduling
Mutexes (POSIX 1003.1 Threads)		**libc**
Reader-writer locks (Unix)		**libc**
Sleepon locks		**libc**
Barriers (POSIX 1003.1j)		**libc**
Priority levels	256, High=*Low* Pri	256, High=*High* Pri
Task creation (POSIX 1003.1)	**taskLib** (Not POSIX)	**libc**
Condition variables (POSIX 1003.1)		**libc**

QNX Neutrino implements the standard POSIX API, and is compliant with the following:

- POSIX 1003.1 (system interface)
- POSIX 1003.1a (system interface extensions)
- POSIX 1003.1b (realtime extensions)
- POSIX 1003.1c (threads)
- POSIX 1003.1d (additional realtime extensions)
- POSIX 1003.1e
- POSIX 1003.1g

- POSIX 1003.1f (transparent file access)
- POSIX 1003.1j (advanced real-time extensions)
- POSIX 1003.1-200x (draft)
- POSIX 1003.4 (draft)

QNX Neutrino also complies with the POSIX 1003.1-2001 specification (which is based on the 1003.1-1996 spec with its amendments P1003.1a, 1003.1d-1999, 1003.1g-2000, 1003.1j-2000) as well as with the features defined by the current POSIX 1003.13 profiles (PSE51, 52 and 53). QNX Neutrino has been granted a certification of compliance as a PSE52 Realtime Controller 1003.13-2003 System.

2.3 Memory accessibility

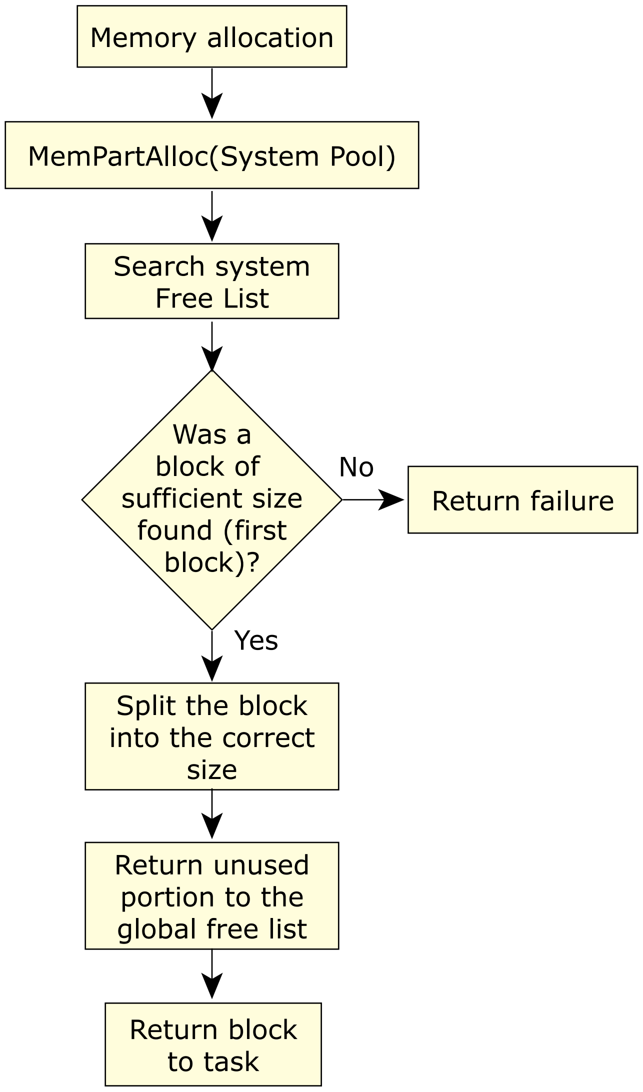
2.3.1 VxWorks

As noted in a previous section, the memory space in VxWorks is fully open to all tasks, and physical addresses are used at the application layer to access memory. The MMU, if used at all in the standard implementation, is turned on only to provide instruction and data-caching capability. Memory may be allocated using the standard library-allocation routines (*malloc()*, *calloc()*, etc.) or by using "pool-specific" access routines that fulfill the request from a specified memory pool (set up during the memory initialization stage of the startup sequence).

For hardware device access, the *cacheDmaMalloc()/cacheDmaFree()* routines are used to provide cache-safe memory for devices that use direct memory access. While translation from "virtual" to "physical" memory may be necessary (using the *cacheDrvVirtToPhys()/cacheDrvPhysToVirt()* functions) depending on the architecture, this is not usually the case.

Blocks of memory are allocated using a "first-fit" allocation strategy and, when freed, are returned to the free list (a singly linked list structure). With this implementation, coalescing of adjacent freed blocks can be carried out only in one direction. Repeated allocate/free cycles can quickly fragment memory, resulting in the inability to fulfill a memory request despite the fact that sufficient memory is physically available. This fragmentation is especially noticeable when C++ libraries such as STL are used, and it can often result in the need to replace the allocator provided with the operating system.

Figure 6: Memory allocation in VxWorks.


Memory allocation
MemPartAlloc(System Pool)
Search system Free List
Was a block of sufficient size found (first block)?
No
Return failure
Yes
Split the block into the correct size
Return unused portion to the global free list
Return block to task

2.3.2 QNX Neutrino

QNX Neutrino offers a fully capable virtual memory implementation using the MMU. A typical MMU operates by dividing physical memory into a number of 4 KB pages. The hardware within the processor then makes use of a set of page tables stored in system memory that define the mapping of virtual addresses (i.e. the memory addresses used within the application program) to the addresses emitted by the CPU to access physical memory. While the thread executes, the page tables managed by the OS control how the memory addresses that the thread is using are "mapped" onto the physical memory attached to the processor.

QNX Neutrino provides a full memory-protection model that relocates all code in the system image into a new virtual space, enabling the MMU hardware and setting up the initial page-table mappings. This allows the process manager to start in a protected, MMU-enabled environment. The process manager will then take over this environment, changing the mapping tables as needed by the processes it starts. In this model, each process is given its own private virtual memory, under control of the CPU's MMU.

Memory allocation by tasks takes place using the POSIX standard *mmap()* function call, on top of which QNX Neutrino has provided a default memory allocator. The memory allocator attempts to minimize fragmentation and optimize performance through a combination of techniques involving pool-allocation as well as the coalescing of free memory blocks as they become available. Since memory is managed by the process manager in 4 KB-sized units, and since the process manager can assemble noncontiguous physical pages to satisfy a memory allocation request to create a contiguous virtual region, physical fragmentation is not typically an issue (the kernel can defragment physical memory if necessary).

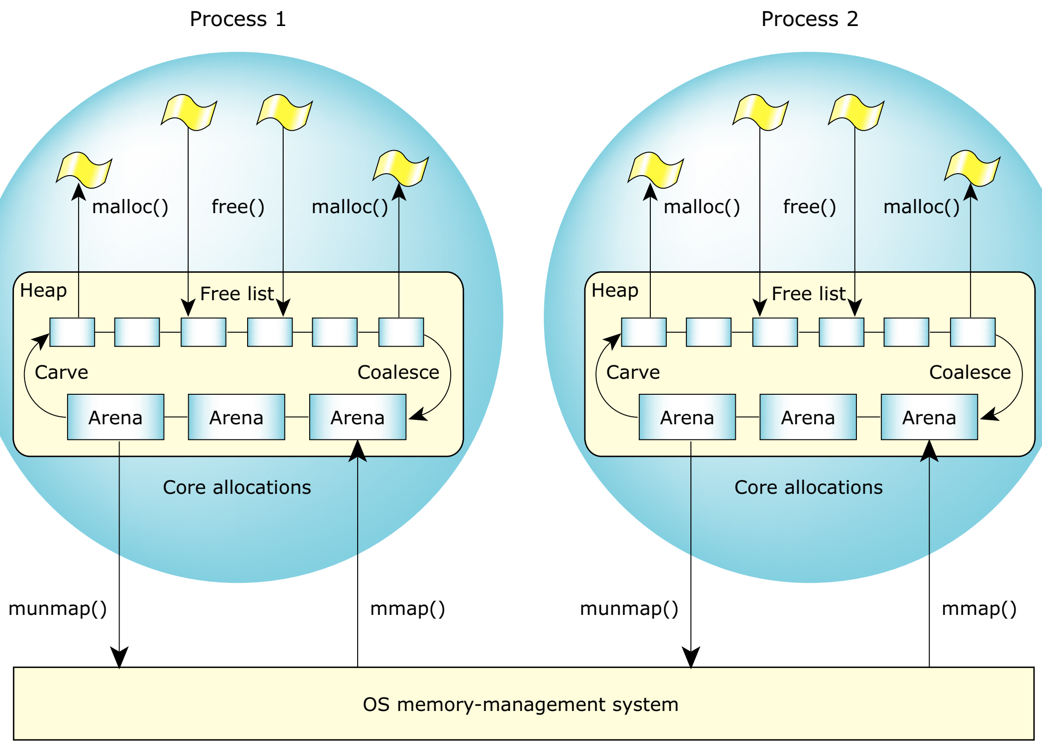
The default QNX Neutrino allocator is based on a first-fit allocation strategy. Each process has its own heap, so a well-designed system will have minimal fragmentation of the heap, since access to the heap is restricted to threads in the same process. The allocator works as follows.

Memory is obtained from the system in integral multiples of "page size" blocks. The blocks (known as arenas) are then carved up to satisfy individual memory allocation requests. When an application makes a request for memory (using a *malloc()*, *calloc()* or *realloc()* call), a free list is searched for an appropriately sized block.

The first such block found is then used to satisfy this request. If the block that is found is significantly larger than the requested size, the block is split into two pieces. The first piece is used to satisfy the allocation request, while the remaining piece is dropped onto the free list. Whenever memory is released to the system, it is placed on the free list. Adjacent blocks are automatically coalesced on the free list. When a complete arena has been coalesced on the free list, it is automatically returned to the system.

***Note**: The free lists in QNX Neutrino are maintained with doubly linked lists. As such, coalescing in both directions can be carried out. This also acts to greatly reduce local fragmentation issues.*

Figure 7: Memory management in QNX Neutrino.



The QNX Neutrino *System Architecture* guide provides additional in-depth information on how the OS manages memory.

2.4 Intertask / interprocess communications

VxWorks provides several different mechanisms for tasks to communicate with each other. The different methods available include:

Semaphores

Semaphores are the primary means of synchronization between tasks in VxWorks. Three different kinds of semaphores are provided: binary, counting, and mutual-exclusion. The binary semaphores are used for simple synchronization between two tasks for a critical region. Counting semaphores permit multiple simultaneous accesses to a critical section that represents a resource that is available in multiplicity. Mutual-exclusion semaphores are special in that they provide solutions to some of the problems inherent in the basic semaphore types, including adding priority-inversion protection, deletion safety, and recursive access to the resource. All VxWorks semaphores also permit timeouts to be associated with obtaining the resource in order to prevent indefinite waiting.

When a semaphore is created, a program can also specify the wakeup mechanism for tasks waiting for a semaphore that is becoming available. Two different wakeup mechanisms are available: *priority-based* wakeup involves waking up the task that is of the highest priority, while *FIFO-based* wakeup wakes the task that has been waiting the longest (regardless of the priorities of the other blocked tasks).VxWorks also offers access to POSIX semaphores.

Message queues and pipes

VxWorks provides message queues as a primary method of intertask communication. Message queues can hold a variable number of messages, each of differing lengths. Application messages are queued onto the message queue using the *msgQSend* function. Receivers block on the queue until a message is available. When a message becomes available, a blocked receiver is woken up by one of two mechanisms: priority or FIFO.

VxWorks native message queues provide the equivalent of only two priority levels: regular and urgent (the message is deposited on the head of the queue when sent). POSIX message queues (*mq_**), which provide 32 priority levels, are also available under VxWorks.

Pipes provide another method of interprocess communication. Pipes offer the ability to do a "select" that allows a task to wait for data to arrive from any one of a set of file descriptors (I/O devices). QNX Neutrino provides an implementation of the pipe interface as described by the POSIX 1003.1 interface specification.

2.4.1 VxWorks message queue example

This example demonstrates how message queues can be used to synchronize and communicate between tasks under VxWorks. Initially, a message queue is created that will be shared between the various tasks. Four tasks are spawned: two sender tasks and two receiver tasks.

```
#include <msgQLib.h>
#include <stdio.h>
#include <sysLib.h>
#include <string.h>
#include <taskLib.h>




#define MAX_NUM_MSG 5
#define MAX_MSG_LEN 100

#define NUM_MSGS 2*MAX_NUM_MSG

MSG_Q_ID    msgQId;

void sendTask1(void)
{
    static char buf[MAX_MSG_LEN];
    int i;
    for (i = 0; i < NUM_MSGS; i++) {

        /* Yield to allow other tasks to run. Otherwise FIFO
           means that this task will run until blocked. */
        taskDelay(0);

        /* Write and send a message. */
        sprintf(buf,"Send1 %d", i);
        if (msgQSend(msgQId,
                    buf,
                    strlen(buf)+1,
                    WAIT_FOREVER,
                    MSG_PRI_NORMAL) == ERROR) {
            printf("Send 1 msgQSend failed!\n");
            return;
        }

        if (i == 3) {
            /* Send an urgent message. */
            sprintf(buf,"Send1 URG %d", i);
            if (msgQSend(msgQId,
                    buf,
                    strlen(buf)+1,
                    WAIT_FOREVER,
                    MSG_PRI_URGENT) == ERROR) {
                printf("Send 1 msgQSend failed!\n");
                return;
            }
        }
    }
}

void sendTask2(void)
{
    static char buf[MAX_MSG_LEN];
    int i;

    taskDelay(sysClkRateGet());
    for (i = 0; i < NUM_MSGS; i++) {
        taskDelay(0);
        sprintf(buf,"Send2 %d", i);
        if (msgQSend(msgQId,
                    buf,
                    strlen(buf)+1,
                    WAIT_FOREVER,
                    MSG_PRI_NORMAL) == ERROR) {
            printf("Send 2 msgQSend failed!\n");
            return;
        }
        if (i == 5) {
            sprintf(buf,"Send2 URG %d", i);
            if (msgQSend(msgQId,
                        buf,
                        strlen(buf)+1,
                        WAIT_FOREVER,
                        MSG_PRI_URGENT) == ERROR) {
```

```
                printf("Send 2 msgQSend failed!\n");
                return;
            }
        }
    }
}


void receiveTask1(void)
{
    char buf [MAX_MSG_LEN];
    int i;

    for (i = 0; i < NUM_MSGS; i++) {
        if (msgQReceive(msgQId,
                        buf,
                        MAX_MSG_LEN,
                        WAIT_FOREVER) == ERROR) {
            printf("Rx task 1 msgQReceive failed.\n");
        } else {
            printf("Task 1 received %s\n", buf);
        }
    }

}



void receiveTask2(void)
{
   char buf [MAX_MSG_LEN];
   int i;

   for (i = 0; i < 10; i++) {
       if (msgQReceive(msgQId,
                        buf,
                        MAX_MSG_LEN,
                        WAIT_FOREVER) == ERROR) {
            printf("Rx task 2 msgQReceive failed.\n");
       } else {
           printf("Task 2 received %s\n", buf);
        }
    }
}




void msgQ_init(void)
{
    int pri;

    taskPriorityGet(0, &pri);
    pri++;        /* Priority is less than init’s priority, so init
                     can run to completion. */

    if ( (msgQId = msgQCreate(MAX_NUM_MSG,
                            MAX_MSG_LEN,
                            MSG_Q_FIFO)) == NULL)
    {
        printf("msgQCreate failed!\n");
        return;
    }

    if ( taskSpawn("task1Tx",
                   pri,
                   0,
                   1000,
                   (FUNCPTR)sendTask1,
                   0,0,0,0,0,0,0,0,0,0) == ERROR)
    {
        printf("taskSpawn Tx1 failed!\n");
    }

    if (taskSpawn("task2Tx",
                   pri,
                   0,
                   1000,
                   (FUNCPTR)sendTask2,
                   0,0,0,0,0,0,0,0,0,0) == ERROR)
    {
        printf("taskSpawn Tx 2 failed!\n");
    }


    if (taskSpawn("task1Rx",
                   pri,
                   0,
                   1000,
                   (FUNCPTR)receiveTask1,
                   0,0,0,0,0,0,0,0,0,0) == ERROR)
    {
        printf("taskSpawn Rx 1 failed!\n");
    }

    if (taskSpawn("task2Rx",
                   pri,
                   0,
                   1000,
                   (FUNCPTR)receiveTask2,
                   0,0,0,0,0,0,0,0,0,0) == ERROR)
    {
        printf("taskSpawn Rx 2 failed!\n");
    }


}
```

2.4.2 QNX Neutrino version (message queue)

To implement the same thing under QNX Neutrino, we have two different options. We can either map the original tasks under VxWorks to threads within the same process, or as different processes themselves. Both of these options are presented here.

