About This Document

This *User's Guide* explains the QNX hypervisor architecture and provides instructions for installing and running a QNX Hypervisor system, changing system components and configuration, and using hypervisor features such as virtual devices (vdevs).

What's in this guide

To find out about:	**See:**
QNX virtual environments, including the architecture of a QNX hypervisor system	"Understanding QNX Virtual Environments"
Physical and virtual devices in a QNX hypervisor system	"Devices", "Virtual devices", and "Physical devices"
The protection features used by QNX hypervisors	"QNX Hypervisor: Protection Features"
Configuring the hypervisor host domain, virtual machines (VMs), and guests	"Configuration"
Building and booting the hypervisor and its guests	"Building a QNX Hypervisor System"
Starting and stopping the hypervisor	"Booting the QNX hypervisor host" and "Shutting down the QNX hypervisor"
Networking in a hypervisor system	"Networking"
Memory sharing between guests, and between guests and the hypervisor host	"Memory sharing"
Debugging your hypervisor system	"Monitoring and Troubleshooting"
Tuning your hypervisor system for optimal performance	"Performance Tuning"
Defining options to assemble and configure the VM in which a guest will run	"VM Configuration Reference"
Configuring the vdevs that will be included in a VM to provide the guest with the devices it needs	"Virtual Device Reference"
Utilities and drivers delivered with the QNX hypervisor	"Utilities and Drivers Reference"

QNX hypervisor variants

WARNING:
QNX hypervisors are available in two variants: QNX Hypervisor and QNX Hypervisor for Safety.

The QNX Hypervisor variant (QH), which includes QNX Hypervisor 8.0, is *not* a safety-certified product. It must *not* be used in a safety-related production system.

If you are building a safety-related system, you must use the QNX Hypervisor for Safety (QHS) variant that has been built and approved for use in the type of system you are building, and you must use it *only as specified in its Safety Manual*. The latest QHS release is QNX Hypervisor for Safety 2.2, which is based on QNX SDP 7.1.

If you have any questions, contact your QNX representative.

Non-hypervisor components

This *User's Guide* includes and identifies as such documentation for some non-hypervisor QNX components that are of interest to QNX Hypervisor users but not described in the QNX OS documentation.

QNX Hypervisor GitLab Repository

The QNX Hypervisor GitLab Repository contains technical notes and FAQs about different QNX hypervisor topics, including building custom virtual devices and supporting guests for a QNX hypervisor system. The location of this repository is: https://gitlab.com/qnx/hypervisor.

The documentation about writing custom devices includes the *Virtual Device Developer's Guide*, which contains instructions and source code samples, and the *Virtual Device Developer's API Reference*, which describes the API for developing the devices.

Page updated: August 11, 2025

Terminology

The following terms are used throughout the QNX hypervisor documentation.

Blob
: The guest as seen by the hypervisor host. The term *blob* is used to emphasize that fact that the hypervisor can't know any more about its guests than a board can know about the OS running on its hardware.

CPU privilege level
: A CPU privilege level controls the access of the program currently running on a processor to resources such as memory regions, I/O ports, and special instructions (see "CPU privilege levels" in the "Understanding QNX Virtual Environments" chapter).

DMA
: Direct Memory Access

Exception level (EL)
: ARM-defined levels of permissions at which processes can run, with EL0 having the least privileges, and EL3 having the most (used for security).

Emulation
: Emulation refers to the capability to mimic a particular type of hardware for an operating system regardless of the underlying host operating system. In a QNX environment, emulation refers to emulated hardware devices that a guest will see as real hardware.

  In a virtualized environment, a hypervisor includes virtual devices (vdevs) that *emulate* physical devices. From the perspective of a guest, these vdevs look like the physical devices they emulate. The guest needs drivers to manage them, and receives interrupts and sends signals to the vdevs, just as it would to a physical device in non-virtualized system.

  For example, the hypervisor host domain typically emulates an interrupt controller for each guest. An emulated driver may communicate with the underlying hardware, but the guest doesn't need to know if this is the case. In fact, the physical device being emulated doesn't even have to exist on the hardware (see "Virtual devices" in the "Understanding QNX Virtual Environments" chapter).

EOI
: End of Interrupt

Execute, Execution
: The act of completing an instruction on a physical CPU. Contrast with "Run" below. In a QNX hypervisor system, the hypervisor host runs directly on the hardware, while a guest OS runs in a virtual machine (VM) which has virtual CPUs (vCPUs). Ultimately, however, both the hypervisor and the guest *execute* on the physical CPUs (see also "Lahav Line" below).

FIQ
: Fast interrupt request: a feature of some ARM boards, FIQs are higher-priority interrupt requests, prioritized by disabling IRQ and other FIQ handlers during request servicing; no other interrupts are processed until processing of the active FIQ interrupt is complete.

Guest
: A *guest* is an OS running in a QNX hypervisor `qvm` process; this process presents the virtual machine (VM) in which the guest runs (see "A note about nomenclature" in the "Understanding QNX Virtual Environments" chapter).

Guest-physical address
: A memory address in guest-physical memory (see "Guest-physical memory" below).

Guest-physical memory
: The memory assembled for a VM by the `qvm` process that creates and configures the VM. ARM calls this assembled memory *intermediate physical memory*; Intel calls it *guest physical memory*. For simplicity, regardless of the platform, we will use the term Bugnion, Nieh and Tsafrir use in *Hardware and Software Support for Virtualization* (Morgan & Claypool, 2017): "guest-physical memory", and the corresponding term: "guest-physical address".

Host
: Either the *development host* (the desktop or laptop, which you can connect to your target system to load a new image or debug), or the *hypervisor host domain*. Unless otherwise specified (as in the instructions for building a hypervisor system or transferring it to a target), in the hypervisor documentation *host* refers to the *hypervisor host domain* (see below).

Host-physical address
: A memory address in host-physical memory (see "Host-physical memory" below).

Host-physical memory

The physical memory; this is the memory seen by the hypervisor host, or any other entity running in a non-virtualized environment (see "Guest-physical memory" above).

Hypervisor

In the QNX context, a software system consisting of a QNX OS microkernel with its hypervisor host mode enabled and one or more virtual machines.

Hypervisor host domain

The hypervisor and all its components. The hypervisor host domain (or simply *hypervisor host* or *host domain*) is the lowest layer of the software stack that boots first after a board power-up or reset, and must be running before any guests can be launched.

The hypervisor host domain includes the `qvm` processes that present VMs in which guests run and can also include drivers that support para-virtualization, and components that monitor VMs and perform other system monitoring and maintenance tasks.

IOMMU

Input/Output Memory Management Unit. A memory management unit (MMU) that connects a DMA–capable I/O bus to the main memory. Like a traditional MMU, which translates CPU-visible intermediate addresses to physical addresses, an IOMMU maps device-visible intermediate addresses (also called device addresses or I/O addresses in this context) to physical addresses. This mapping ensures that DMA devices cannot interact with memory outside their bounded areas.

Intel Virtualization Technology for Directed I/O (VT-d) is an Intel IOMMU implementation. Both x86 (VT-d) and ARMv8 (SMMU) provide hardware support for IOMMU. In a QNX environment, IOMMU is enabled when using pass-through mode in a guest (see your hardware documentation).

IPI

Interprocessor interrupts. Used by the hypervisor to communicate between its own cores, or to interrupt VMs.

Interrupt Translation Service

Lahav Line

A line that describes how a hypervisor host runs directly on the hardware, while a guest runs in hypervisor virtual machine with virtual CPUs, but both *execute* on physical CPUs (see "Two representations of a QNX hypervisor system").

Message Signal Interrupts

NUC

Next Unit of Computing, an Intel embedded platform.

Para-virtualization

A virtualized environment in which a guest knows that it is a guest, and acts accordingly; for example, the guest uses a para-virtualized device (for which no hardware equivalent exists) instead of an emulation vdev (which emulates a hardware device).

Pass-through

Pass-through is a technique for giving a guest direct access to hardware. Pass-through allows a device driver located in a guest to control a hardware interface directly (see "Pass-through devices"in the "Understanding QNX Virtual Environments" chapter).

pCPU

Physical CPU. Each pCPU corresponds to one hardware core. For example, a quad-core SoC will have four pCPUs (compare with *vCPU*).

PIC

Programmable Interrupt Controller

PPI

Per-processor interrupt

PSCI

Power State Coordination Interface: an ARM API used to coordinate power control among supervisory systems running concurrently on an ARM board.

QNX guest

A guest system running a supported QNX OS variant.

QNX Hypervisor (QH)

A non-safety hypervisor product from QNX. When built into an image and deployed to a target board, a QH system consists of a running instance of the QNX OS that has hypervisor host mode enabled.

QNX Hypervisor for Safety (QHS)

A safety-certified variant of a QNX hypervisor. When running on a target board, a QHS system consists of a running instance of the QNX OS for Safety that has hypervisor host mode enabled.

QNX OS

A non-safety variant of the operating system from QNX.

QNX OS for Safety (QOS)

A safety-certified variant of the QNX OS.

QNX Software Development Platform (SDP)

Our cross-compiling and debugging toolset for building binary images and programs for ARM and x86_64 targets running the QNX OS.

QNX virtual machine (qvm)

A process in the hypervisor that hosts a guest by presenting a virtual machine (VM) in which a guest can run.

Ring

x86 levels of permissions at which processes can run, with Ring3 having the least privileges, and Ring0 having the most.

Run

In a QNX hypervisor system, the hypervisor host runs directly on the hardware, while a guest OS runs in a virtual machine (VM) which has virtual CPUs (vCPUs). Ultimately, however, both the hypervisor and the guest execute on the physical CPUs (see also "Lahav Line" above).

SMP

Symmetrical Multiprocessing.

vCPU

Virtual CPU: a hypervisor `qvm` process thread that emulates a pCPU (see "pCPU" above). The guest hosted in the `qvm` process sees the vCPU as a pCPU. As the hypervisor schedules VMs to run, it can attach vCPUs to a pCPU or detach them from a pCPU. You can configure your vCPUs to float between pCPUs, leaving the decision of where the vCPU should run to the hypervisor, or you can restrict vCPUs to individual pCPUs.

vdev

Virtual device (see "Devices").

VIRTIO

A standard for virtual devices that "by design ... look like physical devices to the guest within the virtual machine" (see the OASIS specification: Virtual I/O Device (VIRTIO) Version 1.0). A guest's VIRTIO device driver is aware that it is running in a virtual environment, and cooperates with the hypervisor to allow the sharing of an interface.

The hypervisor and the guest must both be configured for VIRTIO support. Typical use cases for VIRTIO devices involve sharing (between guests) of Ethernet, of block devices (storage), of memory, and of consoles. A VIRTIO driver in the host domain can also be shared between the host domain and the guest, if required.

The QNX virtualized environment supports versions 0.95 and 1.0 of VIRTIO.

Virtualized system

A software system that includes a QNX hypervisor hosting one or more guests.

VM

Virtual machine. In a QNX hypervisor virtualized environment, the hypervisor creates a `qvm` process instance for each guest it hosts. Each `qvm` process presents a VM to its guest.

Page updated: August 11, 2025

Preparing your target board

After you have built your QNX hypervisor system, you must prepare your supported hardware board to boot and run this system.

Preparing a target board for a hypervisor system is no different from preparing one for a non-virtualized system. You need to set DIP switches to configure board behavior (such as where it looks for IPL and startup code), and connect Ethernet, USB, and serial cables so you can connect to your development host and your network. For more information, see the BSP User's Guide for your board, and the board manufacturer's documentation.

x86 boards

On x86 boards, you can use the serial port to establish a terminal connection from the development host to the board (and hence, access the hypervisor system via a shell). This provides a very useful view into the target board without requiring the setup of a TCP or USB (keyboard) connection. For details on how to configure a serial port for terminal connection, see the BSP User's Guide for your board.

For some boards, the system shell is active *only on the serial port*; it isn't active on the VGA console. If you connect via the VGA console, you will see a logo and it will seem like the board is stalled somewhere in its boot process.

The default `smmuman` configuration instructs the service to use the board's ACPI tables to get the locations of the VT-d registers responsible for remapping PCI device DMA. Make sure that your startup doesn't include the `-B` option, which instructs the startup to not get the ACPI tables.

For more information, see the `pass` option reference, and the "DMA device containment" section.

ARM boards

To run a virtualized system, ARM boards require firmware that boots into exception level 2 (EL2). On some older boards the required firmware isn't installed when they are shipped. If you attempt to boot the hypervisor on these boards, the boot will fail with a message like the following:

```
# qvm @qnx80.qvmconf
QVM disabled: hypervisor support not available
```

If your board doesn't boot into EL2, you probably have an old board revision that the QNX hypervisor doesn't support. Contact your board manufacturer to get a newer, supported board with a newer BSP.

See the BSP User's Guide for your board to learn about the DIP switch settings to use for your board variant and revision.

Page updated: August 11, 2025

Building a QNX Hypervisor System

This chapter explains how to build (assemble) a QNX Hypervisor system and transfer a bootable image to a supported target platform.

When you build a QNX hypervisor, you don't recompile the hypervisor. If you added or modified components (e.g., drivers or vdevs) you of course have to compile them. However, QNX hypervisors are provided as binaries only. You don't compile them; you include them unmodified when you assemble a bootable image with your hypervisor system.

Prerequisites

NOTE:

Before you begin to work, check that your board's hardware *and* firmware support virtualization.

Even if your board hardware supports all of the virtualization capabilities needed to run a QNX virtualized system, the firmware on the board may not enable or may even expressly disable some of these capabilities, in which case you won't be able to run the hypervisor.

You can contact your board vendor to confirm that the board you are considering supports the virtualization capabilities you'll need, and that your board has the appropriate firmware.

For more information about firmware requirements, go to "Preparing your target board."

To build a hypervisor system, you need to configure and build both the hypervisor host and its guests. These instructions assume that you know:

- about QNX Board Support Packages (BSPs)
- about QNX buildfiles
- how to use the QNX Software Center
- how to build an IFS with a QNX OS

Page updated: August 11, 2025

Supported build environments

The QNX Hypervisor host must be built in the appropriate QNX SDP environment, and guests must be built in their own environments.

Your development host machine may be a Linux or Windows system. For more information about supported development hosts and for instructions on how to install the QNX SDP onto your own machine, see the *QNX SDP 8.0 Release Notes*.

If you want to work with a supported QNX guest or a Linux or Android guest, you must have the appropriate build environments on your development host. You will need to set up these environments and configure your builds accordingly (see "Configuring your QNX build environment" below, and "Building guests" in this chapter).

