10 Steps to Virtualization

WHEN IT MATTERS, IT RUNS ON WIND RIVER

EXECUTIVE SUMMARY

Virtualization—the creation of multiple virtual machines (VMs) on a single piece of hardware, where each VM emulates an actual computer so that each VM can execute a different operating system—has gained significant momentum in recent years. Enterprises have discovered that virtualization can help reduce costs by making more efficient use of hardware and computing resources. It can often deliver advantages in the areas of performance, flexibility, and time-to-market as well.

For developers of embedded systems, virtualization brings new capabilities and possibilities. It allows system architects to use multiple operating systems to design their devices, combining the best of all words. An embedded system, for example, could use Microsoft® Windows® for the human–machine interface (HMI), VxWorks® for real-time behavior, and Linux for the connection into the cloud. Each operating system runs on a different set of cores and has its own devices. Virtualization allows the system architect to design without compromise.

Focusing on embedded virtualization through VxWorks, this paper discusses how to use virtualization to design an embedded system, considerations for the system architect, and the steps required to design a great new device.

TABLE OF CONTENTS

STEP 1: OPERATING SYSTEMS

Which operating systems should be used to best design a system? It depends on your goals. If networking is important, Linux has the open source advantage of being able to quickly reuse middleware and applications.

Are graphics important, especially in an HMI? Does the end customer want to run existing applications on the device? If so, Windows is a logical choice.

If you need real-time capabilities, low, microsecond (or even sub-microsecond) response times, and determinism in the execution logic, a real-time operating system (RTOS) such as VxWorks is the answer.

Each operating system added will run in its own dedicated VM and will need resources: cores, memory, and devices. The amount of resources depends on the amount of processing required of each operating system.

Virtualization is achieved with a hypervisor used to build and run VMs. An operating system does not need to be modified to run on top of a hypervisor. It may, however, need additional device drivers (as described under Step 7: Virtual I/O).

STEP 2: PROCESSOR

Processors are crucial to support efficient virtualization. Modern processors provide hardware support for virtualization, a special privilege level that gives the hypervisor (the core component of virtualization) ultimate control over the hardware. The hypervisor has more privileges than an operating system and works together with the processor to partition the available hardware between the VMs. The major processor architectures all provide hardware acceleration support: ARM® with its Virtualization Extensions, Intel® with V-PRO, and Freescale with its e500mc and later cores.

The selected processor needs to have sufficient cores to run all the operating systems chosen under Step 1 and give them the necessary processing power. For example, a quad core processor could give one core to VxWorks, one to Linux, and two to Windows.

Hyperthreading is also a useful technique. It makes a quad core processor operate like an 8 core, with logical cores at the hardware level. Hyperthreading is typically effective for operating systems such as Linux and Windows, but the impact on determinism is significant, so it will not be appropriate for true real-time loads in most cases.

The processor needs to have sufficient memory as well. Each VM will require a piece of memory. For Windows, the memory required is typically counted in multiple gigabytes; Linux has more modest demands; and an RTOS like VxWorks counts its minimum memory requirements in kilobytes or megabytes, depending on its configuration.

The hypervisor is the component in charge of separating these resources; it will make sure that a core or memory given to VxWorks will not be used by Windows (or another VM).

STEP 3: SAFETY AND CERTIFICATION

It is important to understand what level of safety and security is required. The higher the level of safety and security, the more restrictions the system architect has to account for, and the more costs will be incurred.

Safety is related to the risks of devices that interact with humans or other systems. Such devices must be certified as meeting one or more safety standards, such as IEC 61508, ISO 26262, CENELEC EN 50128, or DO-178C. In these designs, typically the virtualization layer needs to be safe, as well as one or more of the VMs. Again, safety needs to be considered early in the system design.

Virtualization plays a role in safety, in that it can

a) Combine multiple safety applications in separate VMs on a single piece of hardware to provide redundancy. This makes it possible to achieve redundancy on a single multi-core. As always, this requires careful system design.
b) Combine multiple levels of criticality; that is, it can run one VM that has a high safety requirement next to one that has a low safety requirement. It allows the high-safety VM to remain constant (not requiring re-certification) while the low-safety VM can be updated frequently.

STEP 4: SECURITY

Security is directly linked to how the system is booted, how data is secured at rest and in transit, and which applications are allowed to run in which VM on the system. Security is a huge topic and needs to be considered from the beginning, as retrofitting an existing design is often difficult.

Security demands directly affect the virtualization layer. It needs to have such capabilities as secure boot and image validation before the VM is booted.