Here's the version with threads within one process:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/QNX Neutrino.h>
#include <sys/types.h>
#include <pthread.h>
#include <mqueue.h>
#include <sched.h>

#define MAX_NUM_MSG 5
#define MAX_MSG_LEN 100

#define NUM_MSGS 2*MAX_NUM_MSG

#define MSG_PRI_NORMAL 10
#define MSG_PRI_URGENT 20

#define MQ_NAME "simple_queue"

mqd_t msgQId;

void *sendTask1(void *arg)
{
  static char buf[MAX_MSG_LEN];
  int i;

  for (i = 0; i < NUM_MSGS; i++) {

    /* Yield to allow other tasks to run. Otherwise FIFO means that
    this task will run until blocked. */
    sched_yield();

    /* Write and send a message. */
    sprintf(buf,"Send1 %d", i);
    if (mq_send(msgQId, buf, strlen(buf)+1, MSG_PRI_NORMAL) < 0) {
      printf("Send 1 msgQSend failed!\n");
      return(NULL);
    }

    if (i == 3) {
      /* Send an urgent message. */
      sprintf(buf,"Send1 URG %d", i);
      if (mq_send(msgQId, buf, strlen(buf)+1, MSG_PRI_URGENT) < 0) {
        printf("Send 1 msgQSend failed!\n");
        return(NULL);
      }
    }
  }
  return(NULL);
}

void *sendTask2(void *arg)
{
  static char buf[MAX_MSG_LEN];
  int i;

  for (i = 0; i < NUM_MSGS; i++) {
    sched_yield();
    sprintf(buf,"Send2 %d", i);
    if (mq_send(msgQId, buf, strlen(buf)+1, MSG_PRI_NORMAL) < 0) {
      printf("Send 2 msgQSend failed!\n");
      return(NULL);
    }
    if (i == 5) {
      sprintf(buf,"Send2 URG %d", i);
      if (mq_send(msgQId, buf, strlen(buf)+1, MSG_PRI_URGENT) < 0) {
        printf("Send 2 msgQSend failed!\n");
        return(NULL);
      }
    }
  }
  return(NULL);
}


void *receiveTask1(void *arg)
{
  char buf [MAX_MSG_LEN];
  int i;
  unsigned prio;

  for (i = 0; i < NUM_MSGS; i++) {
    if (mq_receive(msgQId, buf, MAX_MSG_LEN, &prio) < 0) {
      printf("Rx task 1 msgQReceive failed.\n");
    } else {
      printf("Task 1 received %s\n", buf);
    }
  }
  return(NULL);
}



void *receiveTask2(void *arg)
{
  char buf [MAX_MSG_LEN];
  int i;
  unsigned prio;

  for (i = 0; i < 10; i++) {
    if (mq_receive(msgQId, buf, MAX_MSG_LEN, &prio) < 0 ) {
      printf("Rx task 2 msgQReceive failed.\n");
    } else {
      printf("Task 2 received %s\n", buf);
    }
  }
  return(NULL);
}

void msgQ_init(void)
{
  int pri;
  pthread_t tid[4];
  pthread_attr_t attrib;
  struct sched_param param;
  struct sched_param our_param;
  int i;
  struct mq_attr mqattr;

  pthread_attr_init (&attrib);
  pthread_attr_setinheritsched (&attrib, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy (&attrib, SCHED_FIFO);
  sched_getparam(0, &our_param);
  param.sched_priority = our_param.sched_priority-1;
  pthread_attr_setschedparam (&attrib, &param);

  memset(&mqattr, 0, sizeof(mqattr));
  mqattr.mq_maxmsg = MAX_NUM_MSG;
  mqattr.mq_msgsize = MAX_MSG_LEN;
  mq_unlink(MQ_NAME);
  if ((msgQId = mq_open(MQ_NAME, O_RDWR|O_CREAT, S_IRUSR|S_IWUSR,
                 &mqattr)) < 0) {
        printf("msgQCreate failed!\n");
        return;
  }

  if ( pthread_create(&tid[0], &attrib, sendTask1, NULL) < 0)
  {
    printf("taskSpawn Tx1 failed!\n");
  }
  if ( pthread_create(&tid[1], &attrib, sendTask2, NULL) < 0)
  {
    printf("taskSpawn Tx2 failed!\n");
  }
  if ( pthread_create(&tid[2], &attrib, receiveTask1, NULL) < 0)
  {
    printf("taskSpawn Rx1 failed!\n");
  }
  if ( pthread_create(&tid[3], &attrib, receiveTask2, NULL) < 0)
  {
    printf("taskSpawn Rx2 failed!\n");
  }
  for (i=0; i < 3; i++)
    pthread_join(tid[0], NULL);
  mq_unlink(MQ_NAME);
  return;
}

int main(int argc, char *argv[])
{
  msgQ_init();
  return;
}
```

2.4.3 QNX Neutrino version using separate processes

Here's a version of the same code, but this time with separate processes:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/QNX Neutrino.h>
#include <sys/types.h>
#include <pthread.h>
#include <mqueue.h>
#include <sched.h>
#include <unistd.h>
#include <spawn.h>
#include <fcntl.h>
#include <sys/wait.h>

#define MAX_NUM_MSG 5
#define MAX_MSG_LEN 100

#define NUM_MSGS 2*MAX_NUM_MSG

#define MSG_PRI_NORMAL 10
#define MSG_PRI_URGENT 20

#define MQ_NAME "simple_queue"

mqd_t msgQId;

void sendTask1(void)
{
  static char buf[MAX_MSG_LEN];
  int i;

  for (i = 0; i < NUM_MSGS; i++) {

    /* Yield to allow other tasks to run. Otherwise FIFO means that
    this task will run until blocked. */
    sched_yield();

    /* Write and send a message. */
    sprintf(buf,"Send1 %d", i);
    if (mq_send(msgQId, buf, strlen(buf)+1, MSG_PRI_NORMAL) < 0) {
      printf("Send 1 msgQSend failed!\n");
      return;
    }

    if (i == 3) {
      /* Send an urgent message. */
      sprintf(buf,"Send1 URG %d", i);
      if (mq_send(msgQId, buf, strlen(buf)+1, MSG_PRI_URGENT) < 0) {
        printf("Send 1 msgQSend failed!\n");
        return;
      }
    }
  }
  return;
}

void sendTask2(void)
{
  static char buf[MAX_MSG_LEN];
  int i;

  for (i = 0; i < NUM_MSGS; i++) {
    sched_yield();
    sprintf(buf,"Send2 %d", i);
    if (mq_send(msgQId, buf, strlen(buf)+1, MSG_PRI_NORMAL) < 0) {
      printf("Send 2 msgQSend failed!\n");
      return;
    }
    if (i == 5) {
      sprintf(buf,"Send2 URG %d", i);
      if (mq_send(msgQId, buf, strlen(buf)+1, MSG_PRI_URGENT) < 0) {
        printf("Send 2 msgQSend failed!\n");
        return;
      }
    }
  }
  return;
}


void receiveTask1(void)
{
  char buf [MAX_MSG_LEN];
  int i;
  unsigned prio;

  for (i = 0; i < NUM_MSGS; i++) {
    if (mq_receive(msgQId, buf, MAX_MSG_LEN, &prio) < 0) {
      printf("Rx task 1 msgQReceive failed.\n");
    } else {
      printf("Task 1 received %s\n", buf);
    }
  }
  return;
}


void receiveTask2(void)
{
  char buf [MAX_MSG_LEN];
  int i;
  unsigned prio;

  for (i = 0; i < 10; i++) {
    if (mq_receive(msgQId, buf, MAX_MSG_LEN, &prio) < 0 ) {
      printf("Rx task 2 msgQReceive failed.\n");
    } else {
      printf("Task 2 received %s\n", buf);
    }
  }
  return;
}

void msgQ_init(int first)
{
  int i;

  if (first) {
    struct mq_attr mqattr;

    memset(&mqattr, 0, sizeof(mqattr));
    mqattr.mq_maxmsg = MAX_NUM_MSG;
    mqattr.mq_msgsize = MAX_MSG_LEN;
    mq_unlink(MQ_NAME);
    if ((msgQId = mq_open(MQ_NAME, O_RDWR|O_CREAT, S_IRUSR|S_IWUSR,
                    &mqattr)) < 0) {
          printf("msgQCreate failed!\n");
          return;
    }
  }
  else {
    if ((msgQId = mq_open(MQ_NAME, O_RDWR)) < 0) {
          printf("msgQOpen failed!\n");
          return;
    }
  }
  return;
}

int main(int argc, char *argv[])
{
  if (argc == 1) {
    pid_t pid[3];
    char buf[10];
    int i;
    msgQ_init(1);
    for (i=0; i < 3; i++) {
      sprintf(buf, "%d", i+2);
      argv[1] = buf;
      argv[2] = NULL;
      pid[i] = spawnv(P_NOWAIT, argv[0], argv);
    }
    sendTask1();
    for (i=0; i < 3; i++)
      waitpid(pid[i], NULL, WEXITED);
    mq_unlink(MQ_NAME);
  }
  else {
    int num;
    num = atoi(argv[1]);
    msgQ_init(0);
    switch(num) {
      case 2:
        sendTask2();
        break;
      case 3:
        receiveTask1();
        break;
      case 4:
        receiveTask2();
        break;
      default:
        break;
    }
  }
  return;
}
```

Shared memory

Shared memory can be used for simple sharing of data. Since all tasks share the same address space, sharing data structures is straightforward. Tasks can create data structures and use pointers to directly reference the data by code that is running in different contexts.

QNX Neutrino provides access to shared memory via standard POSIX *shm_** calls, which permit threads in different processes (that don't share the same address space) to communicate with each other. Threads in the same process can, as in VxWorks, declare global data structures that are accessible using direct pointer methods in code in different contexts. This requires that the simultaneous access to the shared data be protected using proper synchronization primitives.

Care has to be taken that correct address information is passed between the threads. Virtual memory usage within QNX Neutrino can result in different shared- memory block starting addresses, so all pointers should be calculated relative to the start of the shared-memory block allocated, not to an absolute address.

Multiple threads within a process share the memory of that process. To share memory between processes, you must first create a shared-memory region and then map that region into your process's address space. The *shm_open()* function takes the same arguments as *open()* and returns a file descriptor to the object. As with a regular file, this function lets you create a new shared-memory object or open an existing shared-memory object.

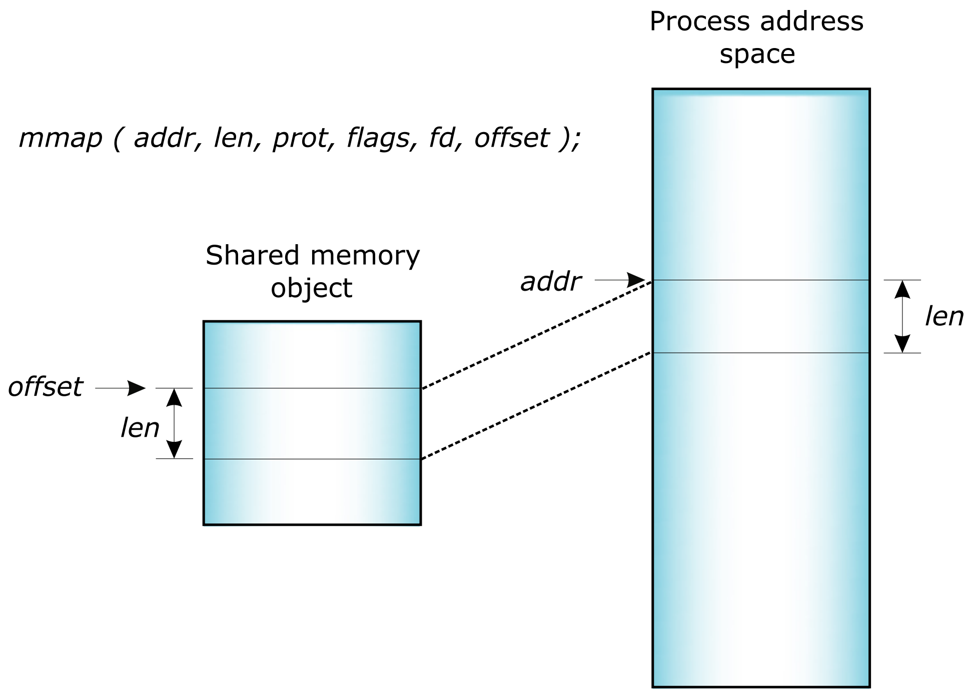
When a new shared-memory object is created, the size of the object is set to zero. To set the size, you use the *ftruncate()* or *shm_ctl()* function. Note that *ftruncate()* is the very same function used to set the size of a file.

Once you have a file descriptor to a shared-memory object, you use the *mmap()* function to map the object, or part of it, into your process's address space. The *mmap()* function is the cornerstone of memory management within QNX Neutrino and deserves a detailed discussion of its capabilities. The *mmap()* function is defined as follows:

```
void *mmap(void *where_i_want_it, size_t length,
            int memory_protections, int mapping_flags,
            int fd, off_t offset_within_shared_memory);
```

In simple terms this says: "Map in *length* bytes of shared memory at *offset_within_shared_memory* in the shared-memory object associated with *fd*."

Figure 8:* *The* mmap() *function in QNX Neutrino.



You can unmap all or part of a shared-memory object from your address space using *munmap()*. This primitive isn't restricted to unmapping shared memory– it can be used to unmap any region of memory within your process. When used in conjunction with the MAP_ANON flag to *mmap()*, you can easily implement a private page-level allocator/deallocator. You can change the protections on a mapped region of memory using *mprotect()*. Like the *munmap()* function, *mprotect()* isn't restricted to shared-memory regions – it can change the protection on any region of memory within your process.

2.4.4 VxWorks shared-memory example

In this example, two tasks declare some shared data region between themselves, and use a semaphore to synchronize access to this region. The semaphore protects against simultaneous access to the region.

```
#include <stdio.h>
#include <sysLib.h>
#include <string.h>
#include <taskLib.h>


SEM_ID mSem;

struct {
    char writeArea[128];
} shareThis;


void shared1(void)
{
    int i;

    for (i = 0; i < 10; i++) {

        /* Shared area is mutex-protected. */
        semTake (mSem, WAIT_FOREVER);

        /* Delay for 2 seconds */
        taskDelay(sysClkRateGet()*2);

        /* Print and then change the shared memory. */
        printf(shareThis.writeArea);
        sprintf (shareThis.writeArea,"This is shared1 (%d).\n",i);

        /* Release the mutex. */
        semGive(mSem);
    }
}


void shared2(void)
{
    int i;

    /* Delay for one second to let the shared1 task grab
       the mutex. */
    taskDelay(sysClkRateGet());

    for (i = 0; i < 10; i++) {
        /* Shared area is mutex-protected. */
        semTake(mSem, WAIT_FOREVER);
        printf("Shared 2 reads %s", shareThis.writeArea);
        semGive(mSem);
    }

}
```

```
void share_init(void)
{
    int pri;

    /* Priority less than init's priority, so tasks start and wait
       for init to complete. */
    taskPriorityGet(0, &pri);
    pri++;


    mSem = semMCreate(SEM_Q_PRIORITY);
    if (mSem == NULL) {
        printf("semMCreate failed\n");
        return;
    }

    strcpy(shareThis.writeArea,"Initial starting string\n");


    if ( taskSpawn("Shared1",
                   pri,
                   0,
                   1000,
                   (FUNCPTR)shared1,
                   0,0,0,0,0,0,0,0,0,0) == ERROR)
    {
        printf("taskSpawn 1 failed!\n");
    }

    if (taskSpawn("Shared2",
                   pri,
                   0,
                   1000,
                   (FUNCPTR)shared2,
                   0,0,0,0,0,0,0,0,0,0) == ERROR)
    {
        printf("taskSpawn 2 failed!\n");
    }


}
```

2.4.5 QNX Neutrino version (shared memory)

Again under QNX Neutrino, we have two options to implement the same functionality. In the single-process example below, threads within the same process automatically share memory. To protect against the simultaneous access to the shared region, we'll need a mutual-exclusion mechanism. This is done using a mutex lock.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <sys/types.h>
#include <sys/QNX Neutrino.h>
#include <pthread.h>
#include <unistd.h>


pthread_mutex_t *mmutex;

struct {
    char writeArea[128];
} shareThis;


void *shared1(void *arg)
{
    int i;

    for (i = 0; i < 10; i++) {

        /* Shared area is mutex-protected. */
        pthread_mutex_lock(mmutex);