If you add vdevs or otherwise modify your hypervisor system, you must build and configure a new bootable image that includes your changes. The table below lists the build environments required for the hypervisor host and its guests:

NOTE:
For readability, we use a "+" (PLUS) as a short form for "or more recent"; for example, "QNX SDP 8.0+" means "QNX SDP 8.0 or more recent".

When you set up your QNX SDP 8.0+ environment, you should update your installation with any available QNX SDP 8.0 patches.

Component	Environment	More information
QNX Hypervisor 8.0 host	QNX SDP 8.0+	QNX SDP 8.0 documentation — this documentation explains the QNX SDP toolkit for developing embedded systems that run the QNX OS.
QNX OS 8.0 guest	QNX SDP 8.0+	QNX SDP 8.0 documentation
Linux guest	Linux	Linux documentation
Android guest	Android	Android documentation

Configuring your QNX build environment

After you have installed QNX SDP, any required updates, and QNX Hypervisor onto your development host, you must run the `qnxsdp-env.sh` script (for Linux) or the `qnxsdp-env.bat` batch file (for Windows) to set up your build environment so you can build QNX OS guests or the hypervisor host.

To do this on Linux, in your QNX SDP installation's base directory, source the script. For example, in a Bash shell:

```
cd ~/qnx800/
source qnxsdp-env.sh
```

If you are using another shell (e.g., Z shell), you may need to use a different command to source the script. For example:

```
export $(bash -c "source qnxsdp-env.sh")
```

To do this on Windows, open a command prompt by running `cmd.exe`, navigate to the installation's base directory and run the batch file. For example:

```
C:\Users\your_name\qnx800>qnxsdp-env.bat
```

This script sets the basic environment variables used to build a QNX OS image. You will set other variables when you configure your build.

NOTE:
If you work with a Linux or Android guest, you need to set up the environment with the toolchains for these OSs.

Page updated: August 11, 2025

Building guests

When you build guests to run in a QNX virtualized environment, you must build them for the appropriate hardware architecture, and configure the VMs in which they will run.

You need to have the build environment and tools that correspond to the guest OSs you're using (see "Supported build environments").

Validating devices and guests

It may be easier to identify and debug some guest OS and device problems in a non-virtualized environment, before you integrate it into in your hypervisor system. In most environments, guests and devices also need further validation in the hypervisor environment.

For example, run the host OS on your target board and validate that devices work as intended with the host, before trying to configure them as pass-through devices. Checking that pass-through devices work with the host OS can also be helpful in figuring out the pass-through parameters that a given device requires (i.e., by running `pidin` to find which memory regions and hardware interrupt requests it uses).

In some environments, it could be useful to run the guest OS as the host on your target board, to validate that devices work as intended with it directly, before trying to configure them as pass-through devices. However, if you're building a QNX OS system to use as a guest, the startup driver required for the guest may not work directly on the board; you may have to change the buildfile for the guest to (temporarily) use the appropriate driver for starting it up directly on the board.

CAUTION:
Remember that the hardware presented by a `qvm` system is fundamentally unlike the hardware presented by the host system. (This characteristic is why the startup driver for guests corresponds to the model that `qvm` presents to the guest, not the board on which the host resides.)

Further validation of a guest in the hypervisor environment is needed in any of the following situations:

- The guest uses para-virtualized devices, which by definition don't exist as hardware.
- The guest uses vdevs, meaning the configuration for the VM that hosts the guest must be updated to include these vdevs (see "Configuring the VMs and the guests to match" below).
- Some devices aren't passed through to the guest. In this case, you need different drivers when you run the hypervisor system so the hypervisor host can manage these devices.

Supported guest formats in images

The hypervisor can launch guests placed on the system as bootable images in the following formats:

- ELF (including multiboot)
- Linux zImage

Configuring the VMs and the guests to match

If you change anything in your guest about how it accesses hardware (physical or virtual), you need to ensure that the `qvm` configuration file for the VM that will host the guest is properly configured.

When building any kind of guest to run in a QNX virtualized environment, it is important to remember that the guest must be configured to match the VM in which it will run. This means that it must have the drivers it needs to access devices, whether they are physical or virtual. In the VM, the drivers must be configured to be where the guest expects them to be.

The only difference between QNX guests and non-QNX guests is that QNX guests use BSPs to bring in the architecture-specific and board-specific components, while other OSs may use other mechanisms to achieve the same end.

For more information, see "Assembling and configuring VMs" in the "Configuration" chapter.

Page updated: August 11, 2025

Building the host

After you have installed the QNX Hypervisor product, all of the components needed to build the hypervisor host should be on your development host. After you have configured your build environment and built one or more guests, you are now ready to build the hypervisor host into an IFS.

Building with a hypervisor buildfile variant

If your BSP provides a hypervisor buildfile variant, to build the host:

1. Unzip the archive that contains the BSP.
2. Locate the hypervisor buildfile variant within your BSP. This buildfile variant is in the same directory as the generic buildfile but its name contains **-hypervisor**; for example, **/images/mek/imx8qm-cpu-mek-hypervisor.build**.
3. If you want to use the `shmem-host` shared memory demonstration application, you need to build it. For more information, go to "Adding the shmem-host shared memory demonstration application."
4. Use the following command to make the BSP's prebuilt binaries available to the BSP's build system:

   ```
   make prebuilt
   ```

5. Change to the **/images** directory (e.g., `cd images/`), and then use the following command to build the host IFS:

   ```
   make hyp
   ```

   NOTE:
   If you run simply `make`, the build utility uses the default buildfile that comes with the BSP and, thus, won't build a hypervisor host image. The command `make hyp` won't work if the QNX Hypervisor product is not installed.

6. Use the following command to build the host disk image:

   ```
   make disk_image
   ```

Building without a hypervisor buildfile variant

If your BSP does not provide a hypervisor buildfile variant, to build the host:

1. Unzip the archive that contains the BSP.
2. If you want to include the `shmem-host` shared memory demonstration application, you need to build it using source code you obtain from a BSP provided with QNX Hypervisor. For more information, go to "Adding the shmem-host shared memory demonstration application."
3. Use the following command to make the BSP's prebuilt binaries available to the BSP's build system:

   ```
   make prebuilt
   ```

4. Change to the **/images** directory (e.g., `cd images/`).
5. Modify the buildfile to include the files you need for your hypervisor system at the appropriate target locations. These include:
   - The QNX virtual machine binary, `qvm`, and, optionally for x86, the `qvm-check` utility. For example:

     ```
     /sbin/qvm = qvm
     /bin/qvm-check = qvm-check
     ```

   - The configuration files (***.qvmconf**) for each VM that you will run in your hypervisor system:

     ```
     infotainment_ubuntu.qvmconf
     ```

   - The shared object (**vdev-*.so**) file for every vdev you want to make available for your guests. For example:

     ```
     vdev-8259.so
     vdev-hpet.so
     vdev-ioapic.so
     ...
     vdev-virtio-console.so
     vdev-virtio-input.so
     vdev-virtio-net.so
     ```

- The `smmuman` service, architecture, and board support libraries, and configuration files. For example, for a QNX hypervisor system on a supported Intel x86 board:

  ```
  /bin/smmuman = smmuman
  /bin/smmu-vtd.so = smmu-vtd.so
  /etc/smmuman/vtd.smmu = ./smmuman-config/vtd.smmu
  ```

6. Modify the buildfile to include `-Q enable` in the startup driver arguments, inside the virtual attribute that specifies arguments to the startup driver and kernel.
7. Run `make` to build the host IFS using the host buildfile.
8. If it's applicable in your environment, build a bootable disk image:
   - Use the `make disk_image` command, if your BSP supports it.
   - For other BSPs, you can create a system disk image using the `diskimage` utility (go to the `diskimage` entry in the QNX OS *Utilities Reference*).
   - For an example of host disk image creation, in a QNX-supplied BSP that supports QNX Hypervisor, examine the Makefile located at **images/*board_variant*/Makefile**. The entry for the `disk_image` target illustrates which tools and scripts are called, and which configuration and buildfiles are used (some of which are found in the **images/** folder).
   - For more information, go to "Hypervisor disk images" and "Transferring the disk image."

Adding the `shmem-host` shared memory demonstration application

If you want to use the `shmem-host` shared memory demonstration application, use the following steps:

1. If you are not using a host BSP provided with QNX Hypervisor:
   a. Unzip the archive that contains a BSP provided with QNX Hypervisor.
   b. Copy the source code from that BSP's **src/apps/hypervisor/demos/shmem-host/** directory to your BSP's **src/** directory.

   (QNX-supplied BSPs that support QNX Hypervisor already have the source code, you just need to build the application.)
2. Optionally, to modify the application, make the required changes to the source code in the BSP's **src/apps/hypervisor/demos/shmem-host/** directory.
3. Build and install the application. For example, in the new **src/apps/hypervisor/demos/shmem-host/** directory, run `make install` to copy the revised binary to **install/**.

   The binary is ready to include in your buildfile.

The `shmem-host` application expects to communicate with a peer via the shared memory virtual device vdev. In most cases, this peer is either `shmem-guest` or `shmem-linux`, running inside of a guest equipped with the appropriate vdev.

Page updated: August 11, 2025

Building Linux and Android guests

QNX hypervisors support Linux and Android guests, provided these guests are built for virtual hardware that the hypervisor can present to the guest in a VM.

Building Linux guests

The information below is only an outline of what you need to do to implement a Linux guest in a QNX hypervisor system. Refer to your Linux documentation for information about how to configure and build your Linux system.

When you build non-QNX guests to run in a QNX virtualized environment, you must build them for the right hardware architecture, and configure the VMs in which they'll run to match their expectations.

To implement a Linux OS as a guest, you need to do the following:

1. Get from your favorite Linux source a Linux build environment appropriate for building your Linux OS, and configure it.
2. Follow your Linux instructions to build the guest in your Linux work directory.
3. Write a VM configuration file (e.g., **linuxvm1.qvmconf**) that assembles in a VM the components your Linux OS will expect to find on the platform where it will run.

   For a sample file, see "Configuration file for VM hosting a Linux guest (example)" below.
4. When you have built your Linux guest, you may include it in a new hypervisor disk image, which you can transfer to your target.

Adding drivers, applications, and utilities to a Linux guest

To be useful in a virtualized environment, your Linux guest will probably need to include some additional drivers, applications, and utilities. For example:

- To use the VIRTIO hardware the hypervisor makes available to it, the Linux guest kernel must be configured to include the VIRTIO drivers (e.g., block, network). For more information, see your Linux documentation.
- To use the shared memory demo application in your Linux guest, you need to take the relevant source code from the board-specific BSP for your target platform and copy it into your work area used for building the Linux kernel. In QNX-provided BSPs that support QNX Hypervisor, this source code is found at **src/apps/hypervisor/demos/shmem-linux/**.
- If you'll use a virtual hardware watchdog, you must include in the Linux kernel a driver for the watchdog device: SP805 (ARM) or IB700 (x86), and include and configure the corresponding watchdog vdev in the VM that will host your Linux guest (see "Watchdogs" in the "QNX Hypervisor: Protection Features" chapter).

Making command-line arguments available to a Linux guest

You start Linux guests the same way you start QNX guests: start the VM, either through the command line, or by adding to the hypervisor host startup routine the instructions for starting the hosting `qvm` process instance (see "Starting and using guests").

When you start your Linux guest, however, you'll probably need to provide it with some command-line arguments. To do this, you can use the `qvm cmdline` configuration option to hand the startup arguments to the Linux kernel (see cmdline in the "VM Configuration Reference" chapter).

For example, we might use the following to tell the Linux kernel where to find the console:

- On ARM platforms, the following configuration:

  ```
  cmdline
      console=ttyAMA0 earlycon=pl011,0x1c090000
  ```

  points the Linux kernel to a virtual PL011 UART device located at `0x1c090000` in guest-physical memory. Don't forget to include this device in your VM configuration.
- On x86 platforms, this configuration:

  ```
  cmdline
      pmtmr=0 nolapic_timer tasks=standard pkgsel/language-pack-
  patterns=
      pkgsel/install-language-support=false console=ttyS0,115200n8
  reboot=t
  ```

  points to a virtual 8250 UART (`ttyS0` is Linux's device name for an 8250 UART device). Don't forget to include this device in your VM configuration.

Booting Linux guests

To boot from disk, a Linux guest needs to know in which partition the root filesystem is installed (i.e., the filesystem that gets mounted at location **/**). When you have this information, you can specify it with a command-line argument in the VM configuration file. For example:

```
cmdline
    root=/dev/vda1
```

where `vda1` refers to the first `virtio-blk` vdev entry in the configuration for the VM.

If you use a `virtio-blk` device for your disk, remember that in Linux these devices present as **/dev/vda***, **/dev/vdb***, etc. If your device is a pass-through device, you'll need to know its device entry (probably **/dev/hda***) and make it known to the Linux guest so it can boot.

Configuration file for VM hosting a Linux guest (example)

Below is an example of a ***.qvmconf** file for a Linux guest. Note the use of the `reserve` and `rom` options for the regions where OSs designed to run on x86 platforms expect to find a VGA device and the BIOS.

```
# A minimalist VM configuration for a Linux guest on an x86 system
ram 0,0xa0000
reserve loc 0xa0000,0x20000
rom 0xc0000,0x40000
ram 1m,1023
cpu
cpu
load ./linux
initrd load ./initrd.gz
cmdline "pmtmr=0 nolapic_timer tasks=standard pkgsel/language-pack-
patterns=
    pkgsel/install-language-support=false console=ttyS0,115200n8
reboot=t
    root=/dev/vda1"
vdev ioapic
        loc 0xf8000000
        intr apic
        name myioapic
# This vdev entry corresponds to the console
# location specified in the above command line
vdev ser8250
        intr myioapic:4
vdev timer8254
        intr myioapic:0
vdev mc146818
        reg 0x0b,0x02
vdev shmem
vdev virtio-net
```

Building and including Android guests

To build an Android guest, you need:

- an Android build environment on your development host
- an Android kernel

Follow the outline instructions for building a Linux guest, adapted for your Android OS.

NOTE:
Remember that your Android guest must be built for the architecture and board you're using, and that your hosting VM must be configured to match the guest's expectations.

Page updated: August 11, 2025

Methods of building a QNX Hypervisor system

Building a QNX Hypervisor system entails adding the necessary hypervisor components to a QNX OS system, then building the host and guests using the appropriate build environments.