The virtualization layer itself typically does not have external interfaces, and hence has a small attach surface; it relies on the projection inside the virtual machines.

STEP 5: BOOTING

Within the constraints of safety and security requirements, the system architect needs to think about how the system will be booted. Typically there is some amount of physical storage from which the boot-loader starts the system. This may be flash storage, USB, Serial ATA (SATA), or anything else that fits within the device functionality and footprint.

Booting takes place in stages. First, the system may run a basic input/output system (BIOS) or some sort of early firmware, followed by the boot-loader, then by the embedded software, which includes the virtualization component.

VxWorks provides virtualization capabilities implemented directly in the operating system. The embedded software layer that is booted is thus a full-fledged operating system, also known as the RootOS. The RootOS is then in charge of booting the VMs as the system architect sees fit.

The kernel for the VMs may be included in the RootOS, and thereby loaded by the boot-loader directly, or it may reside in some form of local or remote storage. This is typically how RTOSes or small executives are booted.

The RootOS can also boot a VM directly from disk by loading a boot-loader in the VM and pointing the boot-loader to a partition. This is typically how larger operating systems such as Linux and Windows are booted.

Booting has a direct connection to Step 9: Upgrades. Today's embedded systems are seldom static—they need to be able to download new workloads; hence the local storage will need to be updated with that.

STEP 6: INPUT/OUTPUT

Because the VMs in the system need to communicate with the outside world, they need I/O devices. Typical I/O devices are network cards, serial ports, graphics cards, SATA, USB, and so forth.

The easiest and most efficient path to I/O is to give each VM direct access to the devices it needs. For example, a graphics card would be passed directly through to Windows, and the standard Windows device driver can make use of it. In this case, the other VMs do not have access to this device. Only one VM can use a device under a direct access or pass-through assignment.

Pass-through device access also optimizes performance because the VM has direct access to the device. The hypervisor layer is not involved in accessing, and the device can have direct memory access to the VM.

The hypervisor is the component that enforces separation. The individual VMs only see the devices that the system architect has assigned to them. For example, if a VM would enumerate the PCI bus, it expects access to the physical bus, but the hypervisor owns the physical bus and provides an emulated PCI controller to the VM. This emulated controller only provides the devices that are assigned to the VM to show up on that PCI bus.

Direct assignment may not be appropriate for all I/O devices. For example, the system architect may not want to give Windows direct access to an external network card, but instead use Linux as a firewall and let Linux share that connection with Windows, after passing the traffic through a firewall.

Another challenge is that often there simply are not enough devices to connect to all the VMs. In that case, virtual I/O can be used, which is detailed in the next step.

STEP 7: VIRTUAL I/O

Virtual devices help in situations where it is not possible to assign devices as pass-through to the VM. For example, the system may only have a single SATA storage controller, but multiple VMs need access to storage. Another example is that VMs need to communicate together, and it would be a waste to use physical devices for this communication.

There are many virtual devices that are of no concern to the system architect. For example, every VM gets access to a virtual PCI bus that shows the devices assigned to that VM. The hypervisor virtualizes and emulates this under the surface. Similarly, the hypervisor provides virtualized access to interrupt controllers and features at the board and processor level. The operating system in the VM does not even know it is acting on virtual devices. The hypervisor hides that fact.

A key artifact with virtual I/O is performance. With virtual I/O the system adds software into the device path, and for embedded systems the performance of accessing devices is typically very important. The hypervisor is perfectly capable of emulating a physical Ethernet card, but the overhead to do so is significant. Instead, virtual I/O systems typically make use of para-virtualization.

With para-virtualization, the hypervisor provides a software access point to the VM. The VM has a special driver, called a para-virtualized driver, that collaborates with the hypervisor and occasionally with the RootOS. These para-virtualized drivers can easily be added to existing operating systems without further need to modify the OS.

VxWorks uses the VirtIO standard to provide virtual serial and virtual storage capabilities. In VirtIO, the operating system in the VM discovers the software access point through an emulated PCI device. This emulated device is used in the configuration path only. The operating system can then load the driver and start using the virtual device. When the operating system interacts with the device, it is actually interacting with a driver in the RootOS that provides the device functionality.

VirtIO is an open source standard and most modern operating systems already provide drivers for many virtual devices. These drivers can be reused to access VirtIO virtualized devices on VxWorks.

VirtIO serial can be used to create a virtual serial connection between the RootOS and the operating system, for instance Linux. This serial can be used for data communication or for serial access to a console. The serial connection shows up in the operating system as any device, /tyCo/x under VxWorks and /dev/hvcx under Linux.

