Porting VxWorks® Applications to Linux
Introduction

Moving from a proprietary operating system such as VxWorks® to Linux can create a strain on the time and resources companies have at their disposal. Depending on the approach to this exercise the level of effort required can be significant and may require an in-depth Linux expertise. This application note identifies different approaches that are available to companies today, providing a good background for decision making process. This application note does not go into details such as tool chains, compilers, etc.

VxWorks vs. Linux architecture differences

Architecture of a Legacy RTOS

The basic architecture of an RTOS-based application consists of application code that is made up of one or more tasks. These tasks are statically linked with the runtime libraries and the kernel itself. All these entities reside and execute in a single physical memory address space, which enables the application and kernel to share global data, data structures, header files etc. It also means that interrupt handling routines can be part of the application code and that application code can directly call device driver functions and/or access the hardware directly.

This architecture allows for maximum coding flexibility and runtime performance but renders the system highly exposed to failure due to data corruption. Any task in the system can corrupt the data of any other tasks and indeed the data of the kernel itself. Furthermore, when such corruption occurs it is very difficult to determine the source of the problem. Many times failure manifests itself within a different task than the one that failed. This scheme is depicted in the following diagram:

![Figure 1: Traditional RTOS Architecture](image)

Linux architecture

The architecture of the UNIX-derived Linux OS is different. In Linux, a process defines its own virtual memory address space and threads within that process all execute within that space. Application processes do not have access to the kernel address space nor are they allowed to directly access the hardware. Access to the underlying hardware is controlled exclusively by the Linux kernel. The kernel provides application processes with access to the hardware through the use of device drivers and system calls. In addition, since each process runs in its own unique name space, processes are prevented from directly accessing the address space of other processes. The virtual memory scheme is implemented by using a hardware-based memory management unit (MMU).
A process may be composed of many execution threads. These threads all share the address space of the owning process. Threads and process will be discussed in more detail later in this paper. The following diagram depicts the Linux environment:

![Linux Architecture Diagram]

Porting Process

This section discusses the major issues involved in porting an application and outlines a recommended process for performing the port. The section begins by describing the parts of the application that need to be ported and how to identify them. After those parts have been identified, porting options will be covered.

Identifying the components that need porting

A system’s software components can belong to one of the following conceptual groups:

1. Application tasks — Implement the application’s logic
2. Drivers — Provide access to the underlying hardware
3. Kernel — Provides the scheduler and other basic OS services

As discussed in the previous section, in a traditional RTOS this distinction can often be blurred. Since the kernel is linked statically with the other application components, and since they all execute in the same address space, the system’s functionality can be easily spread across any of the components.

Interaction between these components does not have to adhere to any specific interface. In Linux on the other hand, the kernel and the device driver components form a completely separate entity from that of the application tasks. Moreover, communication between these different entities must be done via a clearly defined interface.

Another issue that needs to be addressed is that of functions which implement functionality that is used by different tasks. In a statically linked single address space system these functions are assigned a unique global address and subsequently can be accessed by any task. Of course, if these functions use any global data structures (i.e., they are not reentrant), then that data needs to be protected by some mutual exclusion scheme. Several options for porting these functions to Linux will be discussed.

A critical phase of the porting process is therefore to identify the following elements:

- Application tasks
- Device drivers
Porting Application Tasks

RTOS tasks are usually composed of a main loop that calls some C functions and/or kernel system calls. The existing tasks can be mapped to either a Linux process or a Linux thread. As far as the Linux scheduler is concerned, threads and processes are equal. However, there are other important differences:

- **Performance and resource costs**
  In general, threads have a lower context switch overhead than processes. (On some architectures, notably ARM, the difference is considerable.) This overhead is accrued during creation and more importantly during context switches. If performance is critical, then threads are preferred.

- **Different API**
  The available APIs for manipulating threads and processes is different. The pthread library provides the support for Linux threads. As the Linux scheduler treats both processes and threads equally, every relevant system call that can be made for the process can also be made for a thread (except of course the ones for creation and deletion). However, Linux threads may additionally benefit by using mutexes and condition variables that are currently only available for threads.