        /* Print and then change the shared memory. */
        printf(shareThis.writeArea);
        sprintf (shareThis.writeArea,"Thread 1 Writes: This is shared 1
(%d).\n",i);

        /* Release mutex. */
        pthread_mutex_unlock(mmutex);

        /* Delay for 2 seconds */
        delay(2000);
    }
    return(NULL);
}


void *shared2(void *arg)
{
    int i;

    /* Delay for one second to let the shared1 task grab
       the mutex. */
    delay(1000);

    for (i = 0; i < 10; i++) {
        /* Shared area is mutex-protected. */
        pthread_mutex_lock(mmutex);
        printf("Thread 2 reads: %s", shareThis.writeArea);
        pthread_mutex_unlock(mmutex);
        /* Delay for 1 seconds */
        delay(1000);
    }
    return(NULL);
}
```

```
void share_init(void)
{
  int pri;
  pthread_t tid[2];
  pthread_attr_t attrib;
  struct sched_param param;
  struct sched_param our_param;
  int i;

  pthread_attr_init (&attrib);
  pthread_attr_setinheritsched (&attrib, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy (&attrib, SCHED_RR);
  sched_getparam(0, &our_param);
  param.sched_priority = our_param.sched_priority-1;
  pthread_attr_setschedparam (&attrib, &param);

  mmutex = malloc(sizeof(pthread_mutex_t));
  pthread_mutex_init(mmutex, NULL);

  strcpy(shareThis.writeArea,"Initial starting string\n");


  if ( pthread_create(&tid[0], &attrib, shared1, NULL) < 0)
  {
        printf("taskSpawn 1 failed!\n");
  }
  if ( pthread_create(&tid[1], &attrib, shared2, NULL) < 0)
  {
        printf("taskSpawn 2 failed!\n");
  }
  for (i=0; i < 2; i++) {
    pthread_join(tid[i], NULL);
  }
  return;
}


int main(int argc, char *argv[])
{
  share_init();
  return(0);
}
```

2.4.6 QNX Neutrino version using separate processes

When using multiple processes, shared memory needs to be explicitly defined. In this example, two processes map in the appropriate shared memory region so that both processes can access this region. A mutex is defined and placed in shared memory, and is used to synchronise the two processes.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
```

```
#include <sys/QNX Neutrino.h>
#include <pthread.h>
#include <unistd.h>
#include <spawn.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/mman.h>

#define roundup(x, y) ((((x)+((y)-1))/(y))*(y))

#define SHAREDNAME "/shared_test"

struct sharedarea {
  pthread_mutex_t mmutex;
  char writeArea[128];
};

struct sharedarea *sharedThis;

void shared1(void)
{
  int i;

  for (i = 0; i < 10; i++) {

    /* Shared area is mutex-protected. */
    pthread_mutex_lock(&(sharedThis->mmutex));

    /* Print the shared memory before changing it. */
    printf("%s", sharedThis->writeArea);
    sprintf(sharedThis->writeArea,
            "Proc 1 Writes: This is shared 1 (%d).\n",i);

    /* Release the mutex. */
    pthread_mutex_unlock(&(sharedThis->mmutex));

    /* Delay for 2 seconds */
    delay(2000);
  }
  return;
}


void shared2(void)
{
  int i;

  /* Delay for one second to let the shared1 task grab the mutex. */
  delay(1000);

  for (i = 0; i < 10; i++) {
    /* Shared area is mutex-protected. */
    pthread_mutex_lock(&(sharedThis->mmutex));
    printf("Proc 2 reads: %s", sharedThis->writeArea);
    pthread_mutex_unlock(&(sharedThis->mmutex));
    /* Delay for 1 seconds */
    delay(1000);
  }
  return;
}


void share_init(int first)
{
  int size;
  int fd;
  void *addr;
  if (first) {
    shm_unlink(SHAREDNAME);
    fd = shm_open(SHAREDNAME,O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
  }
  else {
    fd = shm_open(SHAREDNAME, O_RDWR, 0);
  }
  size = sizeof(*sharedThis);
  size = roundup(size, 4096); // round to page size
  addr = (void *)mmap(0,size,PROT_READ|PROT_WRITE, MAP_SHARED,fd,
(long)0);
  close(fd);
  sharedThis = (struct sharedarea *)addr;
  return;
}


int main(int argc, char *argv[])
{
  pid_t pid;
  pthread_mutexattr_t attr;

  if (argc == 1) {
    shm_unlink(SHAREDNAME);
    share_init(1);
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&(sharedThis->mmutex), &attr);
    strcpy(sharedThis->writeArea,"Initial starting string\n");
    argv[1] = "1";
    argv[2] = NULL;
    pid = spawnv(P_NOWAIT, argv[0], argv);
    shared1();
    waitpid(pid, NULL, WEXITED);
    shm_unlink(SHAREDNAME);
  }
  else {
    share_init(0);
    shared2();
  }
  return(0);
}
```

Sockets and remote procedure calls

Sockets provide a means of bi-directional communication between tasks, both locally and across a network. When creating the socket, you also specify the protocol (either UDP or TCP). TCP sockets provide reliable two-way communication between the two ends of the socket. VxWorks sockets are code-compatible with the BSD socket interface. QNX Neutrino provides a socket interface that's also based on the BSD model. Remote procedure call (RPC) facilities are also available within both operating systems.

Signals

Signals are a software-signaling mechanism that can be used for intertask communication. Signals are delivered asynchronously to the execution of the receiving task. Tasks can associate signal handlers with specific signals; these handlers will be invoked whenever the appropriate signal is delivered. VxWorks supports the POSIX 1003.1 signal interface as well as the BSD signal interface. QNX Neutrino also provides both the POSIX and the BSD semantic-based signal facilities.

Watchdog timers

A VxWorks watchdog timer allows tasks to execute functions based on the expiration of timers with a given delay. A task creates a timer, and then starts it by associating the timer with the delay and with a function called upon the expiry of the delay.

If the timer is cancelled before the delay expires, the timer won't fire. Note that the function passed to the watchdog timer is executed in the context of the system tick ISR. This poses restrictions on the functionality that can be implemented in the callback routine. An implementation of the POSIX 1003.1b clock and timer interface is also available.

QNX Neutrino implements the POSIX 1003.1b clock and timer interface that allows threads to signal themselves at a certain time in the future. QNX Neutrino provides a rich event-delivery mechanism that can be used to receive the notification of system events, including timer expirations.

2.4.7 VxWorks watchdog timer example

In this example, a timer is created. When the timer fires, the watchdog routine signals a task by using a semaphore. The task that's waiting for this signal goes ahead and performs the required processing, and then rearms the timer.

```
#include <stdio.h>
#include <sysLib.h>
```

```
#include <string.h>
#include <taskLib.h>

#include <semLib.h>
#include <wdLib.h>
#include <logLib.h>



WDOG_ID wDog;
SEM_ID bSem;

void wDogRoutine(void)
{
    logMsg("\nwDog fired!\n",0,0,0,0,0,0);
    semGive(bSem);
}



void watchDogTask(void)
{
    int i;
    for (i = 0; i < 10; i++) {
        /* Wait for signal from wdog to go ahead. */
        semTake(bSem, WAIT_FOREVER);

        printf("Task received watch dog sem.\n");

        /* Fire off the watch dog again in one second. */
        if (wdStart(wDog, sysClkRateGet(),
                   (FUNCPTR)wDogRoutine, 0) == ERROR)
        {
            printf("wdStart failed in task!\n");
        }
    }
}





void watchDog_init(void)
{

    bSem = semBCreate(SEM_Q_PRIORITY, SEM_EMPTY);
    if (bSem == NULL) {
        printf("semBCreate failed\n");
        return;
    }

    wDog = wdCreate();

    if (wDog == NULL) {
        printf("wdCreate failed \n");
        return;
    }
    /* Fire off the watch dog in one second. */
    if (taskSpawn("watchDog",
                  100,
                  0,
                  1000,
                  (FUNCPTR)watchDogTask,
                  0,0,0,0,0,0,0,0,0,0) == ERROR)
    {
        printf("watchDog task spawn failed!\n");
    }

    if (wdStart(wDog, sysClkRateGet(),
                (FUNCPTR)wDogRoutine, 0) == ERROR)
    {
        printf("wdStart failed!\n");
    }


}
```

2.4.8 QNX Neutrino version (watchdog timer)

Under QNX Neutrino, timers are used in much the same way. A thread is created and dedicated to receiving a pulse from the kernel when the timer fires. The watchdog routine signals the waiting thread when the timer fires, by signaling on a condition variable. After performing the necessary processing, the thread rearms the timer.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/QNX Neutrino.h>
#include <pthread.h>


pthread_mutex_t wd_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t wd_cond = PTHREAD_COND_INITIALIZER;
struct sigevent event;
int timer_chid;
timer_t timer_id;
struct itimerspec itime;

#define TIMER_PULSE_CODE   _PULSE_CODE_MINAVAIL


void *wDogRoutine(void *arg)
{
  int rcvid;
  struct _pulse msg;
  while (1) {
    rcvid = MsgReceive(timer_chid, &msg, sizeof(msg), NULL);
    if (rcvid == 0) { /* we got a pulse */
```

```
      if (msg.code == TIMER_PULSE_CODE) {
        printf("\nwDog fired!\n");
        pthread_cond_signal(&wd_cond);
      }
    }
  }
  return(NULL);
}

void *watchDogTask(void *arg)
{
  int i;
  for (i = 0; i < 10; i++) {
    /* Wait for a signal from wdog to go ahead. */
    pthread_mutex_lock(&wd_mutex);
    pthread_cond_wait(&wd_cond, &wd_mutex);
    pthread_mutex_unlock(&wd_mutex);

    printf("Task received watch dog sem: %d.\n", i);

    itime.it_value.tv_sec = 1;
    itime.it_value.tv_nsec = 0;
    timer_settime(timer_id, 0, &itime, NULL);
  }
  return(NULL);
}


void watchDog_init(void)
{
  pthread_t tid[2];
  pthread_attr_t attrib;
  struct sched_param param;
  struct sched_param our_param;
  int i;

  setbuf(stdout, NULL);
  timer_chid = ChannelCreate(0);
  event.sigev_notify = SIGEV_PULSE;
  event.sigev_coid = ConnectAttach(0, 0, timer_chid,
                      _NTO_SIDE_CHANNEL, 0);
  event.sigev_priority = getprio(0);
  event.sigev_code = TIMER_PULSE_CODE;
  timer_create(CLOCK_REALTIME, &event, &timer_id);

  itime.it_value.tv_sec = 1;
  itime.it_value.tv_nsec = 0;
  timer_settime(timer_id, 0, &itime, NULL);

  pthread_attr_init (&attrib);
  pthread_attr_setinheritsched (&attrib, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy (&attrib, SCHED_RR);
  sched_getparam(0, &our_param);
  param.sched_priority = our_param.sched_priority;
  pthread_attr_setschedparam (&attrib, &param);

  if ( pthread_create(&tid[0], &attrib, wDogRoutine, NULL) < 0)
  {
    printf("watchDog routine spawn failed!\n");
  }
  if ( pthread_create(&tid[1], &attrib, watchDogTask, NULL) < 0)
  {
    printf("watchDog task spawn failed!\n");
  }