Building a hypervisor system based on a BSP

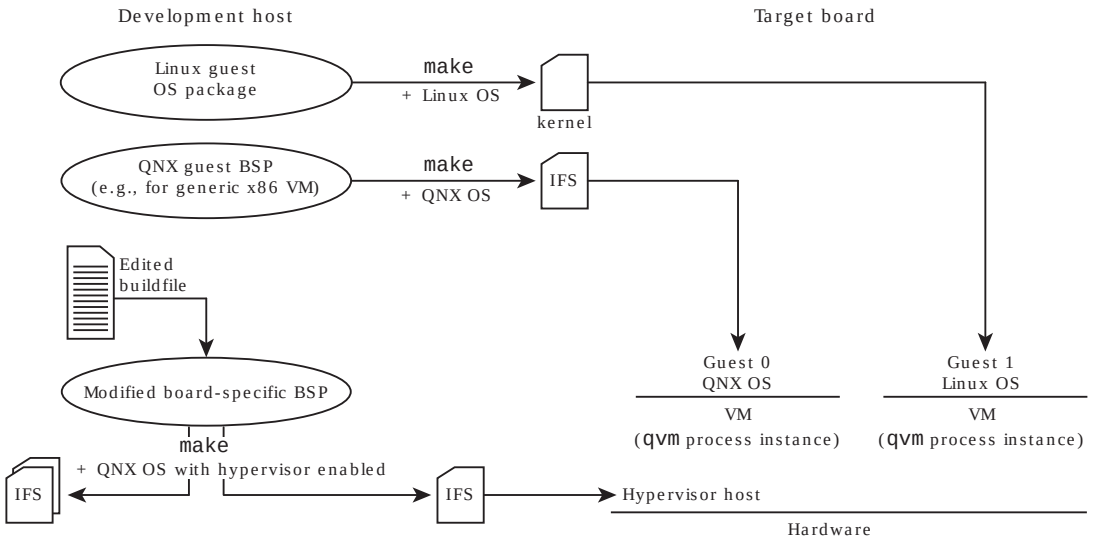
The QNX-supplied BSPs that support QNX Hypervisor include the necessary hypervisor functionality. (For a list of these BSPs, go to the release notes for your version of QNX Hypervisor.) To build a hypervisor host image, you use the hypervisor variant of the buildfile that comes with these BSPs, with a different `make` command. If no QNX-supplied BSP that supports QNX Hypervisor is compatible with your hardware, you must modify one of the buildfiles that comes with the BSP that supports your hardware to include the hypervisor components.

The overall build process is:

1. Configure the proper environments for building the hypervisor host and any guests you plan to build.
2. Build the guest OS images. If your project includes the guest content (e.g, IFS, VM configuration files) in the host IFS, this content has to be available before you build the host IFS. Even if guest content is stored on a system disk instead, QNX recommends that you create the guest first.
3. If your BSP does not already support QNX Hypervisor, add the hypervisor components to the buildfile. These components include the `qvm` binary, its supporting **vdev-*.so** files, one or more VM configuration (***.qvmconf**) files, and the `smmuman` binary and its supporting libraries and configuration files.
4. Build the host IFS.

   If you use the hypervisor variant buildfile that QNX Hypervisor provides, you must build the host IFS with the command `make hyp`.
5. If it's applicable in your environment, build a bootable disk image. QNX-supplied BSPs that support QNX Hypervisor include the disk image as a target. For other BSPs, you can add disk image building functionality, if needed.
6. If you're using a disk image for your system, transfer it to the target hardware board. Otherwise, transfer your host IFS to the board.

Figure 1QNX hypervisor host and guest components used in creating a hypervisor system based on a BSP



Using mkqnximage to build a QNX Hypervisor system

You can use the `mkqnximage` QNX OS virtual machine image generator to build a QNX Hypervisor system, which can be useful for both demonstration and development purposes.

Make sure that you have installed the QNX SDP virtualization drivers (these drivers are included by default when you install QNX Hypervisor).

The overall build process is:

1. To build the guest OS, run `mkqnximage` with the options that are appropriate for your project and specify `qvm` as the type (i.e., `--type=qvm`).
2. To build the host OS, in a separate directory from where you built the guest OS, run `mkqnximage` with the `--qvm` and `--guest` options. For example:

   ```
   mkqnximage --qvm=yes --guest=path_to_guest
   ```

where `--guest` specifies a directory in which you ran `mkqnximage` to build a guest image. The `--guest` option only works for guests that were created using `mkqnximage`.

You can use `mkqnximage` snippet files to include additional guests or ones other than those generated by `mkqnximage` (e.g, update **local/data_files.custom** to include the required files).

3. Use one or both of the following methods to test your hypervisor system:
   - For a basic test as to whether your hypervisor system works, after it has booted, run `qvm-check`.
   - Examine the contents of **/proc/config** for details about the status of the hypervisor on the host system. For example, to display just the status of QNX Hypervisor:

     ```
     # grep virtualization /proc/config
     virtualization:enabled
     ```

     (You can use `cat` to display the full contents of **/proc/config**.)
4. If you used `--guest` to include a guest, you can use the script **/data/hypervisor/start_guest** to start it.

For more information, go to the `mkqnximage` entry in the QNX OS *Utilities Reference*.

Page updated: August 11, 2025

Building QNX guests

A QNX OS system built to run as a guest in a QNX hypervisor environment uses a BSP, which supplies the architecture-specific and board-specific components.

For both ARM and x86 platforms, you need a hypervisor guest BSP to build a QNX guest. If you add new devices to the guest, you probably need to add the appropriate drivers to the BSP, and rebuild the guest with it.

NOTE:
When building QNX guests, be sure to use the architecture-specific *guest* BSP for your guest's OS version (e.g., QNX OS 8.0) and your target board's architecture, *not* the board-specific BSP for your host system's platform.

In most cases, building a QNX guest requires only building a bootable image, as you would for a non-virtualized environment.

When you install QNX Hypervisor, the guest BSPs for all supported architectures are included in the installation, so your development host will have all of the components needed to build the QNX guest.

To modify and build the guest, after you have set up your build environment on your machine:

1. Make sure the `virtual` attribute is present in the bootstrap section of the guest's buildfile, so that it knows the VM architecture on which it will run. For example, for ARM you need:

   ```
   [virtual=aarch64le,elf ...
   ```

   and for x86 you need:

   ```
   [virtual=x86_64 ...
   ```

   For more information about QNX buildfiles in general, see the "OS Image Buildfiles" chapter in the *Building Embedded Systems* guide.
2. Make the necessary changes to your guest OS. For instance, if you add pass-through devices to your guest and these devices require drivers from the board-specific BSP, copy these drivers into the appropriate locations in the guest BSP's **prebuilt/** directory.
3. From the guest BSP's root directory, run `make`. This uses the default buildfile to generate the guest IFS (for example, **images/guest-1/qnx800-guest-1.build**).

When the guest build succeeds, you can include the guest IFS in your hypervisor host system and transfer it to your target.

Page updated: August 11, 2025

Hypervisor disk images

If you make changes to the hypervisor host or the guests, you need to create a new bootable disk image with the hypervisor system and transfer it to your target.

Disk image partitions

If your BSP came with QNX Hypervisor, the hypervisor disk image includes the hypervisor host in a bootable partition and the guests in a data partition:

bootable partition
: Type 11 (DOS). Includes at least one bootable IFS (usually several) for the hypervisor host.

  You can use this partition to copy files to this disk image from any development host that can access a DOS filesystem. This is useful if you ever need to transfer files from your development host to your target.

data partition
: Type 177 (QNX PowerSafe). Includes an IFS for each guest in the reference image, and at least one `qvm` configuration file for each VM for these guests.

For other BSPs, you can create a similar partition configuration. For information about building a disk image with multiple partitions, refer to the "Building an OS image" section in the *Building Embedded Systems* guide, or the `diskimage` entry in the *QNX OS Utilities Reference*.

Changing partition sizes

If you need to change the sizes of your partitions, before you build the hypervisor system, you can adjust the disk configuration in the **disk.cfg** and the **part_*.build** files. These are found in the **images/** directory of the board-specific BSP for your target platform.

Page updated: August 11, 2025

Transferring the disk image

After you have prepared your target board, you must transfer your QNX hypervisor system image onto it. At a minimum this system includes a hypervisor host; in most cases it will have a hypervisor host and at least one guest.

About removable media

The hardware platforms on which you can run a QNX hypervisor system support a variety of removable media, including USB keys, SD cards, and SATA drives. Not all supported hardware platforms support the same removable media. This section provides instructions for USB keys and SD cards, because each supported platform supports at least one of these types of media.

The images expand to about 2 GB when extracted. The following removable storage types are recommended:

- SD or micro SD card, depending on the board: 8 GB Class 10
- USB key: 8 GB USB 3.0 flash drive

If your hardware platform has a SATA drive, you can use it instead of a USB key or SD card.

CAUTION:
Don't write to your storage medium's partition file (e.g., **/dev/sdb1** or **/dev/rdisk3s1**). Write to the raw device file.

USB keys

You can use a USB key with any hardware platform supported by the hypervisor. If you do so:

- Don't use a USB key with security features.
- If you have problems with your USB key, try reformatting it to a FAT32 filesystem, then converting the partition type to GPT. On a Windows system, you can use the MiniTool Partition Wizard Free tool to make these changes. Linux systems have their own tools, such as `diskutil`.

SD cards

If your platform supports SD or micro SD cards, we recommend UHS-I cards for better read/write performance. These cards can be identified by a “U” with a number “1” inside it, as shown below:

Figure 1The UHS-I identifier.

Linux

On a Linux host system, use these command-line instructions to copy an image to removable storage:

```
sudo dd bs=1024k if=base_dir/diskimage of=/dev/sdb
```

where *base_dir* is your hypervisor project working directory, *diskimage* is your hypervisor system disk image, and the USB key appears on your host system as **/dev/sdb**.

This command causes the `dd` utility to write data to the removable storage in chunks of 1 MB at a time.

NOTE:
The device name shouldn't include a partition suffix. For example, do *not* use **/dev/sdb1**. However, on some Linux variants, the device name may be **/dev/mmcblk0**.

Windows

To copy a disk image from a Windows host to a target, you need Win32 Disk Imager installed on your host system. If you don't have this utility, download it from this site, then install it: http://sourceforge.net/projects/win32diskimager/.

On a Windows system, to copy an image to removable storage:

1. Run the Win32 Disk Imager.
2. Browse to the location where you placed the image, and click **Open**.
3. Click **Write** to write the **.img** file to your USB key.

4. Click **Yes** to begin the process of writing the image. When it's complete, you'll see the message “Write successful.”
5. Click **OK**, then exit the Win32 Disk Imager.

Page updated: August 11, 2025

ACPI tables and FDTs

QNX hypervisor VMs make Advanced Configuration and Power Interface (ACPI) tables and Flattened Device Trees (FDTs) available to their guests.

In a QNX hypervisor system, devices available to a guest are specified in the configuration for its VM (i.e., its hosting `qvm` process instance). If the guest requires ACPI tables or an FDT in order to enumerate the devices available to it, you can modify ACPI tables and FDTs and have the corresponding `qvm` process instance load them into the guest's memory so the guest will find them when it starts.

ACPI tables (x86)

Guests running in QNX hypervisor VMs on x86 platforms can access ACPI tables for their VMs. From the guest's perspective, these tables are in the same location in the VM as they are in the hardware; that is, if the tables are at location `0x12340000` in host-physical memory, the guest will find them at location `0x12340000` in guest-physical memory. Check the specifications for your board for the location of the ACPI tables.

You can also create your own ACPI table to supplement the tables supplied by the board firmware and the hypervisor, and use the VM configuration `load` option to load it into your guest's memory. For example:

```
acpi load ./acpi_foo
```

will cause the `qvm` process instance to load the **acpi_foo** file into guest memory as an ACPI table (see "load" in the "VM Configuration Reference" chapter).

FDTs (ARM)

Guests running in VMs on ARM platforms can use FDTs to access devices. The way to configure an FDT in a VM is to provide an *FDT overlay*, which defines only the device properties that you want to override. This approach is taken for the following reasons:

- There is no need to include the parts that `qvm` generates, such as CPU information, the interrupt controller, or vdevs.
- Changing vdev properties such as `compatible` is easy. This is especially useful for the `shmem` vdev. You can define different `compatible` values for separate instances of this vdev to expose different device interface names in the FDT. This way, the guest OS can assign distinct drivers to interact with the different `vdev shmem` instances (and hence, they can be used for different purposes).
- Pass-through information can be inferred by `qvm` which removes the need to duplicate it between the FDT and the `qvm` configuration, thereby removing the chance for contradictions.

In this version of QNX Hypervisor, the FDT overlay must be provided to `qvm` in a device-tree binary (**.dtb**) file. To generate this binary file you must use the Device Tree Compiler (`dtc`) utility version 1.7 or higher. The input to this compiler is a device-tree source (**.dts**) file. This source file is not expected to define a full tree starting at a single root, but rather a collection of fragments that overlay specific nodes in a pre-existing tree which `qvm` supplies. For examples of files that define FDT overlays, go to our GitLab Repository at https://gitlab.com/qnx/hypervisor and look for the sample FDT projects.

NOTE:

- For more information about FDTs, see www.devicetree.org.
- A `dtc` utility is available from https://git.kernel.org/cgit/utils/dtc/dtc.git; a *Device Tree Compiler Manual* can be found at web.mit.edu/freebsd/head/contrib/dtc/Documentation/manual.txt.

Page updated: August 11, 2025

Configuration

A QNX hypervisor, its `qvm` processes that create the VMs in which guests run, and these guests must be configured to work together.

For information about the nomenclature we use for filenames of configuration files and various host components, see "Filenames" in the "Understanding QNX Virtual Environments" chapter.

Configuration and implementation limits

The following limits apply to QNX hypervisor systems:

- Maximum number of VMs: 16
- Maximum number of vCPUs in a VM: usually 16, but hardware dependent (contact your QNX representative)
- Maximum number of users that may access a shared memory region: 16

Page updated: August 11, 2025

Configuring guests

Guest images in a QNX virtualized environment are configured in the same way as they are configured in a non-virtualized environment. For example, for QNX guests, use a buildfile.

> **CAUTION:**
> When you configure a guest, make sure its configuration aligns with the configuration of the VM in which it will run, just as you would have to make sure that an OS is configured properly to run on a specific hardware platform. This means the guest must match the VM in terms of architecture, board-specifics, memory and CPUs, devices, etc. For more information, see "Assembling and configuring VMs".

Configuration of any resources such as devices and memory regions that a guest needs, the interrupts delivered to the guest, and the host-domain vCPUs threads that run guest code is all done through the VM configuration file. The sections that follow explain how to configure each of these components.

Information about building guest images is given in "Building guests" in the "Building a QNX Hypervisor System" chapter. Information about starting guests is provided in "Starting a guest" in the "Booting and Shutting Down" chapter.