- **Robustness**
  Any thread can crash all the other threads with which it shares its address space. A process, on the other hand, can do no harm except to itself. Execution robustness in critical processes may be worth the additional cost associated with their use. Another option is to use processes during the development phases, which help identify and resolve bugs. When the correctness of the application’s logic is proven, a performance-tuning phase may be performed that includes changing processes into threads.

- **Inter Process Communication**
  Another important issue to consider is the communication between the different processes. Unlike RTOS tasks, Linux processes must use some kind of Inter Process Communication (IPC) to communicate with each other. Since Linux threads share a common address space, they don’t require any special method to share data (but they still need to protect the integrity of the data). Standard IPC methods are costlier in terms of performance than the more direct communication possible within the same address space.

The following section describes the available Linux mechanisms.

- **Pipes** — Pipes are the simplest way of passing data between two different processes. However, pipes can only be used between processes that are related (i.e., processes that were created from a common ancestor process). RTOS tasks possess this kind of relationship, so this form of IPC is not likely to be used when migrating from an RTOS.

- **Named Pipes/FIFOs** — Named pipes or FIFOs solve the limitation imposed by standard process pipes. Named pipes use the filesystem to create a special file. Since all processes have access to the filesystem, they can connect through a named pipe even when they don’t have a common ancestor. VxWorks provides a similar interface so if the existing VxWorks application uses pipes, the porting is straightforward.

- **Messages** — Messages are like names pipes in that they allow unrelated processes to exchange data. Unlike pipes, however, messages allow for different types of messages with different priorities. VxWorks message queues are similar but not identical to their Linux counterparts.

- **Signals** — Signals are a means of asynchronously altering the execution flow of a process. Like Linux, VxWorks supports both UNIX BSD style signals and POSIX compatible signals to facilitate easy porting.
• **Shared Memory** — Shared memory is a mechanism for giving unrelated processes access to the same logical memory. Because tasks in VxWorks all run in a single address space, sharing data between these tasks is a trivial matter. In cases where the existing VxWorks tasks make extensive use of this fact for exchanging data, using shared memory may be the most suitable porting option. Of course, the tasks can be implemented as threads, achieving the same functionality.

• **Semaphore** — Semaphores are the primary means of ensuring mutual exclusion and process synchronization. VxWorks provides both optimized VxWorks semaphores and POSIX semaphores. The VxWorks version of the semaphores offers additional features such as priority inheritance, task-deletion safety, semaphore timeouts, and others. If the existing tasks use those extra features the porting may not be trivial.

• **Mutexes and Condition Variables** — In Linux, mutexes and condition variables are provided by the pthread library. This means they are not available for use by processes. While a mutex is just a specialized instance of a semaphore (count = 1), the mutex API provided by the pthread library is much easier to use. Similarly, most of the functionality provided by a condition variable can be achieved by using a semaphore as well.

**Porting Device Drivers**

Both VxWorks and Linux provide an almost identical interface for accessing devices. A typical application initializes a device and then periodically reads and or writes to it. Occasionally due to some stimulus, the device needs to be reconfigured, restarted, or shut down.

The initialization is typically performed by the open function. Reads and writes are performed by the read and write calls respectively. All other calls are implemented by the ioctl function.

After identifying all the code segments that access the hardware, these calls must be grouped into sub-groups according to the devices they access. The result is a list of all the devices that are accessed by the application. For each device, the following question must be considered:

Is there an existing Linux driver for our device?

The possible answers are:

1. There is an existing Linux device driver for this device.
2. This is a custom device for which there is no existing Linux device driver.

1. **Standard device**

Although VxWorks and Linux provide a similar device driver interface, they differ in the way they enforce the application to adhere to it. In Linux all hardware access must be funneled through a device driver. On the other hand, in VxWorks an application can manipulate the device by writing commands directly to the device’s registers. An application might access a device using some combination of standard device driver calls and direct calls.

The porting effort therefore consists of recoding the direct hardware access calls to use a standard interface function. If the existing Linux driver does not implement some feature that is required by the VxWorks application, that feature could be added as an option to the ioctl function of the driver.