  pthread_join(tid[1], NULL);
  return;
}

int main(int argc, char *argv[])
{
  watchDog_init();
}
```

2.5 Exception handling

2.5.1 VxWorks

Hardware exceptions are handled in VxWorks through the *excLib* library functions. During system startup, the exception vector table is initialized using CPU architecture-dependent library calls. Once the kernel has been started, the *excInit()* function call takes care of setting up the OS facilities to deal with exceptions, including spawning *excTask()*, the task-level exception handler.

When a task causes an exception, the default exception handler dumps exception- and task-specific information to the output console, and then simply suspends the task in question so that it can be examined later for debugging purposes. Recovery from the exception may or may not be possible at this point (usually not).

User exception code can be added using the *excHookAdd()* function to augment the default behavior. This code is called at the end of normal exception handling.

It's also possible to use the signal library *sigLib* (a Unix-compatible interface) for dealing with hardware exceptions, in which case *sigvec()* is used to initialize the individual exception vectors. Both a BSD 4.3 and POSIX signal interface are provided by this library, but the two APIs cannot be intermixed. Signals make it possible for tasks to trap and handle exceptions.

2.5.2 QNX Neutrino

QNX Neutrino provides several different mechanisms to both recover from and to deal with exceptions. For example, suppose a device driver crashes because it tried to write to memory that was allocated to another process. The MMU will alert the microkernel, which in turn will cause a dump file to be generated for postmortem analysis. Viewing this dump file, you can immediately determine which line of code is the culprit and then prepare a fix that you can download to all other units in the field before they run into the same bug.

When a process terminates, the kernel automatically reclaims all system resources associated with that process. This not only makes application resource management much simpler and reliable, but also minimizes the fragmentation caused by resources that aren't available because of a lack of automatic reclamation.

With respect to exception handling, many systems, based on conventional monolithic operating systems, aren't designed to include automatic fault-detection. Instead, they rely on a manual approach, i.e. an operator who monitors the health of the system. If the system state is deemed invalid, then the operator takes the appropriate action, which usually includes a complete system reset.

If all tasks – including critical system-level services – share the very same address space, then the integrity of one task can put the integrity of the entire system at risk. If a single component such as a device driver fails, the RTOS itself could fail! In high-availability (HA) terms, each software component becomes a single point of failure (SPOF). What is really needed here is a more modular approach.

Full MMU-supported memory protection between system processes makes it easy to isolate and protect individual processes. The QNX Neutrino process model also offers dynamic creation and destruction, which is especially important for HA systems, because you can more readily perform fault-detection, recovery, and live upgrades in the field. In addition, the process model lets you easily monitor external tasks, which aids in detecting and diagnosing faults.

2.5.3 Conclusions

- Similar APIs exist in both operating systems for exception handling. The degree of porting effort required will depend on which exception library was used within the VxWorks application.

- QNX Neutrino has default exception handlers that are inherently thread/process-based, rather than system-based.

- Signals indicating severe faults in an application under QNX Neutrino can be addressed by restarting a *process*, with application resources being automatically reclaimed by the system.

  This is usually impossible with VxWorks applications (a full system reset is typically initiated). The requirements and recovery strategies for any "health audit" application should be reconsidered with this capability in mind.

3 ELEMENTS OF PORTING

As indicated in Section 1, the ease of porting code is closely related to how tightly the code being ported is tied to the hardware. Device drivers, for example, will likely require extensive redevelopment given the fundamentally different philosophies of the operating systems, while high-level applications can be potentially ported using a simple "recompile, link, and run" method.

Porting efforts generally start with bringing up the OS on the hardware platform (essentially the Board Support Package portion), followed by implementing the appropriate device drivers, integrating them into the operating system, and finally porting the overall higher-layer application. Note that many device drivers are already available in various QNX Neutrino BSPs.

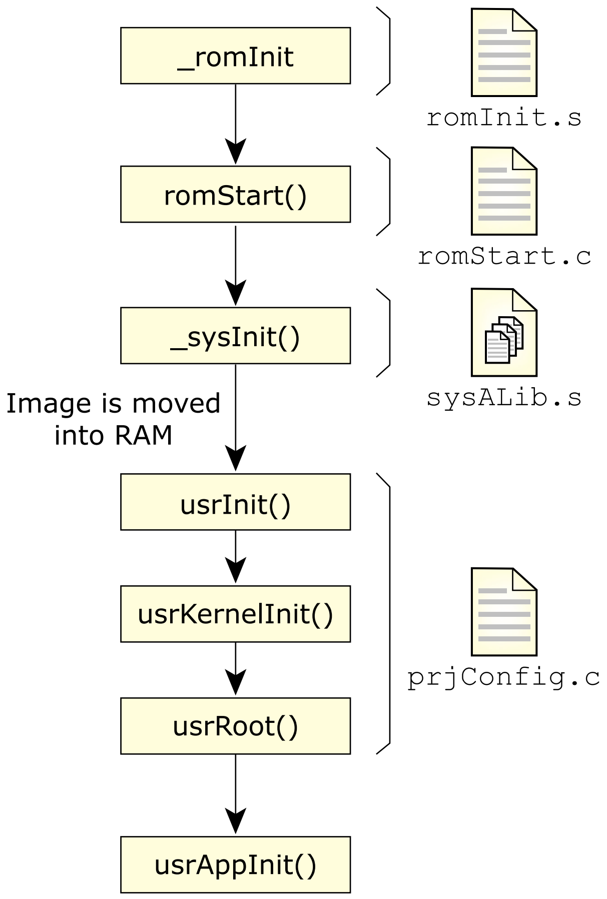
In this section we'll examine each of these elements in the development lifecycle and draw conclusions on the porting efforts involved.

3.1 Startup code

3.1.1 VxWorks startup

The basic VxWorks startup sequence for embedded applications starts with assembly code executed on reset either via the `_romInit` code or the BIOS, depending on the hardware used. This code performs the minimum functionality required to set up the CPU and memory in order to allow the VxWorks application to be loaded into RAM.

Figure 9: VxWorks startup sequence.



Following the execution of the reset code, an image loader (*_romStart()* located in **romStart.c**) is run. This code is responsible for moving the OS/application image from permanent storage (e.g. flash) into RAM and often involves decompression as well as copying. If required, data memory used by the OS and application is zeroed at this point.

Once the image is in RAM, the execution jumps from the ROM into a small amount of assembly in RAM (`_sysInit`) that sets up the processor and stack frame so that a jump into the COS initialization routine *`usrInit`()* an be initiated.

The *usrInit()* function handles the remainder of the setup required to allow the kernel to be initialized (which includes initializing the exception vectors, cache library, interrupts, and any additional hardware initialization). The kernel is then brought up with the *kernelInit()* call. Kernel initialization results in *usrRoot()* being executed.

The *usrRoot()* call takes care of bringing up the remainder of the kernel system (e.g. semaphores, message queues, task support, etc.), initializes the memory pool subsystem, I/O infrastructure, network (if desired), tools, filesystems and other supporting hardware, and finally launches *userAppInit()*, which contains the user-supplied code for starting the user application.

It's important to note that the permanently stored image used for booting usually consists of a single "executable" containing all of the user application and configured kernel. The OS and supporting infrastructure itself is configured using `#define` macros to determine what should be included or excluded from the final image during the build phase. These macros also determine how the initialization code in **prjConfig.c** behaves to bring up that infrastructure.

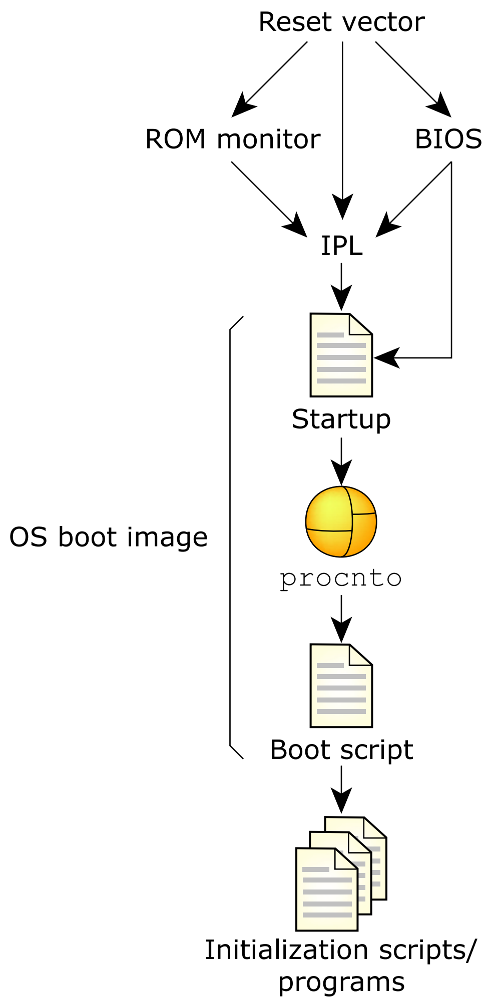
3.1.2 QNX Neutrino startup

From the software perspective, the following steps occur when QNX Neutrino starts up:

Processor begins executing at the reset vector. The Initial Program Loader (IPL) locates the OS image and transfers control to the startup program in the image. Then the startup program configures the system and transfers control to the QNX Neutrino microkernel and process manager (**procnto**). The **procnto** module loads additional drivers and application programs.

The first step performed by the software is to load the OS image. As mentioned above, this is done by the **IPL**. The IPL's initial task is to minimally configure the hardware to create an environment that will allow the startup program, and consequently the QNX Neutrino microkernel, to run. Specifically, this task includes at least the following steps: begin execution from the reset vector, configure the memory controller, which may include configuring chip selects and/or PCI controller, configure clocks, and set up a stack to allow the IPL library to perform OS verification and setup (image download, scan, setup, and jump).

Figure 10: QNX Neutrino startup sequence.



The second step performed by the software is to configure the processor and hardware, detect system resources, and start the OS. As mentioned above, this is implemented by the **startup** program. While the IPL did the bare minimum configuration necessary to get the system to a state where the startup program can run, the startup program's job is to "finish up" the configuration. If the IPL detected various resources, it would communicate this information to the startup program (so it wouldn't have to redetect the same resources.) To keep QNX Neutrino as configurable as possible, the startup program has the ability to program such things as the base timers, interrupt controllers, cache controllers, and so on. It can also provide kernel callouts, which are code fragments that the kernel can call to perform hardware-specific functions. For example, when a hardware interrupt is triggered, some piece of code must determine the source of the interrupt, while another piece of code must be able to clear the source of the interrupt. Note that the startup program doesn't configure such things as the baud rate of serial ports. Nor does it initialize standard peripheral devices like an Ethernet controller or EIDE hard disk controller – these are left for the drivers to do themselves when they start up later.

Once the startup code has initialized the system and has placed the information about the system in the system page area (a dedicated piece of memory that the kernel will look at later), the startup code is responsible for transferring control to the QNX Neutrino kernel and process manager (**procnto**), which perform the final loading step.

To bring up the OS, a single image file is used. This image contains, at a minimum, the OS image itself as well as components required to support basic OS functionality. Application images may also be included, but these are normally stored *in separate filesystems* (e.g. flash or disk) and started up by a script that is contained in the image file.

The image file is actually organized into a small filesystem (called the image filesystem) that has a directory structure for referencing the resident files (application image, libraries, OS image, startup script, etc.). A program called **mkifs** (make image filesystem) uses command-line information in conjunction with a "buildfile" in order to produce the final image.

3.1.3 Conclusions

- You may be able to reuse some of the reset vector code with changes made to the exit point to branch into the startup code. While the reset code is quite different in format, the general scope of desired functionality is the same.
- The remainder of the OS bring-up code will have to be rewritten.
- The startup code in QNX Neutrino contains similar functionality to the *romStart/usrInit* code in VxWorks.
- The QNX Neutrino buildfile contains similar functionality to the *usrRoot()* function in VxWorks.
- The QNX Neutrino kernel doesn't need to be configured. The kernel is provided as a single unit for inclusion in a buildfile. Both instruction caches and data caches are automatically enabled by the kernel, which means that device drivers may have to allocate cache-safe memory.
- The operating system and applications may be stored in different places and in different formats in QNX Neutrino (i.e. the OS itself is part of the buildfile that is used for booting the system). The application may be stored in a persistent filesystem (disk, flash, etc.) apart from the OS. This is different from VxWorks, in which the application and OS must be built into a single image.

3.2 Hardware Input/Output

Various types of I/O are supported (character, block, stream, network, etc.) by both operating systems. Mid-level infrastructure is provided to integrate hardware components into the operating systems so that consistent APIs are available for accessing the hardware. At the low level, device drivers are used to configure, control, and send/receive data from the hardware. At the application layer, standard I/O commands (open, read, write, close, etc.) may be used to access the devices.

This section contrasts the various elements of hardware integration within QNX Neutrino and VxWorks.

3.2.1 Interrupt Service Routines

3.2.1.1 VxWorks

ISRs in VxWorks are dealt with via the **intArchLib** (architecture-dependent interrupt library). This library provides routines for locking/unlocking, enabling/disabling interrupts, for attaching handlers to hardware interrupts (*intConnect()*), and for dealing with the interrupt vector table (*intVec**). Other routines are available, depending on the CPU family being used.

Here are the main ISR functions used within VxWorks applications:

ISR Function	Description
intLock/intUnlock	Lock out or unlock interrupts
intEnable/intDisable	Enable/disable interrupts indicated in parameters
intConnect	Connect an interrupt-handler to a specified interrupt
intVecGet/intVecSet	Set/get an interrupt vector value
intContext	Check and see if in an interrupt or task state
intCount	Determine the current interrupt nesting count

Here's an example of an interrupt handler and how it is attached in the PowerPC architecture:

```
DRV_CTRL_S drvCtrl;
SEM_ID packetSem;
```

```
LOCAL void motFccISR(DRV_CTRL_S *pDrvCtrl)
{
struct fcc_regs  *fcc_regs;
unsigned int     events;
PQ2IMM         *pQ = (PQ2IMM *)vxImmrGet();

fcc_regs  = &pQ->fcc_regs[pDrvCtrl->fccNum - 1];

/* Save the event register. */
events = fcc_regs->fcc_fcce;

/* Clear the event that’s causing the interrupt. */
fcc_regs->fcc_fcce = 0xffff0000;

/* Other processing. */
.
.
.
/* Signal to task. */
semGive(packetSem);

}


DRV_CTRL_S drvCtrl;

attachInterrupt(void) {
    .
    .
    .
    taskSpawn(“tPacketProcessor”,100,0,5000,
              (FUNCPTR)packetProcessor,
               0,0,0,0,0,0,0,0,0,0);
    packetSem = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY);
    drvCtrl.fccNum = 1;
    intConnect(INUM_TO_IVEC(INUM_FCC1),
           (VOIDFUNCPTR) motFccISR, (int)(&drvCtrl));
    .
    .
    .

}



packetProcessor(void) {

    while(1) {
        semTake(packetSem);
        /* Process packet */
        .
        .
        .
    }

}
```

The handler is attached to the interrupt using a macro to map from the interrupt number to the interrupt vector. The handler address and a data value (used as a parameter to the handler when invoked) are also passed as parameters.

Within the handler, the internal memory-mapped region of the processor is accessed directly (using a predefined PQ2IMM structure and information contained in the passed data structure) to determine the appropriate register to access to clear the interrupt. Additional processing can then be undertaken to handle the interrupt. Once the interrupt processing has completed, a semaphore is given to notify a task that the interrupt occurred.

There are restrictions on what functions can be called from an ISR in VxWorks. Generally speaking, anything that may result in a blocking call cannot be used. This would include things like *semTake*, *malloc*, *printf*, etc. Communication between the ISR (in "interrupt space") and the task can be carried out through a variety of means, including giving a semaphore or sending a message on a message queue.

3.2.1.2 QNX Neutrino

In QNX Neutrino, in order to install an ISR, the software must tell the OS that it wishes to associate the ISR with a particular source of interrupts. A thread specifies which interrupt source it wants to associate with which ISR, using the *InterruptAttach()* or *InterruptAttachEvent()* function calls. When the software wishes to dissociate the ISR from the interrupt source, it can call *InterruptDetach()*:

```
// Example to show attaching an ISR
#define IRQ3 3
extern const sigevent *handle_int3(void *, int);
...
// Obtain I/O Privileges
ThreadCtl( _NTO_TCTL_IO, 0 );

// Associate an interrupt handler with IRQ 3
id = InterruptAttach(IRQ3, handle_int3, NULL, 0, 0);
...
// Perform some processing
...
// Done; detach the interrupt source.
InterruptDetach(id);
```

The thread attempting to attach to an interrupt must have I/O privileges – the privilege associated with being able to manipulate hardware I/O ports and affect the processor interrupt enable flag. Only the root account can gain I/O privileges, so this effectively limits the association of interrupt sources with ISR code.

In the example above, the function *handle_int3()* is the ISR. In general, an ISR is responsible for determining which hardware device requires servicing, performing some kind of servicing of that hardware (usually this is done by simply reading and/or writing the hardware's registers), updating some data structures shared between the ISR and some of the threads running in the application, and signaling the application that some kind of event has occurred.

In general, to actually service the interrupt, the ISR has to do very little – the minimum it can get away with is to clear the source of the interrupt and then schedule a thread to actually do the work of handling the interrupt. This is the recommended approach, for a number of reasons. First of all, context-switch times between the ISR completing and a thread executing are very small– typically on the order of a few microseconds. Also, the ISR can execute only a limited set of functions (including seven kernel functions). Moreover, the ISR runs at a priority higher than any software priority in the system – having the ISR consume a significant amount of processor time could have a negative impact on the realtime characteristics of the system.

When the ISR is servicing the interrupt, it can't make any kernel calls — aside from the seven safe ones. This means that the ISR really shouldn't call any library functions, because their underlying implementation may use kernel calls. Since our library is thread-safe, even some of the "simple" library calls may try to allocate a mutex, (possibly) resulting in a kernel call. Because the *str*()* and *mem*()* functions (such as *strcpy()* and *memcpy()*) are very useful in an ISR, these functions are guaranteed safe to call, with the notable exception of *strdup()*, which allocates memory and therefore uses a mutex. The QNX Neutrino *Library Reference* identifies the functions that you can call from an ISR.

Another issue that arises when using interrupts is how to safely update data structures in use between the ISR and the threads in the application. Since the ISR runs at a higher priority than any software thread and since the ISR can't issue kernel calls (except as noted), standard thread-level synchronization mechanisms (such as mutexes, condvars, etc.) can't be used. It's up to the thread to protect itself against any preemption caused by the ISR. Therefore, the thread will issue *InterruptDisable()* and *InterruptEnable()* calls around any critical data manipulation operations. Since these calls effectively turn off interrupts, the thread should keep the data-manipulation operations to a bare minimum.

Since the environment the ISR operates in is very limited, generally you would want to perform most (if not all) of the actual "servicing" operations at the thread level. You may decide that some time-critical functionality needs to be done in the ISR, with a thread being scheduled later to do the "real" work, or, you may decide that nothing needs to be done in the ISR; just a thread needs to be scheduled. This is effectively the difference between *InterruptAttach()* (where an ISR is attached to the IRQ) and *InterruptAttachEvent()* (where a `struct sigevent` is bound to the IRQ).

After it has read some registers from the hardware or done whatever processing is required for servicing, the ISR may or may not decide to schedule a thread to actually do the work. In order to schedule a thread, the ISR simply returns a pointer to a `const struct sigevent` structure; the kernel looks at the structure and delivers the event to the destination. If the ISR decides not to schedule a thread, it simply returns a NULL value.

The event returned can be a signal or a pulse. You may find that a signal or a pulse is satisfactory, especially if you already have a signal or pulse handler for some other reason. Note, however, that for ISRs we can also return a `SIGEV_INTR`. This is a special event that really has meaning only for an ISR and its associated controlling thread. A very simple, elegant, and fast way of servicing interrupts from the thread level is to have a thread dedicated to interrupt processing. The thread attaches the interrupt (via *InterruptAttach()*) and then the thread blocks, waiting for the ISR to tell it to do something. Blocking is achieved via the *InterruptWait()* call. This call blocks until the ISR returns a `SIGEV_INTR` event.

3.2.1.3 Interrupt handler example under QNX Neutrino

```
#include <errno.h>
#include <fcntl.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <sys/neutrino.h>
#include <sys/resmgr.h>
#include <sys/stat.h>
#include <sys/syspage.h>

#define INTR_THREAD_PRIO    27

void *interrupt_thread (void *data);

struct sigevent intr_event;
int             intr_id;

#define NUM_DATA_BYTES 1

unsigned int interrupt_num = 0;

int main (int argc, char **argv)
{
    int i;

    while ((i= getopt (argc, argv, "i:")) != -1)
    {
        switch (i)
        {
        case 'i':
            interrupt_num = atoi (optarg);
            break;

        default:
            printf ("Unknown arg \n");
            return (EXIT_FAILURE);
            break;
        }
    }

    fprintf(stderr, "Using Interrupt %u\n", interrupt_num );
    fflush(stdout);

    pthread_create (0, NULL, interrupt_thread, NULL);

    // Now our main() thread just waits till we get killed
    pause();

    return EXIT_SUCCESS;
}

const struct sigevent *interrupt_handler(void *event, int id)
{
    struct sigevent *return_event= (struct sigevent *) event;

    // Uncomment the following lines to pulse every
    // NUM_DATA_BYTES'th interrupt.  We are pretending we get
    // one byte of data per interrupt and only wake up the
    // thread after NUM_DATA_BYTES bytes.
    // Why do the work of getting the bytes of data from within
    // the thread? Because we are pretending that if we don't
    // do it at handler time, then we'll miss the data (e.g. no
    // buffering on the hardware).