Guest resource types

For a guest, every resource exists in a specific location (or "space"). This space defines the resource type. In the VM configuration, the location entries consist of the common vdev option `loc` followed by a resource type identifier and other parameters specific to the resource type:

`mem:`*`address ...`*
: The location is a guest-physical address (intermediate physical address). The first parameter gives the physical address *as understood by the guest*.

  For a full explanation and examples of all parameters used with this identifier, see the first form of the `pass loc` option in the "VM Configuration Reference" chapter.

`io:`*`port ...`*
: The location is in the x86 I/O space. The first parameter is the port number.

  For an explanation of all parameters used with this identifier, see the first form of the `pass loc` option.

`pci:{`*`pci_vid/pci_did|pci_bus:pci_dev[.pci_func] ...`*`}`
: The location is a PCI device. The device can be specified either by the vendor ID and device ID, or by the bus, device, and optionally, function number.

  For explanations and examples of these two different ways of specifying the PCI device, see the second form and the third form of the `pass loc` option.

Default resource types

If no resource type identifier is specified in the device configuration, a suitable default is chosen. The most common default resource type is `mem:`, but the default changes according to the vdev. For example, the default resource type for VIRTIO devices is `pci:`, though these devices can also be specified as `mem:` resources. See the descriptions of individual devices for each device's default resource type.

Guest interrupts

A guest interrupt is specified by an `intr` entry in the VM configuration. Depending on the platform, this entry can have two parts:

1. the guest device interrupt controller name, which is specified by the vdev `name` property
2. the interrupt controller input line that is asserted when the device wishes to raise an interrupt, which is specified by the number given as the part of the `intr` argument

Information about these two properties is given in "Common vdev options".

On x86 platforms, the Local component Advanced Programmable Interrupt Controller (LAPIC) hardware is automatically supplied. There is no need to specify a vdev for it, and guest vdevs that feed interrupts to the LAPIC hardware should simply specify `apic` for their interrupts; no input line number needs to be stated.

For example, the following creates a virtual I/O APIC device and a ser8250 serial device on an x86 system:

```
vdev ioapic
    intr apic
    name myioapic
vdev ser8250
    hostdev >-
    intr myioapic:4
```

On ARM platforms, the Generic Interrupt Controller (GIC) hardware is automatically supplied; it is not necessary to specify this vdev. You can still specify it if you want to change its option values, including the input line that gets asserted (for details, see the vdev gic reference). The default name for guest devices that feed interrupts to the GIC is "gic", but you can use the vdev's `name` property to change this.

The following example creates a virtual PL011 device on an ARM system:

```
vdev pl011
    loc 0x1c090000
    intr gic:37
    name mygic
```

Controlling guest behavior through vCPUs

Many OSs auto-detect functionality offered by the underlying CPUs. For guest OSs in a hypervisor system, you should usually configure a vCPU to run only on pCPUs (cores) of the same type. For details on doing so, see the `cpu cluster` option. If you want to run a vCPU on different core types, ensure you know which CPU features the guest will use and you restrict the vCPU to a cluster of pCPUs that support these features.

For the vCPU priority, you can set it in the VM configuration and the underlying `qvm` process will apply this priority when starting the guest. Afterwards, there's no `qvm` mechanism for changing it. In the hypervisor host, you can adjust the priority of any thread, including a vCPU thread, via the QNX OS mechanisms such as *ThreadCtl()*. It is best practice that the vCPUs in a given VM be configured with the same thread priority, whether at startup or any time afterwards.

Also, note that thread priorities within a guest are distinct from those visible to the host scheduler. If this scheduler has a low priority host thread that is ready to run and this thread's priority is higher than the vCPU's priority, it does not matter that inside the guest it is running a high priority guest thread. For further explanation, see the "Scheduling" section in the "Understanding QNX Virtual Environments" chapter.

Page updated: August 11, 2025

Configuring the hypervisor host

The hypervisor host domain is configured through a *buildfile*, exactly like a QNX OS image.

For details about how to write QNX buildfiles, including buildfile structure, contents, and syntax, see the "OS Image Buildfiles" chapter in the QNX OS *Building Embedded Systems* guide.

The buildfile for a hypervisor host differs from buildfiles for standard QNX OS images in the following sections:

Flag for enabling hypervisor features at startup

The QNX OS microkernel includes built-in support for virtualization and, thus, QNX hypervisor. To enable the hypervisor features, pass the `-Q enable` flag setting into the startup program. This setting enables the best supported hardware virtualization mode. For AArch64 platforms, you can explicitly specify the exception level (EL) at which the host should run. This is done by modifying the bootstrap portion of the buildfile:

```
[image=0x40100000]
[uid=0 gid=0]
[virtual=aarch64le,raw] boot = {
   [+keeplinked] startup-iMX8QM -P 1 -Q enable,el2-host -W -vv
   [+keeplinked]
PATH=/proc/boot:/sbin:/bin:/usr/bin:/opt/bin/sbin:/usr/sbin
   LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib
procnto-smp-instr -ae -v
}
```

In the example above, `el2-host` is included to turn on Virtualization Host Extensions (VHE) support, which means the host runs at EL2 at all times. If you specify `el1-host` instead, the startup enables virtualization without VHE and runs the host OS at EL1. Note that ARMv8.1 and later CPUs support `el2-host` and VHE; it would be unusual to specify `el1-host` for such systems.

On x86 platforms, you simply specify `-Q enable`; there's no need to request a specific exception level.

The startup program then passes the necessary information onto the microkernel process (`procnto`) so it can enable its hypervisor host mode and, hence, support the `qvm` processes that the hypervisor host creates.

For more information about this flag, see the `startup-*` entry in the QNX OS *Utilities Reference*.

Host domain code

The buildfile contains instructions to include and start the `qvm` process, and other code that will run in the hypervisor host domain. For example, the following command includes and starts a `qvm` process instance, which assembles a VM:

```
qvm
```

VM (`qvm`) configuration files

The buildfile includes the paths to the configuration files that different `qvm` process instances use when creating and assembling VMs, and indicates where to place these configuration files in the image. For example:

```
/vm/config/qnx80.qvmconf = guests/qnx80/qnx80.qvmconf
```

Virtual device (vdev) shared objects

The virtual device modules (**vdev-*.so**) are included in the shared objects list. For example:

```
vdev-8259.so
vdev-ser8250.so
vdev-timer8254.so
vdev-mc146818.so
vdev-virtio-console.so
vdev-virtio-blk.so
vdev-virtio-net.so
vdev-shmem.so
vdev-pckeyboard.so
vdev-ioapic.so
vdev-pci-dummy.so
vdev-hpet.so
vdev-pl011.so
vdev-vgpu-gvtg.so
vdev-progress.so
```

Unused components

The buildfile should include only components required by the hypervisor host.

You should remove all components that aren't required, so they don't get included in the hypervisor host image. For example, if a device (e.g., USB) is passed through to a guest, the host (or another guest) may not also have access to the device, so you should remove the driver from the hypervisor host buildfile (see "pass" in the "VM Configuration Reference" chapter).

Support for PCI pass-through devices

If your hypervisor system will pass through PCI devices, you must include the **pci_server-qvm_support.so** module in your hypervisor host image. For instructions on doing so, see the `pci-server` reference in the QNX OS *Utilities Reference*.

Page updated: August 11, 2025

Assembling and configuring VMs

To configure the VMs in which guests run, you can use command-line input or configuration files.

> **CAUTION:**
> Remember that the hypervisor host configuration takes precedence over any VM configuration. If a VM configuration attempts to allocate to the VM a resource allocated to the host, the `qvm` process creating the VM will exit with an error (see "qvm process exit codes" in the "Monitoring and Troubleshooting" chapter).

See the "VM Configuration Reference" chapter for descriptions and configuration instructions for the individual VM components, and the "Virtual Device Reference" chapter for the vdevs.

About configuring VMs

Every guest in a virtualized system requires a virtual machine (VM) in which it can run. Every VM is created by an instance of a `qvm` process, which is a hypervisor host domain process. To configure a VM, you configure the `qvm` process that creates it. Every `qvm` process instance should have its own configuration input. This input can come from one or more configuration files, the command line, or a combination of command-line input and files (see "Starting VMs").

The VM must match the guest that will run in it: architecture, board-specifics, memory, CPUs, devices, etc. You can think of VM configuration as building or *assembling* a hardware platform. But instead of assembling physical memory cards, CPUs, etc., you specify the virtual components of your machine, which a `qvm` process will create and configure according to your specifications.

The rules about where things appear are the same as for a real board:

- Don't install two things that try to respond to the same physical address.
- The environment your VM configuration assembles must be one that the software you will run (the guest OS and applications) is prepared to deal with.

In short, put into the VM configuration everything that the guest software requires be present on its "board", and put each component at the location where the guest expects to find it.

On ARM platforms, you have a bit more flexibility, because your guest can query the Flattened Device Tree (FDT) for information such as the location of the vGIC registers.

> **CAUTION:**
> When configuring a VM, always consider:
>
> - both the virtual machine you are creating and how the guest will run in it
> - that the guest may not always do the right thing
>
> A poorly considered configuration, especially the vCPU configuration, can have significant undesired consequences. For example, if you configure two vCPUs in a VM to share the same physical CPU (two vCPUs pinned to the same core), your guest may see unexpected behavior: timeouts, delays, spin loops never returning, etc. This problem may worsen if the two vCPUs have different priorities; the lower priority vCPU may never get a chance to run.

How a `qvm` process reads its configuration

When it starts, a `qvm` process reads its configuration input and uses this information to define the virtual hardware it creates for a VM. Thus, when you add, remove, or edit entries in a VM configuration, you are adding, removing, or editing virtual hardware in the VM. You are doing the equivalent of connecting or disconnecting a memory card or a serial device, etc. on a physical board.

A `qvm` process instance reads its input *in a single pass*, from start to finish. If you define a component more than once, your VM will be configured with either multiple instances of the same component (if multiple instances are permitted) or the last configuration information read for the component.

Configuration files

Since VM configuration is complex, we recommend you use configuration files. The VM configuration (`qvm` process) files provided with QNX hypervisors use the following filename nomenclature: ***guest*.qvmconf**, where *guest* is the guest OS (e.g., **qnx80**, **linux44**) that will run in the VM configured by this file.

See the following topics for information about:

- names of supporting files, including VM configuration files: "Filenames"
- defining specific VM components: "VM Configuration Reference"
- configuring specific vdevs: "Virtual Device Reference"
- starting a `qvm` process instance with or without a configuration file: "Starting VMs"

A sample configuration file is given in "VM configuration file example" in this chapter.

Location of configuration files

Typically, we place the VM configuration files at **/guests/*guestos*/**, where *guestos* identifies the guest OS (e.g., **/guests/qnx80/qnx80-guest1.qvmconf**).

This location is arbitrary. You can put the configuration files anywhere you want on the target, and maintain several copies for different configurations while you are developing your system. Just be sure to point each `qvm` process instance to the configuration file you want to use for that particular guest.

Page updated: August 11, 2025

VM configuration file example

The following example shows a VM configuration file for a guest system.

The `qvm` process assembles and configures the VM components specified by the options and their arguments in the configuration file:

```
system qnx8-x86_64-guest

ram 0xa0000
rom 0xc0000,0x40000
ram 1024M

# Specify "cpu" for each vCPU you want to assign to your guest
cpu
cpu
load /vm/images/qnx8-x86_64-guest.ifs

vdev ioapic
        loc 0xf8000000
        intr apic
        name myioapic
vdev ser8250
        intr myioapic:4
vdev timer8254
        intr myioapic:0
# An example network device that connects to a peer named vp0 in the
host.
# This peer is found at /dev/vdevpeers/vp0.
vdev virtio-net
        name ptp
        peer /dev/vdevpeers/vp0
# An example block device. This file has to be created somewhere;
# in this case, it's created in the IFS with dd.
vdev virtio-blk
```

We know that this configuration is for an x86-64 VM because 8259 devices and APIC are supported only on that architecture. For the vdevs that support it, the `name` option is defined to make the devices appear at **/dev/qvm/*system_name*/*vdev_name***. In this case, *system_name* is **qnx8-x86_64-guest**.

Page updated: August 11, 2025

VM configuration syntax

A VM (`qvm` process) configuration file is a human-readable, plain-text file.

When you start a `qvm` process to create a VM and run a guest, the `qvm` process reads the entire configuration information to know how to assemble and configure the VM.

General rules are:

- Everything that follows a number sign (`#`) in a line is a comment and is ignored, as are blank lines.
- Entries (other than comments) in a `qvm` configuration are either *options* or *arguments*.

Rules for options

The rules for options are:

- An option may affect contexts in one of the following ways:
  - The option establishes a context in which other options can be set; for example, the `vdev` option establishes a context in which a virtual device is defined through other options.
  - The option applies to a context; for example, the `sched` option following a `cpu` option applies to the current context established by the `cpu` option, thus `sched` contributes to the definition of the vCPU specified by `cpu`.
  - The option neither establishes a context nor applies to a context; for example, the `ram` option supports arguments setting the location and size of the memory allocated, but it supports no options.

  For more details, see "Contexts" below.
- Options that don't apply to a context specify a component to be included in the VM. For example, `cpu` instructs the `qvm` process to create a virtual CPU, and `load /vm/images/qnx8.ifs` instructs it to copy the contents of **/vm/images/qnx8.ifs** into the guest system address space.
- There is no default value for an unspecified option.
- Most (but not all) options require one *argument* (see "Rules for arguments" below, and the entries for individual options in the "VM Configuration Reference" chapter).
- In general, option sequence isn't important. It doesn't matter in which order you define vdevs, for instance (but see "Exceptions" below).

Rules for arguments

An argument must:

- immediately follow the option to which it applies
- be in the same file as the option to which it applies

Contexts

When the `qvm` process reads through its configuration file to assemble the VM it is creating, everything that follows an option that establishes a context is in this option's context. All further options apply to this option, until either a new option that establishes a context is encountered, or the end of file is reached. Essentially, a context is a block in the configuration file that groups together related options.

For example, in the following snippet each `sched` option applies to the preceding `cpu` option:

```
cpu
    sched 8 ram 32m
cpu
sched 6
```

Since `cpu` establishes a context, the context changes each time this option is encountered; what follows each `cpu` entry either applies to that option or creates a new context.

If an option is repeated in a context, the `qvm` process uses the last instance of the option in that context. For example:

```
cpu sched 8 cpu sched 6
```

creates one vCPU with scheduling priority 8, and one vCPU with scheduling priority 6. However, the following:

```
cpu sched 8 sched 6 cpu
```

creates one vCPU with scheduling priority 6 (`sched 8` is discarded), and one vCPU with the default scheduling priority (i.e., the scheduling priority at which the `qvm` process instance was started).