2. **Custom device**

This section discusses the optimal approach for porting the VxWorks driver to Linux. Conceptually a device driver needs to read and/or write data to or from a device and it needs to be able to convey this data to the application.
The device driver may or may not use interrupts to perform this task. In addition, the device driver may be accessed through one or more of the application’s tasks.

The first issue that must be considered is where the new Linux driver should reside. Typically, a Linux device driver resides completely in kernel space. However, for some devices it is possible to write a device driver that resides completely in user space. Such a driver does not require writing any kernel code, something that requires specific Linux knowledge. It may also improve performance as it saves the overhead associated with kernel-user space context switches.

The key to user space drivers in Linux is the mmap command that allows for the mapping of a device’s address space to the address space of a Linux process. When the process writes to the mapped memory it actually writes to the device. A classical example for such a driver is a driver for a simple graphics display.

To determine if the existing VxWorks driver can be ported to a user space Linux driver, the following questions must be considered:

1. Does the driver use interrupts?
2. Is it accessed by more than one user space application task?

If the answer to any of the above questions is yes, the driver cannot be implemented as a user space driver; a kernel space driver must be created.

Most existing VxWorks driver code can be leveraged to the new Linux driver. However, several issues should be considered. The next section discusses the way in which the data is communicated between the driver and the application.

Communicating data between the driver and application

Although VxWorks and Linux provide an almost identical conceptual I/O interface, the actual implementation can be fundamentally different. The two major differences are:

1. Ability to connect application code to interrupts
2. Functions that can be called from within the driver

In VxWorks it is possible to connect an application C function to a hardware interrupt. The function is registered with the kernel, which adds some code that takes care of saving and restoring the necessary registers and setup of a special stack. Although the connected functions have restrictions on the functions they call, they can still make use of a wide range of facilities. For example, consider a simple sensor device that generates an interrupt when the value it’s monitoring exceeds a specified threshold.

When that happens, a message is sent to an application task that needs to deal with this situation. A possible VxWorks implementation might be:

![Figure 3: Simple Sensor Example in VxWorks](image)
In Linux, hardware access and especially interrupt handling is entirely in the kernel domain. Moreover, kernel functions cannot generally make use of library functions or system calls. For this simple example, this might not be a problem. A standard driver could be implemented whose read function would block the calling task. The ISR would cause the system call to return and wake up the task.

![Simple sensor example in Linux](image)

Figure 4: Simple sensor example in Linux

This solution might succeed for this example, but there may be instances in which the differences may result in some architectural changes to the application during the port. Consider an application that consists of several tasks that communicate with each other using messages. A task can receive messages from either another task or from an ISR. Each task is composed of a main loop that checks for new messages, reads the messages, and performs some action based on the messages. Since the task might be receiving messages from several sources, it doesn’t block the receiving call, which presents a problem with the previous Linux implementation. The read driver call could be made non-blocking, but that would create unnecessary overhead since the device will be polled. The solution is to add another process (or thread) that would make the blocking call and send the message when it wakes up.

![Complex example Linux](image)

Figure 5: Complex example Linux

**Common Utility Functions**

Functions in this group provide the application with a set of common utilities. In the single address space RTOS environment, these functions were accessible to all tasks because all the code was statically linked together. There was just one copy of the function in a fixed memory location.
To port to Linux, the function must be compiled as a library. There are two options here:

1. Create a static library
2. Create a shared library

1. Static Library

A static library is a collection of logically related object modules that are placed together in an archive file. During compilation, the linker extracts the functions it needs from the library and adds them to the output image. RAM duplicates a library function if the function is used from more than one executable. Using the foo example, this will lead to:

```c
void foo (int a)
{ ....
};

Other functions

task A main()
{
    foo(5)
}

task B main()
{
    foo(6)

Kernel code
```

![Figure 6: RTOS Memory Layout](image)

2. Shared Library

Shared libraries provide a mechanism that allows a single copy of code to be shared by several programs in the system. Only a single copy of the library resides in physical memory.
In Figure 8, the shaded areas represent virtual memory mappings of the same code. Note that since each process can map the function into a different virtual address the library code must be compiled as position independent code (PIC).