//  static unsigned interrupt_counter = 0;

//  interrupt_counter++;
//  if ( interrupt_counter == NUM_DATA_BYTES ) {
//      interrupt_counter = 0;
        return return_event;
//  } else {
//      return NULL;
//  }
}

void *interrupt_thread (void *notinuse)
{
    unsigned counter=0;
    struct sched_param param;

    param.sched_priority = INTR_THREAD_PRIO;
    if ( sched_setscheduler( 0, SCHED_FIFO, &param ) == -1 ) {
        fprintf(stderr, "Unable to change priority: %s\n",
                strerror(errno));
        exit( EXIT_FAILURE );
    }

    // We need I/O privileges so we can call InterruptAttach()
    if (ThreadCtl (_NTO_TCTL_IO, 0) == -1) {
        fprintf(stderr, "Unable to get I/O privileges: %s\n",
                strerror(errno));
        exit( EXIT_FAILURE );
    }
    // Set up a sigevent that will wake up InterruptWait()
    SIGEV_INTR_INIT( &intr_event );

    if ( interrupt_num == 0 ) {
        // Set interrupt_num to the timer interrupt. On x86,
        // this is 0. We do this just to show how to pull info
        // from the SYSPAGE.
        interrupt_num = SYSPAGE_ENTRY (qtime)->intr;
    }

    intr_id = InterruptAttach (interrupt_num, interrupt_handler,
        &intr_event, sizeof(intr_event), _NTO_INTR_FLAGS_TRK_MSK);

    if ( intr_id == -1 ) {
        fprintf(stderr, "Unable to attach to irq %u: %s\n",
                interrupt_num, strerror(errno));
        exit( EXIT_FAILURE );
    }

    for (;;) {

        if ( InterruptWait( NULL, NULL ) == -1 ) {
            fprintf(stderr, "\nInterruptWait interrupted: %s\n",
                    strerror(errno));
        }

        else {
            // The interrupt went off, and the interrupt handler
            // was called, and the interrupt handler then woke
            // us up from InterruptWait() by returning with the
            // SIGEV_INTR event.

            counter++;
            fprintf(stderr, "%u ", counter); fflush(stderr);
        }
    }
}
```

3.2.1.4 Conclusions

- Hardware access in QNX Neutrino ISRs needs to be properly set up in terms of privileges and permissions, memory addressing (translation), and memory accessing (shared-memory mapping) before an ISR can be used.

- ISRs should be as short as possible. Both operating systems have clear restrictions on the type of function calls that can be made within an ISR. QNX Neutrino has greater restrictions on the OS calls that can be used in an ISR.

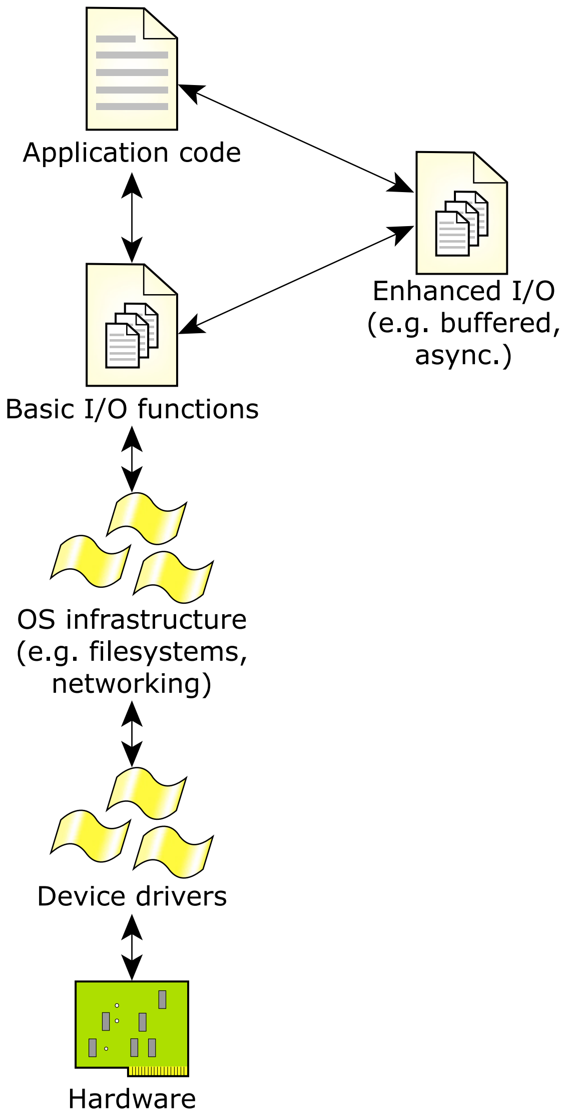
- Simple VxWorks ISRs that clear hardware interrupt causes and perform a *semGive* (to kick a task that performs the remainder of the processing at task level) are relatively straightforward to translate into QNX Neutrino.

3.2.2 I/O

3.2.2.1 VxWorks

The VxWorks I/O system implements a Unix- and ANSI-C-compatible API at the application layer, which provides a consistent interface to hardware (see Figure 11).

Figure 11: VxWorks I/O system.


Application code
Enhanced I/O (e.g. buffered, async.)
Basic I/O functions
OS infrastructure (e.g. filesystems, networking)
Device drivers
Hardware

Seven basic device-independent I/O functions provide this functionality:

I/O Function	Description
creat()	Create a file
open()	Open a device/file

close()	Close a device/file
read()	Read from a device/file
write()	Write to a device/file
ioctl()	Send/receive control and configuration information to/from a device or file
remove()	Remove a device/file

Higher-level standard functions (e.g. socket calls, *printf*, *scanf*, etc.) are also included in the operating system libraries.

All of these basic functions use "file descriptors" to uniquely identify the device or file in question. File descriptors in VxWorks are shared between all tasks and are allocated/returned from a single "pool" of descriptors. A maximum number of 255 file descriptors (which you can set during OS bring-up) are available for simultaneous use in the stock VxWorks 5.4 system.

File descriptors are allocated during an *open()* or *creat()* call and deallocated on a *close(),* and may be reused after being deallocated. The usual convention of assigning file descriptor 0 to standard input, 1 to standard output, and 2 to standard error is also used.

VxWorks device drivers are broken down into the following categories:

- Character-based (e.g. RS-232)
- Block access (e.g. disk drives)
- Networking
- Asynchronous (implemented using a POSIX-compliant API)

Filesystems in VxWorks sit on top of block device drivers. The block device drivers act as the intermediary between the storage media (flash, rotating, etc.) and the upper filesystem. VxWorks supports the following filesystems:

Filesystem	Description
dosFsLib	DOS 4.0-compatible filesystem with 8.3 and VFAT long filenames

rawFsLib	Raw block filesystem
rt11FsLib	RT-11 formatted filesystem (mainly used for backwards compatibility)
cdromFsLib	Read-only ISO 9660 CD-ROM filesystem
nfsLib	Network File System library
tapeFsLib	Sequential tape filesystem

Installing a filesystem is simply a matter of executing a function call from the appropriate library. The call includes the name of the filesystem, the block device driver to attach to, and any other pertinent device-specific filesystem information. Function calls are also available for formatting and checking the media as required. Once the filesystem has been installed, read/writes to the filesystem are implemented through the familiar *stdio* library calls (create, open, read, write, etc.)

3.2.2.2 QNX Neutrino

As in the POSIX and Unix tradition, these devices are located in the OS pathname space under the `/dev` directory. For example, a serial port to which a modem or terminal could be connected might appear in the system as:

```
/dev/ser1
```

In QNX Neutrino, device drivers are broken down into the following categories:

- **Image** – a special filesystem that presents the modules in the image and is always present. The process manager (**procnto**) automatically provides an image filesystem as well as a RAM filesystem.
- **Block**– traditional filesystems that operate on block devices, such as hard disks and CD-ROM drives. This includes the Power-Safe, QNX4, DOS, UDF, and CD-ROM filesystems.
- **Flash** – non-block-oriented filesystems designed explicitly for the characteristics of flash memory devices.
- **Network** – filesystems that provide network file access to the filesystems on remote host computers. This includes the NFS and CIFS (SMB) filesystems.
- **Virtual** – special filesystems (such as the Inflator resource manager) that offer custom facilities to the client.

Under QNX Neutrino, device drivers are typically implemented as *resource managers*. A resource manager is basically a user-level program that accepts messages from other programs and, if necessary, communicates with hardware. The specific binding between the resource manager and the client programs that need to talk to the hardware is maintained via a pathname space mapping; an association is made between a pathname and the resource manager. Client programs continue to use standard POSIX mechanisms (*open()*, *close()*, etc.) to access the resource. Since many of the messages received by most resource managers are for a common set of functions, QNX Neutrino provides default handlers for most of these common functions in a shared library. This allows driver developers to handle these common functions without having to write additional code. The library automatically provides the following common functions (among others):

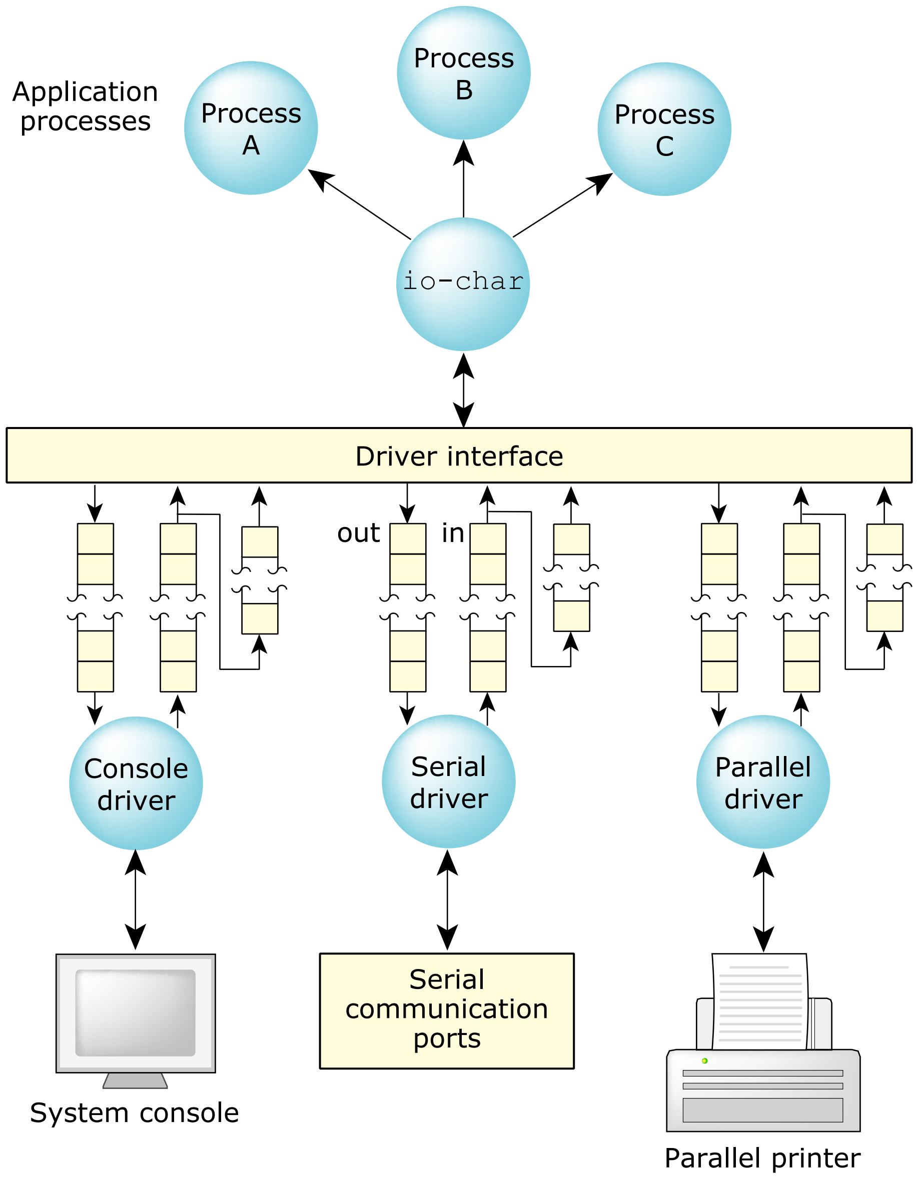
- *open()*
- *close()*
- *read()*
- *write()*
- *stat()*
- *unlink()*
- *fstat()*
- *devctl()*
- *lseek()*

Character-based device drivers are used for such devices as serial ports, parallel ports, text-mode consoles, pseudo terminals (ptys).

Programs access character devices using the standard *open()*, *close()*, *read()*, and *write()* API functions. Additional functions are available for manipulating other aspects of the character device, such as baud rate, parity, flow control, etc.

Since it's common to run multiple character devices, they have been designed as a family of drivers and aggregated in a library called **io-char** to maximize code reuse.

Figure 12: Device I/O in QNX Neutrino.

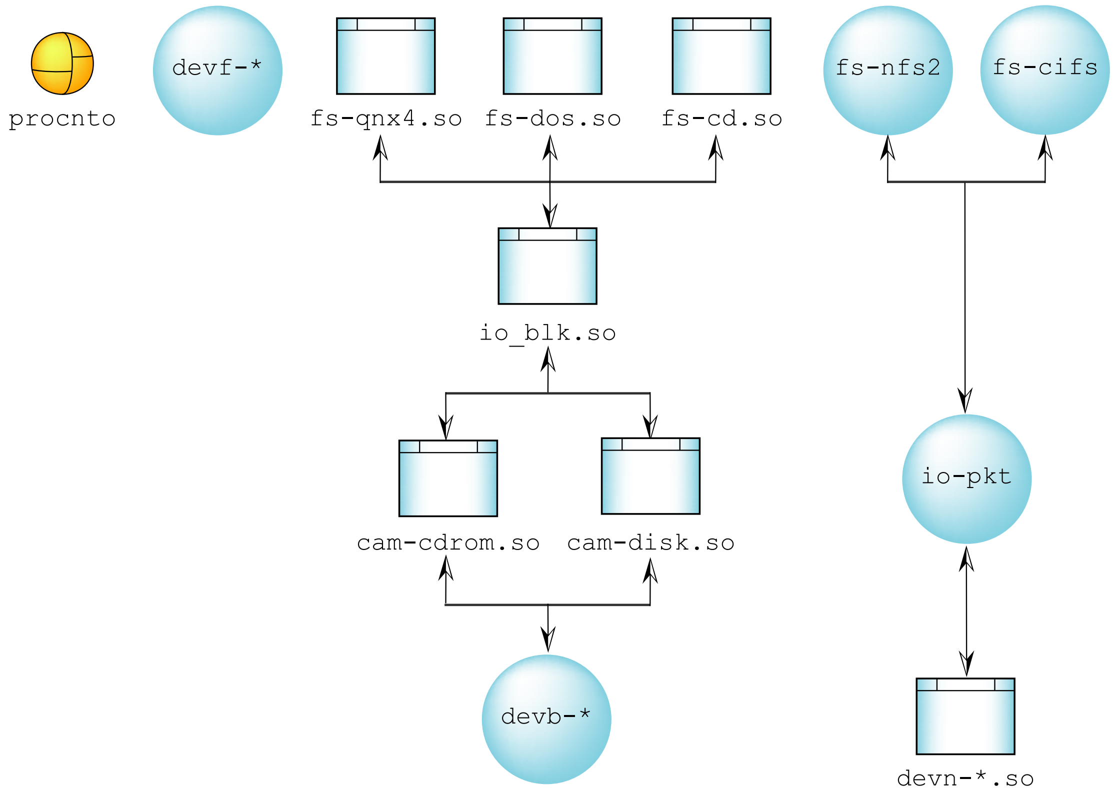


The **io-char** library module contains all the code to support POSIX semantics on the device. It also contains a significant amount of code that implements character I/O features that are beyond POSIX but are desirable in a realtime system. Since this code is in the common library, all drivers inherit these capabilities. The driver is the executing process that calls into the library. In operation, the driver starts first and invokes the **io-char** manager. The drivers themselves are just like any other QNX Neutrino process and can run at different priorities according to the nature of the hardware being controlled and the client's requesting service. Once a single character device is running, the memory cost of adding additional devices is minimal, since only the code to implement the new driver structure would be new.

QNX Neutrino provides a rich variety of filesystems. Like most service-providing processes in the OS, these filesystems execute *outside* the kernel; applications use them by communicating via messages generated by the shared-library implementation of the POSIX API. Most of these filesystems are resource managers. Each filesystem adopts a portion of the pathname space (i.e. a mountpoint) and provides filesystem services through the standard POSIX API (*open()*, *close()*, *read()*, *write()*, *lseek()*, etc.). Filesystem resource managers take over a mountpoint and manage the directory structure below it. They also check the individual pathname components for permissions and for access authorizations.

This implementation permits filesystems to be started and stopped dynamically; multiple filesystems may run concurrently, and applications are presented with a single unified pathname space and interface, regardless of the configuration and number of underlying filesystems.

Figure 13: Filesystem layering in QNX Neutrino.



Since it's common to run many filesystems under QNX Neutrino, they have been designed as a family of drivers and shared libraries to maximize code reuse. This means the cost of adding an additional filesystem is typically smaller than might otherwise be expected. Once an initial filesystem is running, the incremental memory cost for additional filesystems is minimal, since only the code needed to implement the new filesystem protocol would be added to the system. The filesystem shared libraries may be dynamically loaded later to provide filesystem interfaces and services. A "filesystem" shared library implements a filesystem protocol or "personality" on a set of blocks on a physical disk device. Since the filesystems aren't built into the OS kernel, they are dynamic entities that can be loaded or unloaded on demand.

Most of the filesystem shared libraries ride on top of the Block I/O module (**io-blk**). This module also acts as a resource manager and exports a block-special file for each physical device. These files represent each raw disk and may be accessed using all the normal POSIX file primitives (*open()*, *close()*, *read()*, *write()*, *lseek()*, etc.).

3.2.2.3 Conclusions

- At the application layer, the identical ANSI-C/POSIX-compatible API is used by both VxWorks and QNX Neutrino. Consistent support for specific invocations of various routines (e.g. *ioctl*) will depend on how the corresponding device drivers have been implemented.
- The I/O infrastructure used by QNX Neutrino and VxWorks is different and incompatible.
- Installation/integration of a filesystem or device driver into the OS is totally different between the two systems.
- QNX Neutrino provides a much richer selection of filesystem choices than VxWorks.

3.2.3 Device drivers

Device drivers allow the OS and application programs to use the underlying hardware in a generic way (e.g. a disk drive, a network interface).

3.2.3.1 VxWorks

To implement device drivers for direct integration into the VxWorks OS, you need to implement the basic I/O functions (as described in section 3.2.2.1) for the device in question. The underlying implementation of the I/O functions varies depending on the type of device driver being written (e.g. tty, network, disk). The I/O functions include the appropriate responses to any supported IOCTL commands.

Installation of the device driver varies depending upon the type of device. Most devices are installed using the i*osDrvInstall()/iosDevAdd()* functions to enter the I/O functions in the OS Device Driver Table and instantiate a driver. Network device drivers are installed using the *muxDevLoad()/muxBind()* functions. To examine the device driver tables, you can use the *iosShow* (*iosDevShow()/iosDrvShow()*) or *muxShow* routines. The device handler functions may run:

- as a separate task (e.g. portions of the flash filesystem),

- in the context of a system task (e.g. *tNetTask* for the network functions),
- in the context of user tasks (e.g. file I/O functions),

or some combination of the three.

A device driver may also, of course, be implemented with a proprietary API that interacts with the application via some user-defined mechanism.

As in any VxWorks implementation, the drivers operate with the same privilege mode and in the same memory space as the kernel and user application. While adding/removing and starting/stopping device drivers is possible, care has to be taken to ensure that system resources are properly freed on stopping the devices.

3.2.3.2 QNX Neutrino

While most OSs require device drivers to be tightly bound into the OS itself, device drivers for QNX Neutrino *can be started and stopped as standard processes*. As a result, adding device drivers doesn't affect any other part of the OS– drivers can be developed and debugged just like any other application.

Only certain processes are allowed to perform certain operations (e.g. attaching an interrupt) under QNX Neutrino. These processes need to obtain I/O privileges, which are the privileges associated with being able to manipulate hardware I/O ports, etc. Only the **root** account can gain I/O privileges.

3.2.3.3 Conclusions

- The mechanism used to integrate device drivers into the operating system (and therefore the API that has to be implemented by the device driver) is very different in QNX Neutrino and VxWorks. This means that device drivers, for the most part, will have to be rewritten when porting from VxWorks to QNX Neutrino.

- Care must be taken to properly set up memory access (shared memory), addressing (application addresses must be translated to/from physical memory addresses), memory allocation (cache coherency must be taken into account), and privilege/permissions (I/O privileges, root permissions) when writing code that directly accesses hardware in QNX Neutrino.

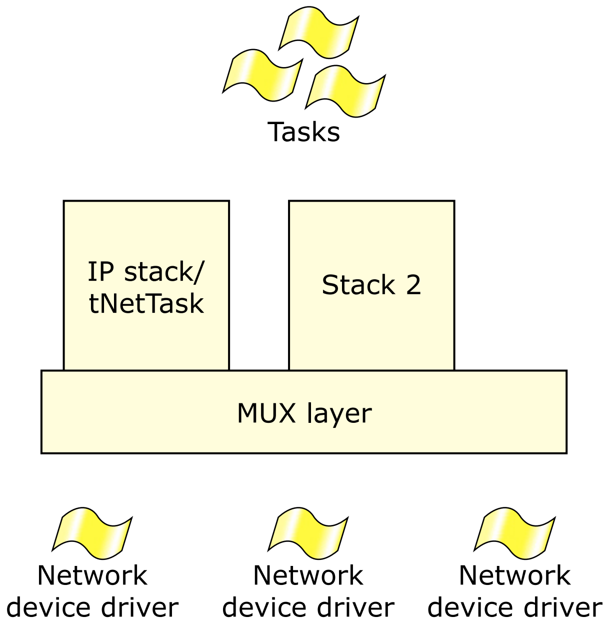
3.3 Networking

The IP networking stack in VxWorks 5.4 / 5.5 is based on the BSD 4.4 implementation; the QNX Neutrino networking stack is based on NetBSD 4.0. There are many similarities in their architectures, allowing potential reuse of device driver elements as discussed in the following sections.

3.3.1 VxWorks networking architecture

The standard VxWorks networking infrastructure is organized into three component levels: device drivers, the MUX layer, and network stacks.

Figure 14: Networking infrastructure in VxWorks.



A network device driver interfaces directly with networking hardware and produces packets that are sent up to the next layer. A user-supplied device driver must provide an API conformant with the MUX layer architecture. Two forms of drivers are possible: an END driver (frame-oriented) or an NPT driver (packet-oriented). Both drivers require that you use an API consisting of the following functions:

load/unload() – attach/detach a driver to/from the MUX layer

send() – send data onto the physical layer

mcastAddrAdd/Del/Get() – multicast address functions

PollSend/Receive() – send packets/frames in polled mode (rather than interrupt-driven)

start/stop() – activate or deactivate the device (including interrupt handlers)

bind() – connect into the protocol stack

ioctl() – I/O control command support function

In addition to these functions, you can also include address-resolution functions (e.g. for ARP) for a protocol/interface, using the *muxAddrResFuncAdd/Get/Del()* function set.

Buffer management for network drivers is handled through the **netBufLib** library calls. This library provides the user with all the facilities required to set up and manage device-level buffers for receiving and sending data.

Note that there is no "interrupt receive" API. This is handled by using the *netJobAdd()* function call inside of the interrupt handler for receiving packets from the device.

The major difference between the END and NPT packets is that NPT drivers strip the datalink header information from the received packets before sending it into the MUX layer, while END drivers include the Ethernet header information.

The MUX layer essentially sits in between the data link layer and the network layer in the OSI network model. Higher-layer protocol stacks connect into the MUX layer using the *muxBind/muxTkBind()* (**ipAttach**) functions and disconnect using the *muxUnbind()* (**ipDetach**) functions. From an application point of view, the *muxBind/muxUnbind()* routines are of some interest, allowing a user to "trap" and potentially process raw packets before they enter the IP stack (a technique commonly called "snarfing").

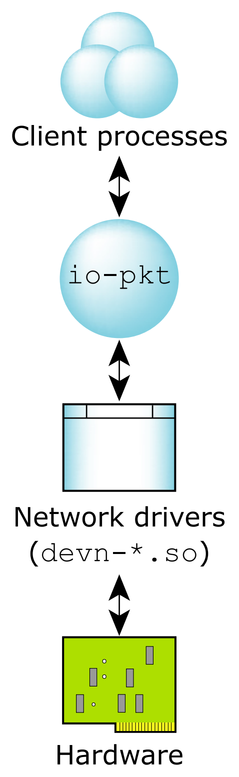
Sitting on the MUX layer are one or more networking stacks that attach to the MUX layer allowing specified devices to receive packets for or send packets from the stack. The standard IP stack used within VxWorks uses *ipAttach/ipDetach()* to attach to the MUX layer.

Global variables may be used to configure the IP stack's operating characteristics (e.g. keep-alive timeout, ARP timeout, etc.).

3.3.2 QNX Neutrino networking architecture

As with other service-providing processes in QNX Neutrino, the networking services execute outside the kernel. The system presents a single unified interface, regardless of the configuration and number of networks involved. This architecture allows network drivers to be started and stopped dynamically; Qnet, TCP/IP, and other protocols can run together in any combination.

Figure 15: Networking architecture in QNX Neutrino.



The native network subsystem consists of the network manager executable (**io-pkt**), plus one or more shared library modules. These modules can include protocols (e.g. **lsm-qnet.so**), drivers (e.g. **devn-ne2000.so**), and filters (e.g. **lsm-pf-v6.so**). There are three variants of io-pkt:

io-pkt-v4

IPv4 version of the stack with no encryption or Wi-Fi capability built in. This is a "reduced footprint" version of the stack that doesn't support the following:

- IPv6
- Crypto / IPSec
- 802.11 a/b/g WiFi
- Bridging
- GRE / GRF
- Multicast routing
- Multipoint PPP

io-pkt-v4-hc

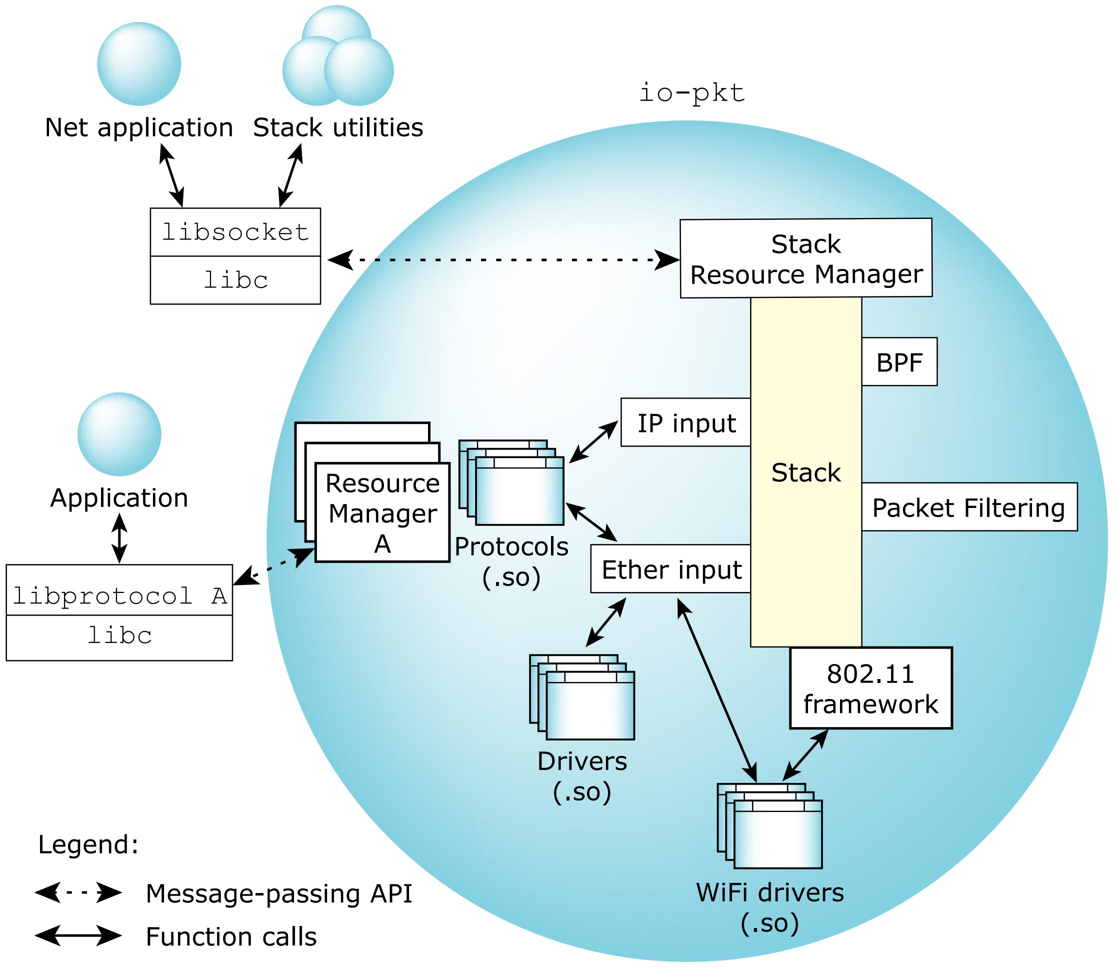
IPv4 version of the stack that has full encryption and Wi-Fi capability built in and includes hardware-accelerated cryptography capability (Fast IPsec).

io-pkt-v6-hc

IPv6 version of the stack (includes IPv4 as part of v6) that has full encryption and Wi-Fi capability, also with hardware-accelerated cryptography.

The **`io-pkt`** stack is very similar in architecture to other component subsystems inside of the QNX Neutrino operating system. At the bottom layer are drivers that provide the mechanism for passing data to, and receiving data from, the hardware. The drivers hook into a multi-threaded layer-2 component (that also provides fast forwarding and bridging capability) that ties them together and provides a unified interface into the layer-3 component, which then handles the individual IP and upper-layer protocol-processing components (TCP and UDP).

Figure 16: Architecture of io-pkt.



A resource manager forms a layer on top of the stack and acts as the message-passing intermediary between the stack and user applications. It provides a standardized type of interface involving *open()*, *read()*, *write()*, and *ioctl()* that uses a message stream to communicate with networking applications. Networking applications written by the user link with the socket library. The socket library converts the message-passing interface exposed by the stack into a standard BSD-style socket layer API, which is the standard for most networking code today.

In addition to the socket-level API, there are also other, programmatic interfaces into the stack that are provided for other protocols or filtering to occur.

At the driver layer, there are interfaces for Ethernet traffic (used by all Ethernet drivers), and an interface into the stack for 802.11 management frames from wireless drivers. The **hc** variants of the stack also include a separate hardware crypto API that allows the stack to use a crypto offload engine when it's encrypting or decrypting data for secure links.

The **io-pkt** component is the active executable within the network subsystem. Acting as a kind of packet redirector/multiplexer, **io-pkt** is responsible for loading protocol and driver modules based on the configuration given to it on its command line (or via the **mount** command after it's started). Employing a zero-copy architecture, the **io-pkt** executable efficiently loads multiple networking protocols, filters, or drivers (e.g. **lsm-qnet.so**, **lsm-autoip.so**) on the fly– these modules are shared objects that install into **io-pkt**.

The networking protocol module is responsible for implementing the details of a particular protocol (e.g. Qnet etc.). Each protocol component is packaged as a shared object (e.g. **lsm-qnet.so**). One or more protocol components may run concurrently. Once **io-pkt** is running, you can dynamically load drivers at the command line using the **mount** command. For example, these commands:

```
io-pkt-v6-hc &
mount -T io-pkt devnp-e1000.so
```

would start **io-pkt** and then mount the driver for an Intel Gigabit Ethernet controller. Once the shared object is loaded, **io-pkt** will then initialize it. The driver and **io-pkt** are then effectively bound together – the driver will call into **io-pkt** (for example when packets arrive from the interface) and **io-pkt** will call into the driver (for example when packets need to be sent from an application to the interface). You can also use the **ifconfig** command to unload a driver:

```
ifconfig wm0 destroy
```

The drivers can be written specifically for io-pkt, or you can port a NetBSD driver. There's even a "shim" layer that lets you use a driver that was written for the earlier networking stack, **io-net**.

3.3.2.1 The QNX Neutrino *sysctl()* interface

The *sysctl()* function retrieves system information and allows processes with appropriate privileges to set system information. The data available from *sysctl()* consists of integers and tables. You can also get or set data using the **sysctl** utility at the command line.

Here are some of the parameters that you can modify using the *sysctl()* interface under QNX Neutrino:

Stack parameters you can modify via *sysctl()*	**Description**
ip.forwarding	Returns 1 when IP forwarding is enabled for the host, meaning that the host is acting as a router.
ip.redirect	Returns 1 when ICMP redirects may be sent by the host. This option is ignored unless the host is routing IP packets. Normally, this option should be enabled on all systems.
ip.ttl	The maximum time-to-live (hop count) value for an IP packet sourced by the system. This value applies to normal transport protocols, not to ICMP.
ip.forwsrcrt	Returns 1 when the forwarding of source-routed packets is enabled for the host. This value may be changed only if the kernel security level is less than 1.
ip.directed-broadcast	Returns 1 if directed-broadcast behavior is enabled for the host.
ip.allowsrcrt	Returns 1 if the host accepts source-routed packets.
ip.subnetsarelocal	Returns 1 if subnets are to be considered local addresses.
ip.mtudisc	Returns 1 if path MTU discovery is enabled.
ip.maxfragpackets	Returns the maximum number of fragmented IP packets in the IP reassembly queue.
ip.sourcecheck	Returns 1 if source checking for received packets is enabled.
ip.sourcecheck_logint	Returns the time interval when IP source address verification messages are logged. Setting this to zero disables the logging.

Stack parameters you can modify via *sysctl()*	Description
icmp.maskrepl	Returns 1 if ICMP network mask requests are to be answered.
tcp.rfc1323	Returns 1 if RFC1323 extensions to TCP are enabled.
tcp.sendspace	Returns the default TCP send buffer size.
tcp.recvspace	Returns the default TCP receive buffer size.
tcp.mssdflt	Returns the default TCP maximum segment size.
tcp.syn_cache_limit	Returns the maximum number of entries allowed in the TCP compressed state engine.
tcp.syn_bucket_limit	Returns the maximum number of entries allowed per hash bucket in the TCP compressed state engine.
tcp.syn_cache_interval	Returns the TCP compressed state engine's timer interval.
udp.checksum	Returns 1 when UDP checksums are being computed and checked. ***Note**: Disabling UDP checksums is strongly discouraged.*
udp.sendspace	Returns the default UDP send buffer size.
udp.recvspace	Returns the default UDP receive buffer size.

3.3.3 Conclusions

- As discussed in the previous section, device drivers will have to be reimplemented to integrate correctly with the **io-pkt** infrastructure.

- You can use native **io-pkt** drivers, ported NetBSD drivers, or legacy **io-net** drivers.

- The application layer API in VxWorks is based on the BSD 4.4 implementation; the QNX Neutrino API is based on NetBSD 4.0. As such, most application-layer code should be ported from VxWorks to QNX Neutrino in a straightforward manner. Small changes (e.g. device names) will be required to complete the port.

- Control and configuration of the stack using **ioctl** commands is similar in the two operating systems. The degree of similarity depends in some part on how the underlying device drivers have been implemented. The behavior offered by some of the VxWorks API calls can be obtained under QNX Neutrino using the **ioctl** interface as follows.

VxWorks API:	**QNX Neutrino *ioctl()* equivalent:**
ifAddrAdd()	`ioctl(SIOCAIFADDR,  struct ifaliasreq *req)`
ifAddrSet()	`ioctl(SIOCSIFADDR,  struct ifreq *req)`
ifAddrGet()	`ioctl(SIOCGIFADDR, struct ifreq *req)`
ifBroadcastSet()	`ioctl(SIOCSIFBRDADDR, struct ifreq *req)`
ifBroadcastGet()	`ioctl(SIOCGIFBRDADDR, struct ifreq *req)`
ifDstAddrSet()	`ioctl(SIOCSIFDSTADDR, struct ifreq *req)`
ifDstAddrGet()	`ioctl(SIOCGIFDSTADDR, struct ifreq *req)`
ifMaskSet()	`ioctl(SIOCSIFNETMASK, struct ifreq *req)`
ifMaskGet()	`ioctl(SIOCGIFNETMASK, struct ifreq *req)`
ifFlagSet()	`ioctl(SIOCSIFFLAGS, struct ifreq *req)`
ifFlagGet()	`ioctl(SIOCGIFFLAGS, struct ifreq *req)`
ifMetricSet()	`ioctl(SIOCSIFMETRIC, struct ifreq *req)`
ifMetricSet()	`ioctl(SIOCGIFMETRIC, struct ifreq *req)`

- Altering certain aspects of the stack behavior (keep-alive timers, ARP timeout, etc.) differs in the two systems. In VxWorks, you modify global variables; under QNX Neutrino, you can use the *sysctl()* interface to modify several of these parameters

3.3.4 Network application programming

3.3.4.1 Protocol and RFC support.

QNX Neutrino provides support for several networking protocols and RFCs, including (but not necessarily limited to) the following:

TCP/IP Stack RFC support (specific to io-pkt-v6-hc)	**Title**
2367	PF_KEY Key Management API, Version 2
1826; 2402	IP Authentication Header
2403	The Use of HMAC-MD5-96 within ESP and AH
2404	The Use of HMAC-SHA-1-96 within ESP and AH

TCP/IP Stack RFC support (specific to io-pkt-v6-hc)	**Title**
2405	The ESP DES-CBC Cipher Algorithm With Explicit IV
2406	IP Encapsulating Security Payload (ESP)
2292	Advanced Sockets API for IPv6
2553	Basic Socket Interface Extensions for IPv6
2463	Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification
2460	Internet Protocol, Version 6 (IPv6) Specification
2461	Neighbor Discovery for IP Version 6 (IPv6)
2462	IPv6 Stateless Address Autoconfiguration
2451	The ESP CBC-Mode Cipher Algorithms
1981	Path MTU Discovery for IP version 6
2373	IP Version 6 Addressing Architecture
2144	The CAST-128 Encryption Algorithm
2401	Security Architecture for the Internet Protocol
2960	Stream Control Transmission Protocol (SCTP)
2526	Reserved IPv6 Subnet Anycast Addresses
2374	An IPv6 Aggregatable Global Unicast Address Format
2375	IPv6 Multicast Address Assignments
2473	Generic Packet Tunneling in IPv6 Specification
2464	Transmission of IPv6 Packets over Ethernet Networks
2893 (portions of)	Transition Mechanisms for IPv6 Hosts and Routers
1701	Generic Routing Encapsulation (GRE)
1702	Generic Routing Encapsulation over IPv4 networks

The following RFCs are supported by all three variants of **io-pkt**:

TCP/IP Stack RFC support (io-pkt-v4, io-pkt-v4-hc, and io-pkt-v6-hc)	**Title**
791	INTERNET PROTOCOL - DARPA INTERNET PROGRAM - PROTOCOL SPECIFICATION
792	INTERNET CONTROL MESSAGE PROTOCOL -

	DARPA INTERNET PROGRAM – PROTOCOL SPECIFICATION
793	Transmission Control Protocol (TCP)
768	User Datagram Protocol
1122	Requirements for Internet Hosts – Communication Layers
2001	TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms
1112	Host Extensions for IP Multicasting
1323	TCP Extensions for High Performance

The following RFCs are supported for networking applications:

Application RFC support	**Title**
1123	Requirements for Internet Hosts – Application and Support
1700	Assigned Numbers
951	BOOTSTRAP PROTOCOL (BOOTP)
2409	The Internet Key Exchange (IKE)
1542	Clarifications and Extensions for the Bootstrap Protocol
1048; 1084	BOOTP Vendor Information Extensions
854	TELNET PROTOCOL SPECIFICATION
1408; 1572	Telnet Environment Option
959	FILE TRANSFER PROTOCOL (FTP)
2389	Feature negotiation mechanism for the File Transfer Protocol
2428	FTP Extensions for IPv6 and NATs
2732	Format for Literal IPv6 Addresses in URL's
draft-ietf-ftpext-mlst-16.txt	Extensions to FTP
1350	THE TFTP PROTOCOL (REVISION 2)
1035	DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION
1058	Routing Information Protocol
1723	RIP Version 2 Carrying Additional Information

Application RFC support	Title
2080	RIPng for IPv6
1256	ICMP Router Discovery Messages
1094	NFS: Network File System Protocol Specification
1813	NFS Version 3 Protocol Specification
1831	RPC: Remote Procedure Call Protocol Specification Version 2
1832	XDR: External Data Representation Standard
1833	Binding Protocols for ONC RPC Version 2
2131	Dynamic Host Configuration Protocol
2132	DHCP Options and BOOTP Vendor Extensions
882	DOMAIN NAMES - CONCEPTS and FACILITIES
883	DOMAIN NAMES - IMPLEMENTATION and SPECIFICATION
931	Authentication Server
973	Domain System Changes and Observations
974	MAIL ROUTING AND THE DOMAIN SYSTEM
1033	DOMAIN ADMINISTRATORS OPERATIONS GUIDE
1034	DOMAIN NAMES - CONCEPTS AND FACILITIES
1035	DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION
1075	Distance Vector Multicast Routing Protocol
1144	Compressing TCP/IP Headers for Low-Speed Serial Links
1321	The MD5 Message-Digest Algorithm
1332	The PPP Internet Protocol Control Protocol (IPCP)
1334	PPP Authentication Protocols
1549	PPP in HDLC Framing
1661	The Point-to-Point Protocol (PPP)
1662	PPP in HDLC-like Framing
1962	The PPP Compression Control Protocol (CCP)
1990	The PPP Multilink Protocol (MP)
2068	Hypertext Transfer Protocol - HTTP/1.1

Application RFC support	Title
1305	Network Time Protocol (Version 3) Specification, Implementation and Analysis
2030 (obsoletes RFC 1769, which obsoletes RFC 1361)	Simple Network Time Protocol (SNTP) version 4 for IPv4, IPv6 and OSI
1119	Network Time Protocol (version 2) - specification and implementation

Note that while **io-pkt-v6-hc** supports all of the above protocols, **io-pkt-v4** and **io-pkt-v4-hc** support a subset:

- IP, UDP, TCP, ICMP, ARP, IPv6, IPSec, SCTP, ICMPv6, IGMPv2
- IP, UDP, TCP, ICMP, ARP, IPv6, IPSec, SCTP, ICMPv6, IGMPv2
- Unix Domain Sockets
- PROXY ARP (PPP interfaces only)
- VPN - GRE (Generic Routing Encapsulation):
  - GIF (Tunnel IPv4 IPv6 eg IPv6 over IPv4)
  - VLAN (IEEE 802.1Q Virtual LAN)
- DLL Support
- PPP, PPPOE, SCTP, NAT/IPfilter, QNET, AutoIP
- Application Support
- BOOTP (server only)
- DHCP (server, client, and relay agent)
- SNMP v1, v2p (port of CMU source base)
- SRI SNMP v1, v2c, v3 (SNMP Research port)
- HTTP1.1/CGI1.1/SSI (small server only)
- NFS v2, v3 (server and client)
- PCNFSD
- CIFS/SMB (client only)
- FTP (client and server)
- Telnet (client and server)
- TFTP (client and server)
- DVMRP
- RIP v1, v2, RIPng
- PPP/PAP/CHAP

- PPPOE (client only)
- NTP v2, v3, v4
- SOCKS
- LPR
- RLOGIN
- IKE (See the "racoon" binary documentation for availability)

3.3.4.2 Socket Programming

The socket layer API used by VxWorks is based on BSD 4.4; that in QNX Neutrino is based on NetBSD 4.0. Differences in the code base used between the two libraries may necessitate some code changes to facilitate the port, but these shouldn't be extensive.

Under QNX Neutrino, the API for the TCP/IP stack includes the BSD Socket API, routing socket, *sysctl()*, *kvm*()*, and *PF_KEY* socket, which facilitates porting. Integration varies from protocol to protocol. Many protocols will use the interfaces described above, which are very portable.

3.3.4.3 VxWorks

VxWorks includes a variety of programmatic APIs for directly interfacing to networking utilities from an application.

A number of modules providing networking utilities are available within VxWorks. These include: DHCP, PPP, ProxyARP, FTP, TFTP and a Telnet server.

3.3.4.4 QNX Neutrino

QNX Neutrino provides implementations of all of the above utilities (and also a Telnet client application). However, theses are usually provided as applications (i.e. binary images) that aren't directly callable from a source application. Standard Unix mechanisms are used to invoke the applications from the command line.

Source code for all of the utilities is also available if the utility needs to be embedded in the application. This isn't usually necessary, since other simpler mechanisms are available for invoking the required utility.

Incidentally, with QNX Neutrino's full-featured Telnet server and fully reentrant shell, multiple clients may connect into a target simultaneously.

3.3.4.5 Conclusions

- VxWorks provides a programmatic interface to networking utilities.

- QNX Neutrino provides binary executables to the networking utilities. These are normally invoked by scripts from a command shell (i.e. standard Unix mechanisms). Source code for utilities is available for incorporation into application code if required.

- QNX Neutrino includes additional utilities not available with VxWorks.

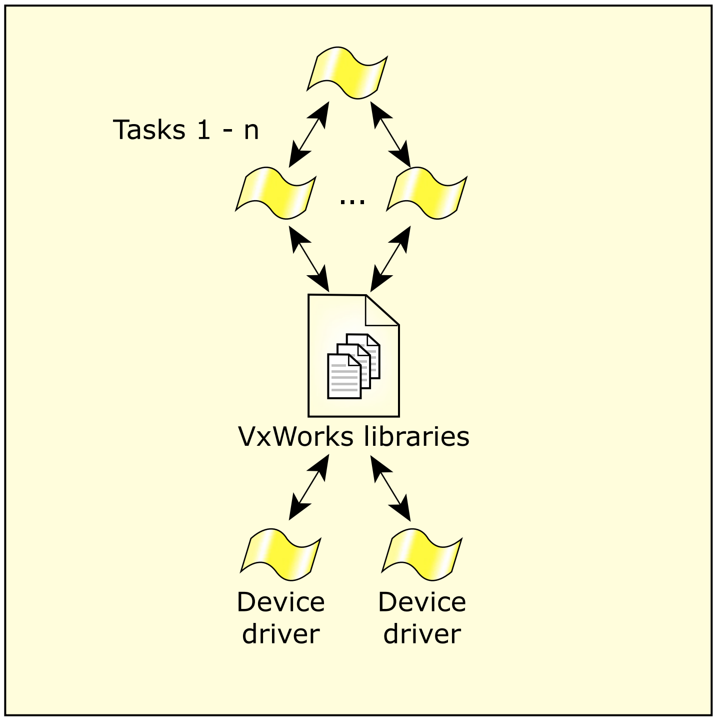
3.4 Application porting

The "application" layer referred to in this section is the portion of the application that is, to a large degree, *hardware-agnostic*. As stated previously, applications that run under both VxSim (a product often used extensively during development) and target platforms should be able to port directly to QNX Neutrino via the porting library.

3.4.1 Overview of *vx2qnx.lib*

In order to ease the initial porting effort of legacy code from VxWorks to QNX Neutrino, a porting library is provided as an intermediate "glue" between existing VxWorks-based application code and underlying QNX Neutrino primitives.

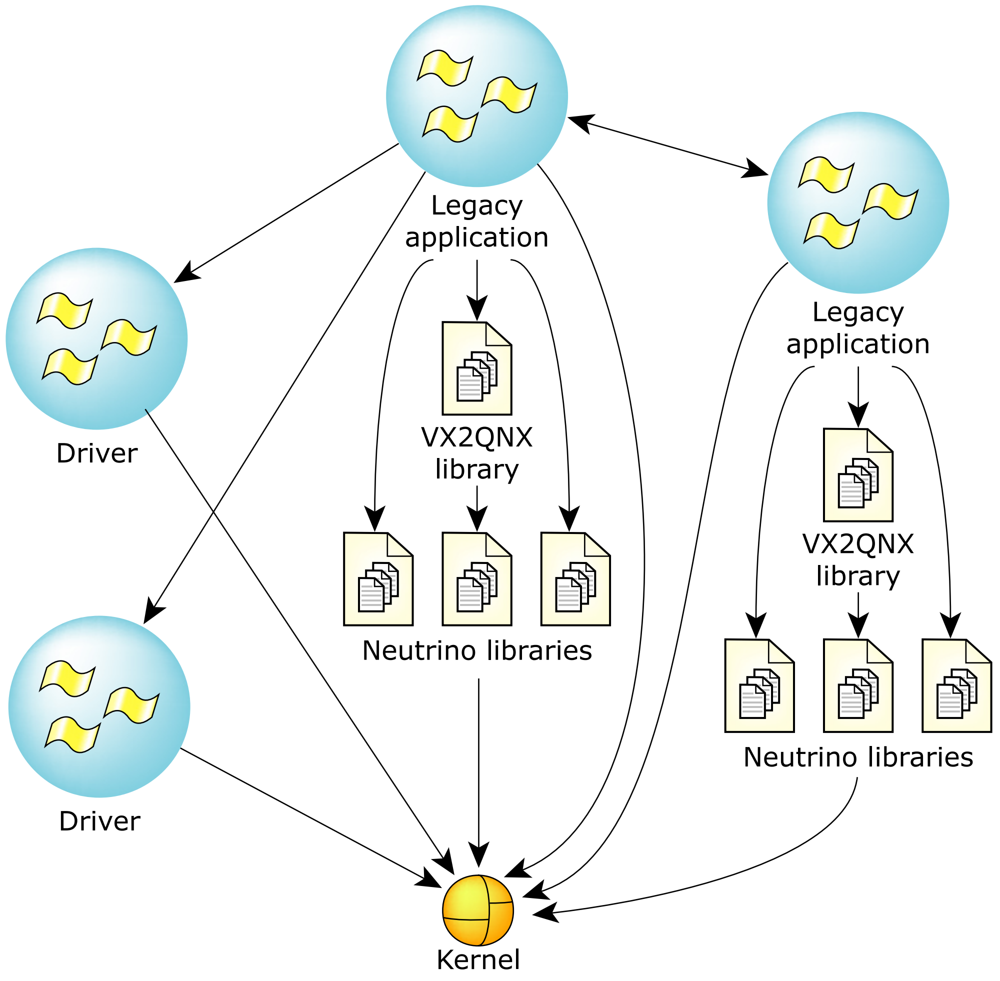
Figure 17: VxWorks legacy application.



The `vx2qnx` porting library contains an implementation of several commonly used VxWorks API functions using underlying QNX Neutrino primitives. This allows for code compatibility with existing legacy code that relies on the VxWorks

API at the application layer. You can find the `vx2qnx` library in the Wind River VxWorks Migration project on our Foundry27 website, http://community.qnx.com.

Figure 18: VxWorks legacy applications ported to QNX Neutrino.



The coverage of the library includes task operations, semaphores, message queues, watchdog timers, list-manipulation routines etc. All of the above functions are implemented via QNX Neutrino primitives to provide functionality equivalent to that provided in VxWorks. The result is code-compatibility with legacy code that would otherwise take a considerable amount of effort to rework. The `vx2qnx` library implementation is supplied in source-code form.

> ***Note**: A more direct, but more elaborate, approach (as described below) of converting calls to functions in VX libraries by mapping them functionally onto underlying QNX Neutrino primitives is a better alternative. This approach can ensure that the code is more closely implemented using native mechanisms under QNX Neutrino.*

The porting layer is provided to both minimize an initial porting effort and also to demonstrate how specific VxWorks functionality can be implemented using native QNX Neutrino primitives. Typically, you should expect to undergo a more elaborate porting exercise that isn't limited only to code and API compatibility, but also comprises specific architectural design modifications in order to take full advantage of several of the intrinsic benefits of the QNX Neutrino architecture.

The `vx2qnx` porting library provides the interface code to implement in QNX Neutrino a similar API call made under VxWorks. In situations where the behavior doesn't have a direct equivalent provided by the underlying primitives, it is implemented in the library itself. This may lead to a performance penalty, since there's an additional layer in between the application making the call and the OS implementing the call.