The order of the *different* options that follow the `cpu` option isn't significant, though. For example:

```
cpu sched 8 ram 32m
```

is equivalent to:

```
cpu ram 32m sched 8
```

NOTE:
The line breaks are to improve human readability and are ignored by the `qvm` process when it parses the configuration file.

Exceptions

There are some exceptions to the above rule that option sequence isn't important. These exceptions apply to the following components:

system
: If the `system` option is specified, it must be the *first* entry in a `qvm` configuration (see `system` in the "VM Configuration Reference" chapter).

ram
: Memory must be allocated before any option specifying a component that will use the memory. Thus, the `ram` and `rom` options must be specified before any options that refer to the guest memory. For example:

```
ram 32m
load /qnx8.ifs
```

is valid because the `ram 32m` has allocated 32 MB, into which the `qvm` process can load the IFS file. However, the following will fail:

```
load /qnx8.ifs
ram 32m
```

because no memory has been allocated, so the `qvm` process has nowhere to load the IFS file.

rom
: Same sequencing rule as with `ram`. Must be specified *before* any option refers to it.

PIC vdevs
: Any vdevs for Programmable Interrupt Controllers (PICs) must be specified before any other vdevs that reference them. For example:

```
vdev ioapic
        loc 0xf8000000
        intr apic
        name myioapic
vdev ser8250
        intr myioapic:4
```

is valid because `vdev ioapic` is specified before `vdev ser8250`, which references it. However, the following will fail:

```
vdev ser8250
        intr myioapic:4
vdev ioapic
        loc 0xf8000000
        intr apic
        name myioapicr
```

because `vdev ser8250` references `vdev ioapic` before this vdev has been specified.

NOTE:
You should always name your system, then allocate RAM and ROM right at the beginning of your `qvm` configuration.

Textual substitutions

As it reads through its configuration information, a `qvm` process instance performs textual substitutions when it encounters the following character sequences:

$env{*var*}
: Replace the text string with the value of the *var* environment variable.

$asinfo_start{*name*}
: Replace the text string with the starting address of the system page *asinfo* entry specified by *name*.

$asinfo_length{*name*}
: Replace the text string with the length of the system page *asinfo* entry specified by *name*.

You can use this textual substitution to make your configuration more robust. For example, you can pass a region of memory to the guest without specifying the host address for the memory in the VM configuration, as follows:

1. Have the startup for the hypervisor host allocate the memory and record its location with a system page *asinfo* entry such as `guestmem1`.
2. Use textual substitution to place the information from the system page *asinfo* entry called `guestmem1` into the `qvm` configuration:

   ```
   pass loc
   0x10000000,$asinfo_length{guestmem1},rw=$asinfo_start{guestmem1}
   ```

Addresses and lengths resulting from the expansion of `$asinfo_start{`*name*`}` and `$asinfo_length{`*name*`}` must obey the same restrictions as if they were manually configured in the VM. That is, the host-physical starting address of the memory allocated by the startup must be aligned to a page boundary, and the length must be a multiple of the page size on the host system. If not, the guest may experience I/O failures.

Similarly, you can use `$env` to put parameters in a configuration file. Suppose you have the following in a `qvm` configuration file (e.g., **myconfig.qvmconf**):

```
vdev ser8250 hostdev $env{HOST_DEV}
```

You could then define the *HOST_DEV* value that the vdev ser8250 `hostdev` option gets set to, then start the `qvm` process instance, as follows:

```
export HOST_DEV=/dev/ser3
qvm @myconfig.qvmconf
```

Note that when using `$asinfo_start`, pass the leaf name only, and *not* the full path. For example, the following is incorrect:

```
pass
  loc
0x10000000,$asinfo_length{guestmem1},rw=$asinfo_start{/foo/guestmem1}
```

but the following is correct:

```
pass
  loc 0x10000000,$asinfo_length{guestmem1},rw=$asinfo_start{guestmem1}
```

For more information about the system page *asinfo* data structure array, see the “System Page” chapter in *Building Embedded Systems*.

About notation

The default notations (no prefix needed) for specifying memory addresses and sizes are:

- address in memory – hexadecimal
- size or length of memory region – decimal

If you prefer to write a memory address or region size with a non-default notation, use a prefix to specify the notation:

- decimal – `0d` (e.g., `0d1234`)
- hexadecimal – `0x` (e.g., `0x4D2`)

You can use size multipliers: “K”, “M”, “G” (or “k”, “m”, “g”) in the address and length arguments; for example: `4K,1k` is equivalent to `0x1000,0x400`. (Remember: the size multipliers are *decimal* multipliers, so `4K` is `4 x 1024 = 4096` or `0x1000`.)

NOTE:

- Other numeric configuration values are specified in decimal.

- We recommend that, to avoid confusion, you specify the prefix when using hexadecimal values.

Page updated: August 11, 2025

Copyright notice

BlackBerry Limited
2200 University Avenue E.
Waterloo, Ontario
N2K 0A7
Canada

Voice: +1 519 888-7465
Fax: +1 519 888-6906
Email: info@qnx.com
Web: https://www.qnx.com/

August 11, 2025

Page updated: August 11, 2025

Monitoring and Troubleshooting

This chapter describes some tools and techniques you can use to monitor and troubleshoot your hypervisor system.

A QNX hypervisor comprises the QNX OS microkernel and one or more `qvm` process instances. This means that our hypervisor product has the same debugging, monitoring, and troubleshooting capabilities as QNX OS. Thus, all tools—QNX Momentics IDE, command line, and third-party—and techniques available for working with the QNX OS are also available for working with the hypervisor host or QNX guests. For Linux guests, use your Linux tools (GDB TraceCompass, etc.).

Page updated: August 11, 2025

qvm process exit codes

When a `qvm` process exits, the shell exit status indicates the reason for the exit.

The possible `qvm` process exit codes include:

No error

- 0 – the `qvm` process ran as designed, and terminated in response to a request to shut down or restart that is part of normal operations

VM- and vdev-related errors

- 3 – the guest was terminated by an action analogous to its power being cut (e.g., its hosting `qvm` process instance received a SIGQUIT signal)
- 4 – watchdog-triggered termination
- 5 – unsupported operation
- 6 – unexpected vdev error
- 7 – unexpected error
- 32 to 63 – the exit code is set by the user; for example, to indicate that a specific vdev caused the exit (see the *Virtual Device Developer's Guide*)

Fatal errors

- 64 – fatal error before the guest started (e.g., out of memory)
- 65 – fatal configuration error
- 96 – fatal error after the guest started

More information

Exit/error codes for `qvm` are defined in the **guest.h** public header file.

For information about:

- DSSs – "Design Safe States" in the "QNX Hypervisor: Protection Features" chapter
- `qvm` process exit and error codes – the **guest.h** chapter in the *Virtual Device Developer's API Reference*
- filtering messages from `qvm` process instances – "logger" in the "VM Configuration Reference" chapter
- getting a dump – `dump` in the "VM Configuration Reference" chapter
- the hypervisor while it is running and after it has quit – `pidin` and `slog2info` in the QNX OS *Utilities Reference*

Page updated: August 11, 2025

Getting a guest dump during a crash

You can configure your guest and the watchdog vdev in its VM to dump the guest's state in the event of a crash.

This dump will provide you with the guest's state at the time that the watchdog intervened: the guest memory and its CPU registers (i.e., the vCPU registers in the hosting `qvm` process instance's vCPU threads).

To cause the guest to dump during a crash:

1. In the guest, add a watchdog kicker (e.g., `wdtkick`) and configure it to kick the virtual watchdog in the VM at the appropriate intervals (see `wdtkick` in the QNX OS *Utilities Reference*).
2. Add the appropriate watchdog vdev shared object to your hypervisor host's buildfile, and add the watchdog vdev to the hosting VM's configuration. QNX hypervisors include two watchdog vdevs: `vdev wdt-sp805` (ARM) and `vdev wdt-ib700` (x86).
3. Configure the watchdog vdev to dump if the guest fails to kick it within the specified time interval (see "Watchdogs" in the "QNX Hypervisor: Protection Features" chapter).
4. In the configuration for the hosting VM, specify the `dump` option so that it writes the hosted guest's dump to a file (see `dump` in the "VM Configuration Reference" chapter).

Page updated: August 11, 2025

OS-VM configuration mismatch

A guest may fail to start or not perform as expected if the VM in which it is running is not configured with the vdevs the guest OS expects to find.

Just as an OS running on a non-virtualized system expects certain physical devices to be present on the board, a guest OS expects certain vdevs to be present in its VM. For example, a QNX guest on an x86-64 platform expects its VM to include the standard devices (as vdevs) that would be found on a physical board. These include:

- Either hpet or timer8254; if hpet is used, its options override any conflicting timer8254 parameters
- ioapic
- pckeyboard; if this vdev is missing, you can start the guest, but you won't be able to return to the host after a guest shutdown
- ser8250

In a non-virtualized system, the device driver in an OS must match the hardware device on the physical board. In a virtualized system, the device driver in the guest must match the vdev. If there's a mismatch, the guest won't be able to use the vdev.

For more information, see “Devices” in the “Understanding QNX Virtual Environments” chapter.

Page updated: August 11, 2025

Troubleshooting strategies

There are numerous ways to get information about the different parts of your hypervisor system.

For guests, use the same debugging tools you would use for the OS and its applications in a non-virtualized system, by connecting to a TCP/IP stack running within the guest. For example, if you run a QNX guest and it starts such a stack along with the `qconn` service, the QNX Momentics IDE can connect to the guest through TCP/IP and do application-level debugging. For information about debugging with the IDE, see the "Debugging Applications" chapter in the IDE *User's Guide*.

For hosts, the troubleshooting ways include:

- Add the `logger` option to the `qvm` configuration (see logger in the "VM Configuration Reference" chapter) to log certain message types.
- Use tracelogging (see "Getting hypervisor trace information") to examine trace events from a VM.

A typical approach to troubleshooting would be to use the `logger` option to gather general information about what a `qvm` process is doing (and, by indirection, what a guest is doing), then to use tracelogging for more detailed analysis.

If you need to keep more than the hypervisor's default 4K of slog information, you can use the `set` option to set the *slog-buffer* variable to the right buffer size (see *slog-buffer* in the "VM Configuration Reference" chapter). You can also send the output to your console.

Page updated: August 11, 2025

Getting hypervisor trace information

You can use the `tracelogger` utility and the QNX Momentics IDE System Profiler to capture and examine trace events from a VM.

A `qvm` process instance is a process in the hypervisor; this process creates the virtual machine (VM) in which a guest OS can run. For the hypervisor, guest OSs are blobs about which it can know nothing. Thus, a trace view of a `qvm` process instance can show what a `qvm` process is doing; it doesn't (and can't) show anything about what is going on inside the guest.

However, the microkernel in the hypervisor host and the `qvm` process are instrumented with trace events (see "The Instrumented Microkernel" in the QNX OS *System Architecture Guide*), and by looking at what the hypervisor is doing, you can learn a great deal about what a guest is doing or trying to do.

For more information about logging trace events, see the:

- QNX Momentics IDE *User's Guide*
- QNX OS *System Analysis Toolkit (SAT) User's Guide*
- QNX OS *Utilities Reference*, in particular `tracelogger` and `traceprinter`

About `qvm` threads

The threads in a `qvm` process instance include:

main thread
: Thread 1, the `qvm` process instance's main setup thread.

vCPU threads
: Every vCPU is a `qvm` thread. There is a thread for every vCPU defined by a `cpu` option in the configuration (***.qvmconf**) file for the VM. These threads control the VM, and are the source of most of the trace events you will see.

vdev threads
: Created by vdevs, these are present only if a vdev creates them. The vdev that creates a thread names that thread.

  These threads are loaded as part of shared objects (***.so**). For example, a graphics vdev shared object may use one worker thread for managing a virtual cursor, another for updating a display, etc.

You may also see additional worker threads in the `qvm` process instance.

Page updated: August 11, 2025

Updating hypervisor event trace descriptions (optional)

The QNX Momentics IDE System Profiler includes text descriptions of `qvm` events, so you don't need to do anything special to show these events.

Copying an event-descriptions file into your IDE project

If you don't have the `qvm` event descriptions for the IDE System Profiler, or if you need to update the `qvm` event descriptions to newer versions, you can create an event-descriptions file and copy it into your IDE project. To do this:

1. Get or create an events description file (e.g., **qvm_events.xml**), and save it to a convenient location.
2. Copy this file into the root directory of the IDE project in which you will use the IDE's System Profiler.

See "Example of a descriptions file" below for an example of a `qvm` event-descriptions file.

Loading the event-descriptions file

After you have **qvm_events.xml** in your project's root directory, you need to load the file into the IDE:

1. Start the QNX Momentics IDE.

   For instructions on doing so, refer to the "Starting the IDE" section in the *IDE User's Guide*.
2. Switch to the System Profiler perspective and select the Project Explorer tab.
3. Right-click the target system for your QNX hypervisor target, then select **Import**.
4. In the **Import** wizard, select **General** > (and then)**File System**, then click **Next**.
5. Next to the **From directory** field, click **Browse** to open the file selector.
6. Browse to the directory containing your **qvm_events.xml** file, select this file, then click **Finish**.
7. Select **Window** > (and then)**Preferences** in the menu area at the top, then **QNX** > (and then)**System Profiler** > (and then)**User Event Data** in the resulting dialog box.
8. Select **Browse...** and in the file selector that opens, select your **qvm_events.xml** file, then click **OK**.
9. Exit the QNX Momentics IDE, then restart the IDE and connect to the target board running your QNX hypervisor system.
10. Load any ***.kev** file into the System Profiler. You should see the `qvm` process trace events, including the Class 10 events (see "Hypervisor trace events").

    For more information about this IDE tool, see the "System Profiler" section in the *IDE User's Guide*.

NOTE:
You need to do this for only one project on your development seat (i.e., license type meant for development). When you have done this once, the System Profiler will be able to use the **qvm_events.xml** file and its descriptions for all other projects.