**Static and Shared Libraries compared**

Static libraries are simple to use and the resulting executable is self-contained. They are also the only available choice on systems that do not support shared libraries or in cases in which it is not possible to generate position-independent code. Shared libraries can save system resources by ensuring that only a single copy of the library resides in physical memory. In addition, if a shared library is changed (e.g., to fix a bug), all the programs that use this library will benefit from the change.

**Library limitations**

When implementing commonly used functions from an RTOS as a Linux library, special care must be taken when dealing with global variables. Consider the following RTOS implementation:

```c
void foo (int a)
{
    ....
}

void foo (int a)
{
    ....
}

main()
{
    foo(5)
}

void foo (int a)
{
    ....
}

main()
{
    foo(6)
}
```

Here the function foo uses a global variable status that is declared in another file. Everything works here because all the components are linked together into a single object file. Of course, a race condition still exists in which task A and task B simultaneously call function foo, but these calls are logically assumed to be mutually exclusive. A previous step discussed the mappings of RTOS tasks to Linux threads or processes. If the tasks were mapped to threads then the solution is simple:
However, if the tasks were mapped to processes, the solution isn’t so simple. One might be tempted to do the following:

Note how the declaration of the global variable was instead moved to the library code because the processes are in separate name spaces. This implementation is fundamentally different from the original. Two completely separate copies of the global variable status — one for each of the processes — have been created.

The nature of the global variables (or other shared resources) must be determined. If they are used to indicate the state of some hardware device, then a library implementation is not suitable. A new driver must be implemented. Another possibility is that they are used to improve performance. For example, a string manipulation library might use a static buffer to do its work. This improves run time performance by eliminating the need to malloc/free space for the buffer. If that is the case, then a library can still be used, provided it is accessed only from different processes.

**System Calls and Library APIs**

Linux provides a rich set of system calls and library functions. Some of these calls are also supported by most RTOSes. However, the RTOS calls and Linux calls are rarely identical. For each of the system calls made by the application, a suitable substitute in Linux must be located. A substitute for an existing system call can fall into one of the following three categories:
1. **Identical** — The system call has an identical Linux system call. This is not uncommon since many operating systems offer at least some degree of POSIX compliance. Calls in this category require little if any porting work.

2. **Similar** — Calls in this category have a similar, but not identical, Linux counterpart. The differences are usually in the list of required parameters. Calls in this category can be ported in the following techniques:
   - **Using an emulation layer:** This technique creates an abstraction layer, most easily implemented as a library, that maps the old application calls to the most appropriate Linux call. Since the calls require different parameters, the abstraction takes care of supplying values for the missing parameters. This technique has the advantage of allowing the existing application code to remain untouched. However, it potentially adds run time overhead and it requires space for the abstraction layer.
   - **Recoding:** The application needs to be recoded to use existing system calls. This might require many changes to the existing application code.

3. **Other** — This category includes unique calls that have no corresponding Linux counterpart. There is no other option here other than to recode these application segments to use the system calls and library functions available in the Linux environment.

**Summary**

This paper examined the current application and identified all of its tasks. First, these tasks must be mapped to either a Linux process or a Linux thread. Reaching the optimal solution is an iterative process. Here are some of the more important guidelines:

- Use processes when coarse grain application level parallelism is required
- Use threads when finer grained parallelism is required within each process
- Use processes for better robustness and maintenance
- Use threads when mapping tasks that rely heavily on the fact that they are all in a single address space

Once the task mapping has been established, we need to identify all the code segments that access the hardware. These segments will need to be grouped according to the devices they access. Each group will then have to be implemented as a device driver. Once the drivers are identified, access to and from the hardware must be funneled through them.

The next step is to identify any utility code that is used by multiple tasks. To make this code accessible to different processes, it must be made into a library that is linked to the process’s image. This paper surveyed the possible options for making these libraries and discussed some of the possible problems that may arise due to reentrancy considerations.