Using the `vx2qnx` porting library, the complete VxWorks system is implemented as a single QNX Neutrino process: Each task in VxWorks is mapped to a distinct thread in QNX Neutrino, but all threads are in the one process. The negative aspect of this model is that the system doesn't take full advantage of the memory protection between processes under QNX Neutrino. (For information on the behavioral differences between the QNX Neutrino and VxWorks implementations, see section 5.2 in the Appendix in this document.)

The following API is implemented within the porting library:

VxWorks library	**Functions**
taskLib	*taskSpawn(), taskInit(), taskActivate(), exit(), taskDelete(), taskDeleteForce(), taskSuspend(), taskResume(), taskRestart(), taskPrioritySet(), taskPriorityGet(), taskLock(), taskUnlock(), taskSafe(), taskUnsafe(), taskDelay(), taskIdSelf(), taskIdVerify(), taskTcb()*
msgQLib	*msgQLibInit(), msgQCreate(), msgQDelete(), msgQSend(), msgQReceive(), msgQNumMsgs()*
semLib	*semLibInit(), semGive(), semTake(), semFlush(), semDelete()*
semMLib	*semMLibInit(), semMCreate()*
semCLib	*semCLibInit(), semCCreate()*
semBLib	*semBLibInit(), semBCreate()*
wdLib	*wdCreate(), wdDelete(), wdStart(), wdCancel()*
errnoLib	*errnoGet(), errnoOfTaskGet(), errnoSet(), errnoOfTaskSet()*
taskInfoLib	*taskName(), taskNametoId(), taskIdDefault(), taskIsReady(), taskIsSuspended(), taskIdListGet()*
kernelLib	*kernelInit(), kernelTimeSlice(), kernelVersion()*
lstLib	*lstInit(), lstAdd(), lstConcat(), lstCount(), lstDelete(), lstExtract,lstFirst(), lstGet(), lstInsert(), lstLast(), lstNext(), lstNth(), lstPrevious(), lstNStep(), lstFind(), lstFree()*

QNX Neutrino already supports the following VxWorks libraries via the conforming implementation of equivalent POSIX/Unix calls:

VxWorks library	POSIX/Unix functions
schedPxLib	*sched_setparam(), sched_getparam(), sched_setscheduler, sched_getscheduler(), sched_yield(), sched_get_priority_max, sched_get_priority_min(), sched_rr_get_interval()*
mqPxLib	*mq_open(), mq_receive(), mq_send(), mq_close(), mq_unlink(), mq_notify(), mq_setattr(), mq_getattr()*
clockLib	*clock_getres(), clock_setres(), clock_gettime(), clock_settime()*
semPxLib	*sem_init(), sem_destroy(), sem_open(), sem_close(), sem_unlink(), sem_wait(), sem_trywait(), sem_post(), sem_getvalue()*
sigLib	*sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember(), signal(), sigaction(), sigprocmask(), sigpending(), sigsuspend(), pause(), sigtimedwait(), sigwaitinfo(), sigsetmask(), sigblock(), raise(), kill(), sigqueue()*
timerLib	*timer_cancel(), timer_create(), timer_delete(), timer_gettime(), timer_getoverrun(), timer_settime(), nanosleep()*

4 BUILD ENVIRONMENT

Setting up a build environment to produce build images can involve a considerable amount of effort. Fortunately, both VxWorks and QNX Neutrino can use the GNU tool chain, so you can reuse elements of the VxWorks build environment with QNX Neutrino.

This section discusses some of the potential issues involved in moving the build environment from VxWorks to QNX Neutrino.

4.1 Makefiles

The level of effort required to change the build environment from one system to another obviously depends on the complexity of the build process. For smaller builds, the easiest approach will likely be to rewrite the makefiles that go into the build. QNX Neutrino provides a complete set of makefiles that can be used (with suitable modification) to recursively build a directory tree.

The QNX Momentics integrated development environment (similar to Tornado) can also be used to auto-generate and manage the make scripts for smaller projects.

For large, complex builds, reuse of portions of the existing build scripts may be possible if the scripts have been written in such a manner as to abstract the tool chain. Within VxWorks make files, macros are normally used to define the appropriate command lines for archiving, compiling, linking, etc. After including the appropriate QNX Neutrino **.mk** files, these macros can be replaced with the corresponding QNX Neutrino commands.

Build scripts may also have to be modified to account for differences in the versions of **make** (although these are likely to be few) and support utilities. It's quite common, for example, for build processes on Microsoft Windows platforms to include the **cygwin** set of tools for building (VxWorks uses **cygwin)** This provides a Unix-like command set for MS-Windows that can be used to maintain consistency across different build platforms. The **cygwin** utilities depend on a shared library (**cygwin1.dll**) for operation. QNX Neutrino uses MinGW (Minimalist GNU for Windows) and MSYS (Minimal SYStem).

> ***Note**: Care has to be taken to ensure that the correct **cygwin1.dll** location is included in the PATH variable to prevent versioning conflicts.*

4.2 C/C++ Compiler

Both operating systems use the GNU C/C++ compiler as their standard compiler. Version 5.4 of VxWorks uses version 2.7.2, and 5.5 uses 2.96+ (essentially version 2.95 with some Wind River add-ons). QNX Neutrino 6.3 uses version 3.3 (with version 2.95 also supported), and QNX Neutrino 6.4.1 uses version 4.3.

Users of either the VxWorks 5.4 tool chain can expect compiler issues to appear as they pass their code through the compiler used by the QNX Momentics tool suite. In particular, much stricter type-checking rules have been implemented in the GNU compiler. This manifests itself in the need to provide exact typecasts to function parameters that were acceptable to the 2.7.2 compiler.

For the most part, the compiler issues encountered in porting to QNX Neutrino should be identical to those encountered when porting from VxWorks 5.4 to 5.5, given that the compiler chain for VxWorks 5.5 and QNX Neutrino comes from the same base.

Slight variations in the compiler chain provided by QNX Neutrino and VxWorks 5.5 may also result in code changes being required to obtain a clean compile.

VxWorks environments that use the Diab tool chain can also expect code changes to be required as a result of compiler differences.

Although many of the standard libraries have fixed APIs, others (the C++ Standard Template Library comes to mind) have evolved over time. With VxWorks using older versions of the libraries, code changes may have to be made to accommodate the newer libraries. Note that the QNX Momentics tool suite includes two versions of the STL: GCC and Dinkum. The Dinkum library is a "clean room" implementation of STL by Dinkumware, Ltd. (www.dinkumware.com) that has no GPL issues associated with it.

4.3 Linker

Within VxWorks, a number of link products can be introduced during the development phase. These include partially linked "downloadable" modules that are downloaded and dynamically linked into a running application, as well as full application images (which include the operating system). The build process in VxWorks often consists of compiling to create object modules, partial linking of the object files to produce a module, and then executing a variety of steps (e.g. "munching" to retrieve constructor/destructor information) to produce a downloadable module that is used for unit-testing.

For final product, the modules are in turn "partially" linked together into a "partial image" object file that includes the OS libraries. This object file is parsed for a variety of other information (constructor/destructor, symbol table, etc.) to include in the final link stage. The final link stage results in the complete application image.

With QNX Neutrino, the end result of the application build stage is an *application* image rather than an OS image. As such, the link stage involves taking the compiled object files (**.o**) and linking them together with the OS libraries to form the application image (i.e. the same as a "desktop" environment). This image can then be uploaded into the target and executed using shell commands or scripts. In a self-hosted environment, the user can execute an x86 target application immediately from the command line.

Unlike VxWorks, there is no need to "munch" a partial C++ image file to retrieve constructor/destructor information (since the static constructor/destructor setup is done during the startup of the application process as opposed to the startup of the operating system). The fact that a complete executable image is formed for all individual processes means that developers should take advantage of the dynamic linking capability of code that can be shared between processes in order to reduce memory usage. Such considerations are discussed in the next section.

Once all the application images have been created, they can either be included in the OS image (using the image filesystem) or stored as separate entities on other filesystems for invocation at runtime. Command execution at the shell uses searches of the mounted filesystem (determined by the PATH environment) to find the indicated image for execution (i.e. again the same as a desktop system).

One final note: Each separate QNX Neutrino process must have a *main()* function added to indicate to the linker where the application starts running from. The *main()* function initializes the application and starts up the threads in the appropriate manner.

4.4 Application memory usage

Within VxWorks 5.4/5.5, given that the full address space is visible to all tasks, "code sharing" is implemented transparently (with obvious care having to be taken in the code design to deal with reentrancy issues). This allows several threads to share the same code that's in memory. Within a single process in QNX Neutrino, this ability also naturally exists. However, between multiple processes in QNX Neutrino, the memory isolation/virtual memory implementation prevents this from happening with "normal" application code (i.e. it isn't possible to pass the address of a function between two processes and have the second process run that function). Using a normal link process, objects are statically linked together into a final image. If multiple processes require the same code, then each image contains its own copy of the code.

This has obvious implications for memory utilization (whether for storage of the image in flash or storage of the executable in RAM on execution). A software library can be shared by turning it into either a shared object or a dynamically linked library. This allows multiple processes to dynamically link into the library, thereby reducing memory requirements. Shared libraries can also be used to allow for dynamic upgrades of an application. If the multiprocess scheme is used to port an application from VxWorks to QNX Neutrino, the code base should be examined to determine if portions of it should be aggregated into a shared library.

5 APPENDIX

5.1 Porting reference

The following table gives an approximate correspondence in terms of functionality for libraries in VxWorks to libraries/calls in QNX Neutrino:

VxWorks	QNX Neutrino/POSIX
taskLib	*pthread_*, posix_spawn_*, fork*
semCLib, semLib, semMLib, semBLib, semPXLib	*sem_*, pthread_mutex_*, pthread_cond_*, pthread_sleepon_*, pthread_barrier_*, pthread_rwlock_**
msgQLib, msgPXLib	*mq_*, MsgSend/MsgReceive, MsgSendPulse*
taskVarLib	*pthread_key_**
sigLib	*sig**
sockLib	**libsocket**
wdLib, timerLib	*timer_*, SIGEV_**
schedPxLib	*sched_*, pthread_*;* **Sporadic Scheduling Policy**.
ansiAssert, ansiCtype, ansiLocale, ansiMath, ansiSetjmp, ansiStdarg, ansiStdio, ansiStdlib, ansiString, ansiTime	Mostly in **libc** (standard Unix/POSIX)
clockLib	*clock_** (**libc**)
dirLib	Mostly in **libc** (standard Unix/POSIX)
fioLib	Mostly in **libc** (standard Unix/POSIX)
hostLib	Mostly in **libsocket**
ioLib	Mostly in **libc** (standard Unix/POSIX)
resolveLib	Mostly in **libsocket**
memPartLib	*malloc, free, shm_*, mmap/munmap*

5.2 Behavioral differences

5.2.1

- **taskSwitchHook**

Existing *taskVariables* calls can map to *pthread_key_*()* calls, but the VxWorks task variables can directly access/modify the variables in each task context without going through the task var interface.

The *pthread_key_*()* interface, on the other hand, assumes that all declared "thread-specific" variables are accessed and modified only using the *pthread_*()*-specific calls.

- **Changing the timeslice for RR scheduling**

VxWorks provides for a mechanism to modify the global time slice that is used for round robin scheduled tasks. QNX Neutrino does not have an exactly equivalent call, but it provides for a "sporadic scheduling" policy that can be used instead to control the execution budget of a thread in a fine-grained fashion.

- **File Descriptor Sharing**

Stdin, stdout, and stderr are task-specific, while all other FDs are system-wide. In QNX Neutrino, *all* FDs are process-specific, so a mapping from VxWorks tasks to QNX Neutrino threads in a process would lose this behavior. On the other hand, a mapping from VxWorks tasks to QNX Neutrino processes would lose the shared address space property enjoyed by threads within a process.

- **FIFO-order wakeup**

VxWorks works provides the ability for tasks to be woken up in FIFO order when a resource that they are pending on (such as a semaphore or a message queue) becomes available. In QNX Neutrino, the basic synchronization objects don't provide an equivalent wakeup mechanism, although you can use native QNX Neutrino primitives to simulate this type of behavior.

- **timer_connect**

The *timer_connect* function is in the VxWorks **timerLib** (i.e. POSIX timers), but is not actually a POSIX function.

- **Priority levels**

As of 6.3.0, QNX Neutrino supports 255 distinct priority levels. For versions prior to 6.3.0, the 255 priority levels in VxWorks need to be "squashed" to the available 63 distinct priority levels. As long as the applications don't actually require more than 63 distinct priority levels, this shouldn't be a problem.

- **Inversion-Safe Semaphores in VxWorks**

If a task is raised in priority, it isn't lowered until *ALL* inversion-safe semaphores that it currently owns are released. From the VxWorks documentation:

> "Because the priority of a task which has been elevated by the taking of a mutual-exclusion semaphore remains at the higher priority until all mutexes held by that task are released, unbounded priority inversion situations can result when nested mutexes are involved."

Under QNX Neutrino at any time, the priority of a given thread holding a mutex is the highest of the priorities of any other thread that is blocked on any mutex held by the original thread. This priority is updated upon any events associated with mutexes, and hence is protected from the unbounded priority inversion.

6 REFERENCES

QNX Neutrino sources:

System Architecture
Programmer's Guide
Building Embedded Systems
Library Reference

Note: All of the above documentation is available on the QNX Software Systems website (http://www.qnx.com/developer/docs).

VxWorks sources:

Wind River VxWorks
(http://www.windriver.com/products/vxworks/)

VxWorks/Tornado II FAQ
(http://www.xs4all.nl/~borkhuis/vxworks/vxworks.html)

VxWorks Cookbook
(http://www.bluedonkey.org/cgi-bin/twiki/bin/view/Books/VxWorksCookBook)

QNX SOFTWARE SYSTEMS

About QNX Software Systems

QNX Software Systems, a Harman International company, is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® development suite, and QNX Aviage® middleware family together provide the industry's most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for network routers, medical instruments, vehicle telematics units, security and defense systems, industrial robotics, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com