Example of a descriptions file

Below, for reference, is a copy of the content of an XML event-descriptions file (**qvm_events.xml**):

```
<eventdefinitions>
    <!-- Guest Entry -->
    <datakey format="%8u1x guest_ip">
        <event class="10" id="0" />
    </datakey>
    <!-- Guest Exit -->
    <datakey format="%4u1x status %4u1x reason %8u1x
        cc_offset %8u1x guest_ip %8u1x payload">
        <event class="10" id="1" />
    </datakey>
    <!-- vCPU create -->
    <datakey format="%4u1x vcpu_id">
        <event class="10" id="2" />
    </datakey>
    <!-- Raise interrupt -->
    <datakey format="%4u1x vcpu_id %4u1x interrupt">
        <event class="10" id="3" />
    </datakey>
    <!-- Lower interrupt -->
    <datakey format="%4u1x vcpu_id %4u1x interrupt">
        <event class="10" id="4" />
    </datakey>
    <!-- Create timer -->
    <datakey format="%4u1x timer_id %4u1x cpu_id">
        <event class="10" id="5" />
    </datakey>
    <!-- Timer fire -->
    <datakey format="%4u1x timer_id">
```

Page updated: August 11, 2025

Hypervisor trace events

Most `qvm` trace events are Class 10 events. However, some `qvm` threads may also make kernel calls to the QNX OS microkernel; these will have corresponding kernel call events.

The following should help you interpret the data provided with Class 10 events from a hypervisor host.

ID 0 — guest entry

The guest is executing code at *guest_ip* address.

Arguments:

guest_ip
: The execution address, as seen by the guest.

NOTE:
The timestamp for the ID 0 event should *not* be used to calculate the time spent in the guest during an entry/exit cycle. For an explanation, see the note in the ID 7 event description.

ID 1 — guest exit

The guest has stopped executing code.

Arguments:

status
: Architecture-specific. On x86 platforms, a non-zero ID 1 *status* field indicates that the guest entry was aborted before guest was able to run.

reason
: Architecture-specific.

clockcycles_offset
: Offset between guest view of timestamp and host view of timestamp. Measured in clock cycles (see *ClockCycles()* in the *C Library Reference*). A negative number means the guest is behind the host; a positive number means the guest is ahead of the host.

  On x86 platforms, if the *status* field is non-zero, the guest didn't run, so the value of *clockcycles_offset* is stale (and therefore meaningless).

guest_ip
: The execution address, as seen by the guest.

hw_payload
: Architecture-specific.

NOTE:
For both ARM and x86 platforms, to know if the guest actually ran before it exited, check for the presence of ID 7 events between the ID 0 and ID 1 events (see "ID 7 — provide guest clock cycle values at guest entry and exit" below). If no ID 7 events are present, then the guest didn't run.

The timestamp for the ID 1 event should *not* be used to calculate the time spent in the guest during an entry/exit cycle. For an explanation, see the note in the ID 7 event description.

ID 2 — create a vCPU thread

A virtual CPU (vCPU) thread has been created to schedule the vCPU.

Occurs only once per vCPU in the lifetime of a `qvm` process instance. If you start your trace after the `qvm` process has started, you won't see these events.

Arguments:

vcpu_id
: vCPU-specific ID internal to hypervisor. The ID number as seen in the trace logs will be marked as `virtual cpu num`, where *num* is the vCPU number, starting at zero (0).

ID 3 — assert an interrupt to a guest CPU

An interrupt has been asserted on a vCPU (seen by the guest as a CPU). The VM in which the guest is running must be configured to deliver the interrupt to the guest. This configuration is specified in the ***.qvmconf** file for the underlying `qvm` process instance.

Arguments:

vcpu_id
: ID of the vCPU where the interrupt is being asserted.

interrupt
: The interrupt as seen by the guest (e.g., on x86-64 platforms, a vector number).

ID 4 — de-assert an interrupt to a guest CPU

An interrupt is being de-asserted on a vCPU (seen by the guest as a CPU).

Arguments:

vcpu_id
: ID of the vCPU where the interrupt is being de-asserted.

interrupt
: The interrupt as seen by the guest (e.g., on x86-64 platforms, a vector number).

ID 5 — create a virtual timer

A timer has been created for a vCPU.

Occurs only once per timer in the lifetime of a `qvm` process instance. If you start your trace after the `qvm` process has started, you may not see these events.

Arguments:

timer_id
: The value returned by *timer_create()*, which has been called by the `qvm` process instance.

cpu_id
: The ID of the vCPU where the timer was created.

ID 6 — trigger a virtual timer

A virtual timer has been triggered.

Argument:

timer_id
: The ID of the vCPU where the timer was triggered.

ID 7 — provide guest clock cycle values at guest entry and exit

Guest clock cycle values at guest entry and exit.

Arguments:

at_entry
: The guest clock cycles (*ClockCycles()*) value at guest entry.

at_exit
: The guest clock cycles value at guest exit.

The Cycles trace event records the amount of time spent in the guest between Entry/Exit trace events.

NOTE:
You can't draw meaningful conclusions from the difference between the timestamps of the Entry (ID 0) and Exit (ID 1) events. These events are emitted at the user level, and thus the vCPU thread can be preempted, migrated, interrupted, etc. between the time of the guest exit and re-entry.

This is especially true on a system where the host OS is heavily loaded by threads that are unrelated to the hypervisor's vCPU threads. Also, work is often done in a hypervisor-privileged context when the hypervisor prepares to enter a guest and when it handles a guest exit.

The Cycles trace event should be used because it includes only the time spent in the guest and excludes both the time that the hypervisor may not have been running and the time taken to perform the guest exit/re-entry processing.

For an example of how you can interpret the trace events emitted by a hypervisor to measure its overhead, see "Measuring the hypervisor overhead" below.

Kernel call events

The kernel calls you will see in trace logs for a hypervisor host will also include events that are not specific to the hypervisor, such as `MMap` and `MMunmap`. See the *System Analysis Toolkit User's Guide* for more information about kernel call trace events.

Measuring the hypervisor overhead

If you want to measure the hypervisor overhead, you must have an accurate indication of the time spent in the guest versus the time spent in the host. Here, we list some of the trace events emitted by the hypervisor during a guest entry/exit cycle, and show how to calculate the percentage of time spent in the guest.

In the pseudo-code example below, we provide an abstract and idealized vCPU loop running on a ARM target system:

```
begin
    HYP EL 0 - Guest Entry Trace Event
    HYP EL 0 - prepare to enter guest
    HYP EL 2 - enter guest
    GST EL 0/1 - running in guest
    HYP EL 2 - guest exited because the CPU needs the hypervisor to
intervene
    HYP EL 0 - Cycles + Guest Exit Trace Events
    HYP EL 0 - handle guest exit (may involve a vdev or emulation of
PSCI calls, etc.)
end
```

While the vCPU thread is running hypervisor code at EL 0, the host QNX OS sees it as an ordinary application and will handle interrupts and context switch between host threads. Some systems run other software on the host, at the same or higher thread priorities as the vCPUs, that's unrelated to what the hypervisor and guest are doing. If you take the difference between the guest exit and guest entry timestamps, you will include all time that is completely dependent on what the other software concurrently running on the host is doing. This other time is *not* hypervisor overhead.

The Cycles trace event provides the start and end times between which the guest ran during the entry/exit cycle. Subtracting the start time from the end time provides the actual time spent in the guest. If you want to know when those times were from the host's perspective, you can examine the "offset" value in the Exit trace event; this lets you convert the start or end time from the Cycles event into a timestamp that is meaningful for the host.

You can measure the hypervisor overhead as the percentage of time spent in the guest, as follows:

```
% time in guest = (time spent in guest) / (total thread RUNNING time)
```

Note that:

- Determining the amount of thread RUNNING time involves many other QNX OS trace events.
- Given the variation in overheads from one exit to another, you would usually do this calculation across a time interval spanning many entry/exit cycles.

Page updated: August 11, 2025

Comparing guest and host timelines

You can use information from trace events to compare timelines of events in the guest and the host.

The following is an example of trace event output that can be used to compare the guest and host at timestamp counts (TSCs) when a specific event occurs:

```
t:0x2ebe7b16b CPU:00 QVM :GUEST_ENTER guest_ip:0x00000000fe087700
t:0x2ebe7c20c CPU:00 QVM :CYCLES
at_entry:0x0000000068ed5b7f
at_exit:0x0000000068ed620e
t:0x2ebe7c6f7 CPU:00 QVM :GUEST_EXIT
status:0x00000000
hw_reason:0x07e00000
clockcycles_offset:0xffffffffd7d05a4eb
guest_ip:0x00000000fc4047f0
payload:0x0000000000000000
```

Where:

- `status:0x00000000` indicates that the guest entry was successful, so the *at_entry* and *at_exit* values are meaningful (see "ID 1 — guest exit").
- `at_entry:0x0000000068ed5b7f` is the value for the guest timestamp count *just before* the vCPU thread transitioned into the guest environment.
- `at_exit:0x0000000068ed620e` is the guest timestamp count *just after* the vCPU thread transitioned out of the guest environment.
- `clockcycles_offset:0xffffffffd7d05a4eb` is the clock-cycle offset between the guest view of the timestamp and the host view of the timestamp.

The host necessarily begins counting before the guest, so the host timestamp count will always be greater than the guest timestamp.

To obtain values that can be used for a meaningful comparison of when an event occurred for the guest with when it occurred for the host, you need to:

1. Construct a timeline for the guest with *at_entry*/*at_exit* event pairs.
2. Construct a timeline for the host with the same *at_entry*/*at_exit* event pairs, combined with the value in *clockcycles_offset* at the guest exit.

   As you construct your host timeline, note which event pair in the host corresponds to which event pair in the guest.

Anything that occurs between an *at_entry* and *at_exit* event pair in the guest will have occurred between the corresponding event pair in the host. For example, an event that occurred between the *at_entry* and *at_exit* event pair in the output above will have occurred in the host between:
`(0x0000000068ed5b7f - 0xffffffffd7d05a4eb) = 0x2ebe7b694`
and:
`(0x0000000068ed620e - 0xffffffffd7d05a4eb) = 0x2ebe7bd23`

NOTE:
Remember that:

- The value of *clockcycles_offset* is set during `qvm` startup and never changes.
- Any *ClockCycles()* value outside the *at_entry*/*at_exit* intervals will never be observed by a guest.

For more information, go to "Time" in the "Understanding QNX Virtual Environments" chapter.

Page updated: August 11, 2025

Viewing hypervisor activity

After the hypervisor guest has booted, you can use QNX Momentics IDE to open a command terminal to either the host or a guest and then view its resource usage and activity.

NOTE:
You need the `qconn` daemon running in any OS running on the target—the hypervisor host or any guest—that you want to interact with (for details, see the `qconn` entry in the QNX OS *Utilities Reference*).

Connecting and interacting with a guest

If your hypervisor host is connected to a DHCP server, you can get your guest's IP address as follows:

1. In your serial terminal on the target, enter the command `ifconfig`.

   The command should return a valid IP address for your `wm0` device.
2. Remember this address. It is the IP address for the hypervisor guest.

You can then use the QNX Momentics IDE to connect to the guest on the target board as follows:

1. Launch the IDE on your development host (for instructions on doing so, see the *IDE User's Guide* or the *IDE Release Notes*).
2. Switch to the QNX System Information perspective.
3. Right-click on the **Target Navigator** view, then select **New QNX Target**.
4. In the dialog box that appears, enter the guest's IP address, then press **Enter**.

You can now use the IDE tools to work with the guest. Here are a few things to try:

- Open the **Target File System Navigator** view to explore the guest's filesystem on the target. You should see the built-in RAM disk at **/dev/shmem**; this is a convenient place for storing temporary files.
- Open the **System Information** and **Process Information** views to see details about the processes running in the VM that hosts the guest. These processes make up the various applications the guest is running.
- Open the **System Resources** view to examine the memory usage by regions (e.g., heap, stack) and the total memory usage for a given process. For more information, see “System Resources” in the “Reference” chapter of the *IDE User's Guide*.
- Use the same **System Resources** view to examine the usage of individual CPUs by the guest's process threads. (The CPUs shown are actually vCPUs but the guest doesn't know that.)
- Start another shell window; SSH is preferred: open the **Target Navigator** view, select the target for your guest, then right-click to select SSH.

You can also start an SSH session in another terminal program (e.g., PuTTY), and log in as `userid root`, `password root`. (You can change these settings when you modify the hypervisor bootable image.) Then, you can interact with the guest through this session. For example, if you run `pidin info` you might see a line that starts with `CPU:X86_64 Release:8.0.0` and four processors listed; these processors appear real to the guest but are actually vCPUs created by the VM.

Connecting and interacting with the hypervisor host

To connect to the host, you can use the IP address of the target board, which you can find through `ifconfig` or `ipconfig`. The name of the interface for your target depends on your system setup.

The same steps used for connecting to the guest by launching QNX Momentics IDE and configuring a new QNX target apply here, except you'll use a different IP address.

After you've connected to the hypervisor host through the IDE, you can use the same views as described above to see resource usage and system activity. Some host-specific things to try are:

- Open the **System Information** and **Process Information** views to see all processes and threads in the hypervisor host domain, including the `qvm` processes, the vCPU threads in each VM, and any threads the vdevs are running. This is useful to understand the CPU load that the hypervisor imposes on the target hardware.
- Open the **System Resources** view to monitor the memory usage of the `qvm` processes and the CPU usage of threads within these processes, including vCPU threads and other vdev-specific threads. The host knows only whether the vCPUs are running but cannot see inside the guest to know which guest threads are running on a given vCPU.

You can also launch an SSH session outside of the IDE (e.g., through PuTTY) and run a command such as `pidin info` to see system information about the OS and processors; in this case, the listed processors are the physical CPUs on your target.

Page updated: August 11, 2025

Gathering information about a virtual machine

QNX Hypervisor provides ways for gathering information about the VM running inside of a `qvm` process.

This runtime VM information is useful for systems integrators and anyone setting up guest OSs and drivers. Here we only summarize the mechanisms for getting this information. For details about the specific data that each mechanism provides, go to the corresponding note in the QNX Hypervisor GitLab Repository at: https://gitlab.com/qnx/hypervisor/tools-for-qnx-hypervisor/gathering-information-about-a-virtual-machine.

Getting guest status (SIGUSR1) and guest dumps (SIGUSR2)

You can send the SIGUSR1 signal to a `qvm` process to obtain human-readable (i.e., text-based) information about the VM, as follows:

```
slay -sSIGUSR1 qvm
```

This signal causes `qvm` and all loaded vdevs to log free-form text describing their current status at the QL_QVM_INFO level (i.e., information type messages). For an explanation of filtering messages by their type and specifying the logging destination for a VM, refer to the logger option description.

The information logged includes runtime data from vdevs and other sources, and is useful for troubleshooting guest OS and driver behavior.

Alternatively, you can send the SIGUSR2 signal to a `qvm` process to dump the hosted guest's state— meaning its memory and CPU registers—and other troubleshooting-related information at any time. (The guest continues running after the dump.) For information on specifying the file that the dump is written to and on its contents, go to the `dump` option reference.