The last phase of the port is to find a Linux equivalent to any RTOS kernel or library call made by the application. This paper discussed the various categories that these calls fall into with respect to how easy they are to port. Appendix A lists all the task-related system calls in VxWorks, with a short description on how to implement them in Linux.
Appendix A

The following is a list of all task-related system calls in VxWorks. The list was compiled from the VxWorks reference manual (version 5.4, Edition 1). The category names are mostly based on the VxWorks programmer's guide (version 5.4 edition 1) nomenclature.

Creation and Activation

1. taskInit — Initialize a new task
2. taskActivate — Activate a task that was created using taskInit()
3. taskSpawn — Create and activate a new task

Identification Information

4. taskName — Get the task name associated with a task ID
5. taskNameToId — Look up ID associated with task name
6. taskIdSelf — Get calling task’s ID
7. taskIdVerify — Verify existence of specified task

Task options

8. taskOptionsGet — Get tasks options
9. taskOptionsSet — Set task options

Task Information

10. taskIdListGet — Fill array with IDs of all active tasks
11. taskIdGet — Get information about a task
12. taskPriorityGet — Examine the priority of a task
13. taskPrioritySet
14. taskRegsGet — Examine the task's registers
15. taskRegsSet — Set task’s regs.
16. taskRegsShow
17. taskIsSuspended — Check if task is suspended
18. taskIsReady — Check if task is ready
19. taskTcb — Get pointer to task's control block
20. taskIdDefault — Set the default task ID
21. taskShow — Display task information
22. taskShowInit — Initialize the task show facility
23. taskStatusString — Get a task status as a string
**Task Deletion**

24. exit — Terminate the task (ANSI)
25. taskDelete — Terminate a specific task
26. taskDeleteForce — Delete a task without restriction
27. taskSafe — Protect calling task from deletion
28. taskUnsafe — Undo a taskSafe

**Task Control Routines**

29. taskSuspend — Suspend a task
30. taskResume — Resume a task
31. taskRestart — Restart a task
32. taskDelay — Delay a task for specified ticks
33. taskLock — Disable task scheduling
34. taskUnlock — Enable task scheduling

**Task Variables**

35. taskVarAdd — Add a task variable to a task
36. taskVarDelete — Remove a variable from a task
37. taskVarGet — Get the value of a task variable
38. taskVarInfo — Get a list of task variables of a task
39. taskVarInit — Initialize the task variables facility
40. taskVarSet — Set the value of a task variable

**Task Extension functions**

41. taskHookInit — Initialize task hook facilities
42. taskHookShowInit — Initialize task hook show facility
43. taskCreateHookAdd — Add a routine to be called at task creation
44. taskCreateHookDelete — Delete a previously added task create routine
45. taskCreateHookShow — Show the list of task create routines
46. taskSwitchHookAdd — Add a routine to be called every task switch
47. taskSwitchHookDelete — Delete a previously added task switch routine
48. taskDeleteHookAdd — Add a routine to be called at every task delete
49. taskDeleteHookDelete — Add a previously added task delete routine
50. taskDeleteHookShow — Show the list of task delete routines
Definition of Terms

Following are definitions of terms that are used throughout this paper:

- **Kernel** — The kernel is the core component of any operating system. A minimal kernel would include an interrupt handler to handle hardware interrupts, a scheduler that determines the order in which the available tasks should run and a supervisor that actually allocates the CPU to the most appropriate task. Usually kernels also manage the rest of the system’s hardware devices and provide a host of other services that may be requested by other parts of the operating system or by application code. These services are provided through a specified set of program interfaces known as system calls.

- **Device Drivers** — A device driver is a logical grouping of functions that access a given device. A driver encapsulates the complexities of the hardware device into an object that has certain properties and methods. Other parts of the system can access the actual device by accessing this object.

- **Task** — A task is a basic unit of code that can be scheduled for execution by an RTOS kernel.

- **Thread** — A thread is the Linux equivalent to an RTOS task (above).

- **Process** — A process may be composed of one or more threads all sharing the same address space.

- **Inter Process Communication** — Inter Process Communication is the information exchange between the different application tasks or processes.