In the case of VirtIO storage, the RootOS virtualizes the block device and provides access through the VirtIO block device driver to the VMs. The RootOS can give a VM access to a partition and the operating system will use its VirtIO storage driver to access that partition.

As an example, a system can have a single SATA disk with four partitions. The first partition contains the system image of VxWorks that brings up the virtualization layer. The second partition is persistent storage for VxWorks to store configuration and data log files. The third partition is for Linux, and the fourth is for Windows. Each VM can be configured to have access only to its own partition, but Linux can be given read-only access to the VxWorks partition so VxWorks can send log data into the cloud if needed.

The system architect will need to make sure that the RootOS has sufficient processing cycles to support the VirtIO functionality. For example, if the RootOS virtualizes a disk, then all storage actions to that disk will flow through the RootOS.

Another virtual device that is available is the virtual network. VxWorks can provide a high-speed virtual network bus between the VMs, which can be used for the VMs to communicate with each other using standard TCP/IP stacks. This virtual network uses shared memory between the VMs, and therefore can achieve higher speeds then normal physical networks.

As an example, Linux can be given a physical Ethernet device in pass-through mode while Windows and Linux are each given a virtual network device. Linux is then the firewall and can provide firewalled external access to Windows.

STEP 8: PERFORMANCE

Performance is crucial in embedded systems. Different projects require different types of performance. Typical performance characteristics include deterministic response latencies measured in microseconds, algorithmic performance measured in operations per second, or network throughput and latency measured in Gbps and microseconds.

Performance is highly dependent on how the system is configured—the number of cores a VM has, the amount of memory, and whether it has pass-through devices or virtual devices.

It also depends on how much load the system carries. Modern CPUs have invisible shared resources such as a system bus or cache, and the load on these shared resources can impact the performance of the overall system.

The initial design for a system can be worked out in PowerPoint® or on a white board, but the final performance can only be determined by running benchmarks and fine-tuning.

Tuning can involve re-assigning memory and cores, re-prioritizing interrupts or tasks, or temporarily slowing down VMs, known as "throttling," to make sure the real-time task has the processing power it needs.

STEP 9: UPGRADES

Systems are often updated in the field. The system architect will need to plan for this possibility. In the case of a virtualized system, the initial embedded software may need updating, as well as the images running inside the VM. The hypervisor provides lifecycle control to be able to halt and reset VMs, but the system architect will need a plan for getting the bits to the system.

For example, in a system consisting of VxWorks, Linux, and Windows, the Linux VM may have connectivity into the cloud and receive a new system image or a new image for VxWorks. It can then pass that image to the RootOS, which can store it on the disk, and either a VM or the entire system can be rebooted to use the new image.

STEP 10: DEVELOPER TOOLS

Systems are built by people. It is important to give people the tools that enable them to quickly explore virtualization. VxWorks has always been very flexible and adaptable to this end. A developer can use the VxWorks command line to launch and stop tasks and processes, debug, and access file systems. This same command line can also be used to create VMs and control device assignment in a very flexible, and either interactive or programmatic, fashion.

The VxWorks debug interface provides standard debugger actions such as run, breakpoints, and stepping, but it also provides an extremely useful host file system capability that allows the developer to launch binaries for processes and VMs directly from the host development workstation. The developer can also use the Wind River® Workbench integrated development environment for developing and debugging. Workbench is a single (Eclipse-based) environment that can be used to develop, debug, and analyze the virtualization layer, VxWorks, and Wind River Linux workloads.

The VxWorks interface provides access to the system through the services of the operating system. In certain cases it is important to be able to debug close to the hardware without relying on the operating system—for example, when you are doing device driver development. In these cases a hardware-based debugger provides great value, as it allows you to do stop-mode debugging, stopping the entire system to look at the system state. This is what Lauterbach provides in its TRACE32 line of products. These products allow you to go underneath the operating system and access the bare hardware, while still having visibility into OS-level concepts such as tasks and processes and their memory spaces, and while being aware of VMs and the hypervisor.

CONCLUSION

Virtualization gives system architects tremendous flexibility in system design and enables them to overcome hardware limitations in the delivery of multiple applications. It has performance advantages as well, bringing high application availability and exceptional scalability. Understanding how to successfully deploy virtualization—the 10 steps enumerated in this paper—is the first step to realizing its many benefits.

Wind River offers the industry's most comprehensive embedded virtualization development portfolio. Our solutions support multiple architectures and are backed by knowledgeable professional services, global customer support, hardware integration expertise, and a vast partner ecosystem. Leveraging this combination of technology and expertise, Wind River helps developers harness the many benefits of embedded virtualization.