Reading VM runtime information from the **env** file

Each `qvm` process creates a file that provides details about the VM. This file is kept at **/dev/qvm/*name*/env** on the host, where *name* is the system name for the guest. This system name is set via the `system` option in the `qvm` configuration. For example, if your hypervisor system has a guest called `linux1`, the underlying `qvm` process produces a file called **/dev/qvm/linux1/env**.

This file is in JSON format, so it is machine- and human-readable, and it provides information that can't be known before `qvm` starts, such as vCPU thread IDs and architecture-specific errata data. The exact errata reported are controlled by configuration variable settings; for details, refer to the "Configuration variables" section.

Page updated: August 11, 2025

Virtual Machine Firmware

The virtual machines (VMs) in a QNX Hypervisor system provide a comprehensive firmware interface for guest OSs to manage VM power settings, access timers, and more.

The firmware interface depends on the target platform's architecture. For AArch64, the VMs support many key firmware services that follow modern standards defined by ARM. For x86_64, firmware access is provided by ACPI. Much of the firmware functionality is emulated, meaning it's implemented in software on the host, but for a few functions, the requests are forwarded to and handled by the host hardware.

Page updated: August 11, 2025

Firmware interface for AArch64

The firmware interface provided by QNX Hypervisor VMs on AArch64 (ARMv8) platforms adheres to the Secure Monitor Call Calling Convention (SMCCC) which governs the use of the SMC and HVC (HyperVisor Call) instructions. This interface includes the ARM Architecture Service, ParaVirtualized Timer (PV TIMER), and Power State Coordination Interface (PSCI) services.

The conventions followed by this interface are fully described in the *SMC Calling Conventions* document that is available at: https://developer.arm.com/documentation/den0028/c.

The conduit (mechanism) that QNX Hypervisor supports for calling firmware functions is the HVC machine instruction, which accepts a function ID to indicate the call being made. The HVC instruction generates a synchronous exception that the hypervisor handles, running at exception level 2 (EL2). Arguments and return values are passed in registers.

The SMCCC specification defines function ID ranges for the various HVC-accessed services while other ARM specifications further define individual services and functions. The table that follows describes the firmware services supported by `qvm` (i.e., the VMs), including the total range of function IDs reserved for each service.

NOTE:
The function ID ranges shown below start at 0xC0000000 which means the SMC64/HVC64 calling convention is in effect.

Service name	Function ID range	Description
ARM Architecture	0xC0000000 – 0xC0008000	SMC Calls defined in the ARM Architecture range.
PSCI	0xC4000000 – 0xC400001F	Interface for OS-directed system power state control.
PV Timer	0xC5000020 – 0xC5000040	Interface for measuring stolen time on virtualized systems.

The sections that follow list the individual functions of each service and whether they are emulated by `qvm`, forwarded to the host hardware, or not supported. The unsupported functions are included to clarify which firmware operations guest OSs can or cannot perform via the VMs, and also to give insight into the overall design of these services.

Support for ARM Architecture service

The ARM Architecture service allows a guest OS to enable or disable workarounds for known hardware vulnerabilities. The following table lists the function calls defined by this service and their `qvm` implementation statuses:

Call name	Function ID	Implementation status
SMCCC_VERSION	0x80000000	Emulated
SMCCC_ARCH_FEATURES	0x80000001	Emulated
SMCCC_ARCH_SOC_ID	0x80000002	Not supported
SMCCC_ARCH_WORKAROUND_1	0x80008000	Forwarded to host
SMCCC_ARCH_WORKAROUND_2	0x80007FFF	Supported but ignored
Call_Count	0x8000FF00	Not supported
Call_UID	0x8000FF01	Not supported
Revision	0x8000FF03	Not supported

For more information, go to Chapter 7 (“Arm Architecture Calls”) in the *SMC Calling Conventions* document at: https://developer.arm.com/documentation/den0028/c.

Support for PSCI service

The PSCI service allows guest OSs to control the power of their underlying VMs. By suspending power to a VM, a guest can make its vCPU threads go idle. This action provides more CPU cycles for other threads in the hypervisor host to run, improving performance for host applications and other guests.

The following table lists all the functions defined for this service and their support by `qvm`:

Call name	Function ID	Implementation status
PSCI_VERSION	0x84000000	Emulated
PSCI_FEATURES	0x8400000A	Emulated
PSCI_AFFINITY_INFO_32	0x84000004	Emulated
PSCI_AFFINITY_INFO	0xC4000004	Emulated
PSCI_CPU_ON_32	0x84000003	Emulated
PSCI_CPU_ON	0xC4000003	Emulated
PSCI_CPU_OFF	0x84000002	Emulated
PSCI_CPU_SUSPEND_32	0x84000001	Emulated
PSCI_CPU_SUSPEND	0xC4000001	Emulated
PSCI_CPU_DEFAULT_SUSPEND_32	0x8400000C	Emulated
PSCI_CPU_DEFAULT_SUSPEND	0xC400000C	Emulated
PSCI_SYSTEM_OFF	0x84000008	Emulated
PSCI_SYSTEM_RESET	0x84000009	Emulated
PSCI_SYSTEM_SUSPEND_32	0x8400000E	Emulated
PSCI_SYSTEM_SUSPEND	0xC400000E	Emulated
PSCI_CPU_FREEZE	0x8400000B	Not Supported
PSCI_MIGRATE_32	0x84000005	Not Supported
PSCI_MIGRATE	0xC4000005	Not Supported
PSCI_MIGRATE_INFO_TYPE	0x84000006	Not Supported
PSCI_MIGRATE_INFO_UP_CPU_32	0x84000007	Not Supported
PSCI_MIGRATE_INFO_UP_CPU	0xC4000007	Not Supported
PSCI_NODE_HW_STATE_32	0x8400000D	Not Supported
PSCI_NODE_HW_STATE	0xC400000D	Not Supported
PSCI_SET_SUSPEND_MODE	0x8400000F	Not Supported
PSCI_STAT_RESIDENCY_32	0x84000010	Not Supported
PSCI_STAT_RESIDENCY	0xC4000010	Not Supported
PSCI_STAT_COUNT_32	0x84000011	Not Supported
PSCI_STAT_COUNT	0xC4000011	Not Supported

The full specification for the PSCI service and its various functions is given in the *Arm Power State Coordination Interface Platform Design Document* that's available at: **https://developer.arm.com/documentation/den0022/db**.

Support for PV Timer service

The PV Timer service allows guest OSs to measure their *stolen time* on hypervisor systems. This helps the guests accurately account for their used CPU time and set correct deadlines when scheduling tasks. Further explanation of stolen time is provided in the "Time" section in the "Understanding QNX Virtual Environments" chapter.

The following table lists the function calls defined by this service and their support by `qvm`:

Call name	Function ID	Implementation status
PV_TIME_FEATURES	0xC5000020	Emulated
PV_TIME_ST	0xC5000021	Emulated

The full specification for the PV Timer service and its various functions is given in the *Arm Paravirtualized Time for Arm-based Systems Platform Design Document* that's available at: https://developer.arm.com/documentation/den0057/latest/.

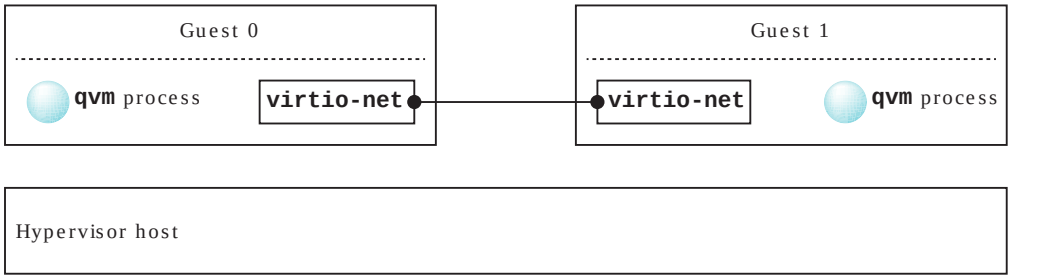
Page updated: August 11, 2025

Guest-to-guest

This section describes a configuration for `virtio-net` vdevs used to establish communication between guests.

The figure below illustrates the peer-to-peer connection with a `virtio-net` vdev in each guest that would result from the configuration described below.

Figure 1Guest-to-guest communication using `virtio-net` vdevs as peer nodes in each guest.



To establish communication between two guests via `virtio-net` vdevs, the guests *must* specify the `name` option (see "Common vdev options" in the "Virtual Device Reference" chapter) to create a peer `virtio-net` vdev path that appears as **/dev/qvm/*system_name*/*vdev_name***. Here, *system_name* is the value you gave to the `system` option and *vdev_name* is the name of the peer vdev (e.g., **/dev/qvm/qnx80-x86_64-guest/virtio-net_0**).

When you use the `virtio-net` vdev's `peer` option, the full name of the end of the peer link is formed from the value of *system_name* concatenated with the value of the `virtio-net` vdev's `name` option. Thus, in the `qvm` configuration files for two QNX guests, you might have, for Guest 0:

```
system qnx80-x86_64-guest
    vdev virtio-net
    mac 56:4d:51:ae:23:0e
    name curly
    peer /dev/qvm/qnx80-x86_64-guest/moe
```

where `/dev/qvm/qnx80-x86_64-guest/moe` points to Guest 1.

For Guest 1, you might have this configuration:

```
system qnx80-x86_64-guest
    vdev virtio-net
    mac 56:4d:51:87:3d:ac
    name moe
    peer /dev/qvm/qnx80-x86_64-guest/curly
```

where `/dev/qvm/qnx80-x86_64-guest/curly` points to Guest 0.

With these configurations, the peer `virtio-net` vdevs appear as follows on the host:

```
/dev/qvm/qnx80-x86_64-guest/moe
/dev/qvm/qnx80-x86_64-guest/curly
```

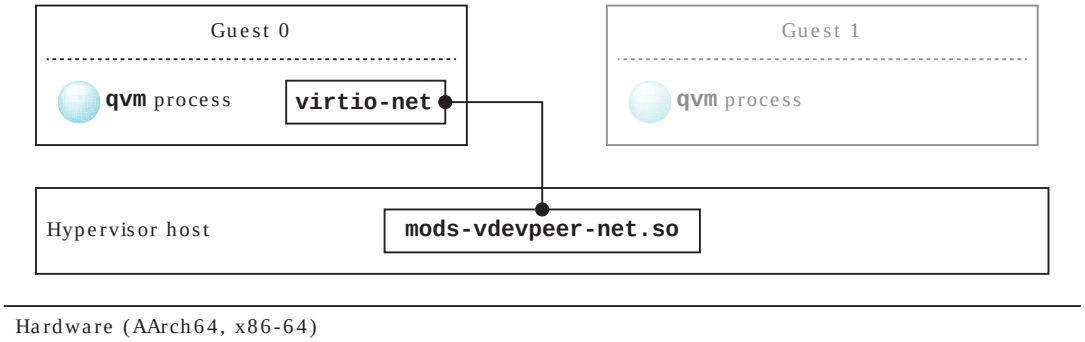
Page updated: August 11, 2025

Guest-to-host

This section describes a configuration for a `virtio-net` vdev in a guest communicating with a network driver in the hypervisor host.

The figure below illustrates a peer-to-peer connection between the `virtio-net` vdev in a guest and the `io-sock` driver `mods-vdevpeer-net.so` in the hypervisor host.

Figure 1Guest-to-host communication between a `virtio-net` vdev and a peer node provided by the `mods-vdevpeer-net.so` driver.



Configure a `virtio-net` vdev for guest-to-host communciation

The following excerpt shows the `virtio-net` vdev configuration in the ***.qvmconf** file for the VM hosting the guest.

For a QNX guest on an ARM board, configure a `virtio-net` vdev as follows:

```
system qnx80-arm-guest
...
# The loc and intr gic options are for ARM only. The guest will see the
# virtio-net vdev as a memory-mapped I/O device at the specified
location.

vdev virtio-net
  loc 0x1c0c0000
  intr gic:40
  mac aa:aa:aa:aa:aa:aa
  name p2p
  peer /dev/vdevpeers/vp0
  peerfeats checksum
```

where the following settings are defined:

qnx80-arm-guest
: The system name of the VM for the guest (see the `system` option for more details), also specified by the second last token in the `vdevpeer-net peer` option (`peer=/dev/qvm/`**`qnx80-arm-guest`**`/p2p`).

loc 0x1c0c0000
: The base address of the device registers.

intr gic:40
: The interrupt number of the current vdev (see "Common vdev options" in the "Virtual Device Reference" chapter).

mac aa:aa:aa:aa:aa:aa
: The locally assigned MAC address of the node in the guest.

name p2p
: The name of the node in the guest, also specified by the last token in the `vdevpeer-net peer` option (`peer=/dev/qvm/qnx80-arm-guest/`**`p2p`**).

peer /dev/vdevpeers/vp0
: The path to the node in the host, also specified by the `vdevpeer-net bind` option (`bind=/dev/vdevpeers/vp0`).

peerfeats checksum
: The VIRTIO Network feature bits supported by the peer (for more information, go to the note at the end of this section and "Using the `peerfeats` option" in the `vdev virtio-net` entry).

Start `vdevpeer-net`

The following excerpt shows the `vdevpeer-net` driver startup options that enable the hypervisor host to connect to a node in the guest.

When starting `io-sock` in the host, specify the following options and configuration for the `vdevpeer-net` driver:

```
io-sock -m vdevpeer-net
ifconfig vp0 create
vpctl vp0 peer=/dev/qvm/qnx80-arm-guest/p2p bind=/dev/vdevpeers/vp0
mac=a0b0c0d0e0f0
```

where:

`peer=`*`vdev_path`*
: The path to a `virtio-net` vdev in the guest, inside the **/dev/qvm/** directory. The remainder of the path is specified by the `virtio-net` vdev `system` option for the guest's directory (**qnx80-arm-guest**), and inside this directory, the node in the guest, specified by the vdev's `name` option (**p2p**).

`bind=`*`host_node_path`*
: The path to the node in the host, inside the **/dev/vdevpeers/** directory. In this case the default prefix `vp` is used for node zero (`vp0`).

`mac=`*`host_mac_address`*
: The MAC address for the node in the hypervisor host.

Enable the guest interface

This subsection provides instructions for enabling the interface. Suppose that you have already enabled an interface on the host, as follows:

```
ifconfig vp0 up
ifconfig vp0 192.168.1.1
```

where `vp0` is the peer-to-peer interface on the host, and `192.168.1.1` is its address.

You must now enable the interface on the guest and assign it a static IP address in the same subnet as the host. For example, for a QNX guest:

```
ifconfig vtnet0 192.168.1.2
```

where `vtnet0` is the name of the interface on the guest, and `192.168.1.2` is an address in the same subnet as the host.

For a Linux guest:

```
sudo ifconfig enp0s3f0 up
sudo ifconfig enp0s3f0 192.168.1.2
```

where `enp0s3f0` is the name of the interface on the Linux guest, and `192.168.1.2` is an address in the same subnet as the host.

NOTE:

If you specify `peerfeats checksum` or `peerfeats guest-checksum` for the peer interface on the guest, you must ensure that the checksum settings for the interface on the host match those for the interface on the guest. For example, to disable checksums for TCP and UDP on the node, configure it as follows:

```
ifconfig vp0 -rxcsum -txcsum -rxcsum6 -txcsum6
```

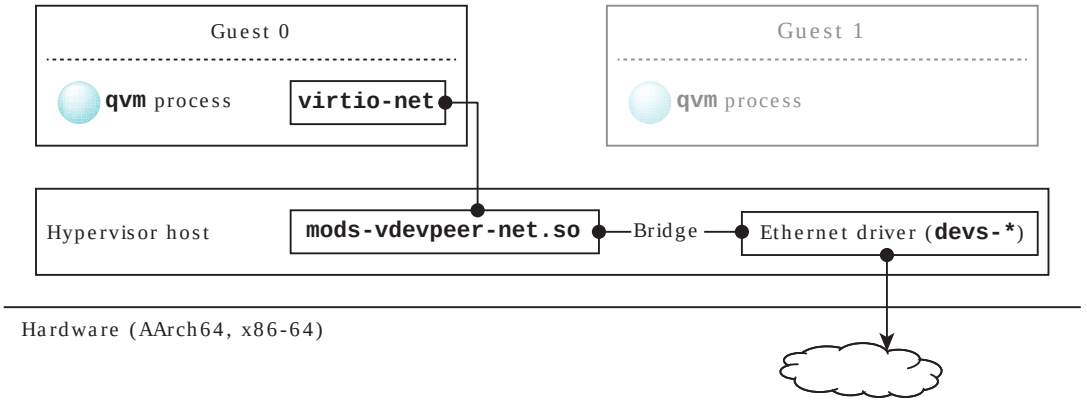
Page updated: August 11, 2025

Guest-to-world

This section describes a configuration for a guest communicating with the outside world through network drivers in the hypervisor host.

The figure below illustrates guest-to-world communication using a `virtio-net` vdev in the guest and a `mods-vdevpeer-net.so` driver in the hypervisor host connected via a bridge to an Ethernet driver also in the host.

Figure 1Guest-to-world communication with a `virtio-net` vdev in the guest connected to a `mods-vdevpeer-net.so` driver connected via a bridge to an Ethernet driver in the host.



Configure a `virtio-net` vdev

In the configuration for the VM hosting the guest, configure a `virtio-net` vdev so that you can establish peer-to-peer communication with the `mods-vdevpeer-net.so` driver. For example, for a QNX guest on an ARM board:

```
system qnx80-arm-guest
...
# The loc and intr gic options are for ARM only. The guest will see the
# virtio-net vdev as a memory-mapped I/O device at the specified
location.

vdev virtio-net
  loc 0x1c0c0000
  intr gic:40
  mac aa:aa:aa:aa:aa:aa
  name p2p
  peer /dev/vdevpeers/vp0
  peerfeats checksum
```

This is exactly like the `virtio-net` vdev configuration in “Guest-to-host”.

Enable services in the host

In the host, start by starting `io-sock`, specifying `mods-vdevpeer-net.so` and other network drivers. For example:

```
io-sock -o config=/etc/system/config/net/io-sock.conf -m phy -m fdt -m
phy_fdt \
    -m pci -d em -m vdevpeer-net
```

Note the `-d em` option for `io-sock`, which starts the driver for an Intel PRO/1000 Gigabit Ethernet adapter needed to connect to the outside world.

Next, bring up the network interfaces in the host. For example:

```
if_up -p em0
ifconfig em0 up
ifconfig vp0 create up
vpctl vp0 peer=/dev/qvm/qnx80-arm-guest/p2p bind=/dev/vdevpeers/vp0
mac=a0b0c0d0e0f0
```

Use a command like the following to configure the MAC address for the bridge and add the appropriate network interface (or interfaces) to it:

```
sysctl -w net.link.bridge.inherit_mac=1 > /dev/null
ifconfig bridge0 create up addm vp0 addm em0
```

Use `dhcpcd` to assign network IP addresses. For example:

```
dhcpcd -bqq -w bridge0
```

Finally, configure other networking services. For example:

```
ifconfig vp0 -txcsum -txcsum6 -tso6 -tso4
ifconfig vp1 -txcsum -txcsum6 -tso6 -tso4
/proc/boot/.ssh-server.sh
```

Page updated: August 11, 2025

Networking

You can set up communications between the host and guests, between guests, and between guests and the outside world.

When you design a network interface for a guest, you can use:

- a `virtio-net` vdev connected to another `virtio-net` vdev, usually in another guest
- a `virtio-net` vdev connected to a `mods-vdevpeer-net.so` driver instance running in the hypervisor host
- a pass-through device

A `virtio-net` vdev offers peer-to-peer communication; for more information, see `vdev virtio-net` in the "Virtual Device Reference" chapter.

The `mods-vdevpeer-net.so` driver (a FreeBSD-compatible module) runs in the hypervisor host. It enables the `io-sock` service in the host to communicate with guests through appropriately configured vdevs in the `qvm` process instances that host the guests. For more information, see the `mods-vdevpeer-net.so` entry in the *Utilities Reference*.

MAC addresses in a hypervisor system

Note the following about assigning MAC addresses:

- MAC addresses must be unique in any given Ethernet segment. There can be multiple Ethernet segments on a target board, and a target board may share a segment with some external entity.
- If you don't specify MAC addresses, the vdev and `mods-vdevpeer-net.so` drivers generate them for you. Since these addresses are generated, they change each time you restart the networking manager (`io-sock`) or the VM.
- When you assign MAC addresses, set the locally assigned bit (see *IEEE 802.3-2015* Section 1, paragraph 3.2.3, item b).

Enabling peer-to-peer networking

To support connectivity between guests and the outside world, peer-to-peer networking must be enabled in the host, along with a network driver (e.g., `em`).

You can do this by starting `io-sock` in the host and then using `vpctl` to specify the peer interfaces you want to create.

You can add the `io-sock` startup instructions to the hypervisor host's startup scripts, or use the command line to start `io-sock` after the host has booted.

You can load the `mods-vdevpeer-net.so` driver when you start `io-sock`, using the `-m vdevpeer-net` option, or later, using `mount`. For example:

```
mount -T io-sock mods-vdevpeer-net.so
```

The following commands start both `io-sock` and the host drivers and create three interfaces:

```
io-sock -m pci -d em -m vdevpeer-net
ifconfig vp0 create
ifconfig vp1 create
ifconfig vp2 create
```

The following commands configure the interfaces to connect to the guest's `virtio-net` vdev:

```
vpctl vp0 peer=/dev/qvm/qnx8-guest1/p2p bind=/dev/vdevpeers/vp0
vpctl vp1 peer=/dev/qvm/qnx8-guest2/p2p bind=/dev/vdevpeers/vp1
vpctl vp2 peer=/dev/qvm/linux-guest/p2p bind=/dev/vdevpeers/vp2
```

You use the `io-sock -d` and `-m` options to start `devs-*` and `mods-*` drivers (see "Starting `io-sock` and Driver Management" in the QNX OS *High-Performance Networking Stack (`io-sock`) User's Guide*). In the example above, the `em` driver is required only if your system needs to connect to the outside world (see "Guest-to-world" in this chapter). If you only need to connect guests to each other, you can omit this driver.

Page updated: August 11, 2025

Guest exits

Guest exits are one of the most significant sources of overhead in a hypervisor system.

It is impossible to avoid all guest exits in a hypervisor system. However, since guest exits are costly, reducing the number of these exits can improve the performance of guests and of the overall system.

Why a guest exits

When a guest is running, its instructions execute on a physical CPU, just as if the guest were running without a hypervisor. However, guests are not allowed to do everything they would be allowed to do in a non-virtualized system. This is one way that a hypervisor system protects itself from its guests as well as its guests from each other.

A guest exit can be triggered by:

- the guest itself (see "Guest-triggered exits")
- the hardware or the hypervisor host (see "Interrupts")

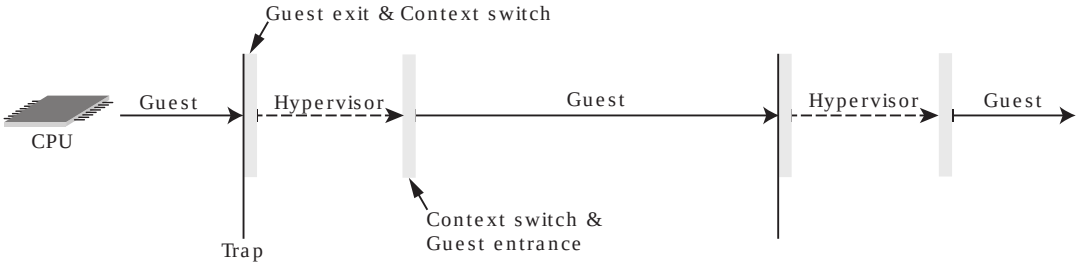
The work required for a guest exit

The following presents an overview of what happens when a guest attempts to execute an instruction that it isn't permitted to execute, but which the hypervisor host can look after for the guest (assuming that the instruction will not cause the hypervisor to return an error to the guest):

1. The virtualization hardware traps the attempt, and forces the guest to stop executing (guest exit).
2. On the trap, the hardware notifies the hypervisor.
3. The hypervisor saves the guest's context, then completes the task the guest had begun but was unable to complete for itself.
4. When it has completed the task, the hypervisor restores the guest's context, updated to reflect the results of whatever task it completed for the guest.
5. The hypervisor hands execution back to the guest (guest entrance). The guest is none the wiser. It may not even know that it was the hypervisor and not the hardware that completed the task (with the exception of para-virtualized devices); it doesn't even know that any time has passed since it was forced to exit (see "Para-virtualized devices" and "Time" in the "Understanding QNX Virtual Environments" chapter).

The Lahav Line below presents an overview of the guest exit–guest entrance sequence that is required every time the virtualization hardware traps a guest instruction. For simplicity, it assumes an execution path on a single CPU:

Figure 1A Lahav Line showing how in a QNX hypervisor system execution on the physical CPU alternates between the hypervisor and its guests. On a trap, the hypervisor manages the guest exit, saving the guest's context, then restoring it before the guest entrance. This diagram is the same as the one in "Two representations of a QNX hypervisor system" in the "Understanding QNX Virtual Environments" chapter.



The steps in responding to a guest exit are essentially the same if the hardware or the hypervisor triggers the exit.

The cost of a guest exit

At each guest exit, the hypervisor must store the guest's context; before each guest entrance, it must restore the guest's context, updated to reflect the results of whatever task it completed for the guest.

The minimum overhead time for a guest exit–guest entrance cycle is the time the hypervisor takes to save then restore a guest's context, or, starting from the hypervisor rather than the guest, the time it takes for:

1. The hypervisor to restore the guest's context and pass execution to it.
2. The guest to execute a NOP instruction.

3. Execution to return to the hypervisor.

Not surprisingly, the time required for a hypervisor–guest–NOP–hypervisor round-trip depends largely on the hardware. It is typically from three to ten microseconds, depending on the SoC, but consistent on an SoC; that is, if one round-trip takes five microseconds, then you can count on all round-trips during normal operation taking five microseconds, with very little variance.

Nonetheless, three or ten microseconds repeated often enough can significantly impact a guest's performance, especially if we remember not only that the exit uses the hypervisor's time but also that during this time the guest is waiting.

Page updated: August 11, 2025

Guest-triggered exits

Guests in a hypervisor system can be tuned to reduce the number of guest exits they trigger.

Below are brief descriptions of common guest actions that trigger exits; they should help you identify changes you can make to your guest configuration and to applications running in the guest that will improve performance.

In addition to the causes described below, guest-initiated interprocessor interrupts (IPIs) may cause guest exits (see "Guest IPIs" in this chapter).

For information about how to modify your guests' behaviors to reduce the number of exits they trigger, consult the user documentation for your guest OSs. For example, for QNX guests, see the QNX OS documentation.

Reading or writing a virtual device register

When a vdev emulates a physical device (e.g., `pl011` or `ser8250`), the guest doesn't know that it is working with a virtual device rather than with a hardware device. This means that when a guest reads or writes to one of the vdev's registers, the hypervisor must interpret the register reference so that, when it has completed the device emulation, it can update the state of the vdev to match what the guest expects from hardware. Of course, this interpretation and update requires the hypervisor to execute, which forces a guest exit.

Para-virtualized devices (e.g., `virtio-blk`) can't eliminate all guest exits, but they are designed to minimize the number of times guest exits are required. When designing your system, consider using where possible para-virtualized devices rather than emulation vdevs.

For more information about both types of virtual devices, see "Emulation vdevs" and "Para-virtualized devices" in the "Understanding QNX Virtual Environments" chapter.

Reading or writing a privileged system register

Boards with virtualization support are designed to minimize the number of times a guest needs to read or write privileged system registers. Nonetheless, both ARM and x86 boards still have some system registers (e.g., `OSLSR_EL1` on ARM, `IA32_APIC_BASE` on x86) that the guest may not directly read or modify.

Thus, the virtualization hardware traps any attempts by a guest to access these registers, forcing a guest exit so that the hypervisor can emulate the operation. If you can reduce the frequency at which a guest needs to access these privileged system registers, you will reduce the frequency of your guest exits.

Requests for services from higher exception levels

One of the fundamentals of hardware virtualization is the separation of hypervisor host and guest activities into different privilege levels. On ARM platforms, these privilege levels are now called *exception levels* (ELs); on x86 platforms, they are called rings (see "CPU privilege levels" in the "Understanding QNX Virtual Environments" chapter, and "Exception Level (EL)" and "Ring" in the "Terminology" appendix).

Guests run with fewer privileges than the hypervisor. When a guest tries to perform an action for which it doesn't have sufficient privileges, the virtualization hardware traps the request, forcing the guest to exit and the hypervisor to complete the operation. For example, on an ARM board, a guest isn't permitted to execute the SMC opcode to request a Trustzone service. If a guest requests a Trustzone service it will be forced to exit so that the hypervisor can look after the request, either re-issuing the request to the host Trustzone code or handling the operation itself.

Regardless of how the hypervisor handles the guest's attempt to perform an action for which it has insufficient privileges, the attempt forces a guest exit along with its attendant costs. If you can reduce the incidents of requests for services that require privilege levels the guest doesn't have, you can improve the guest's performance, and reduce the overall guest exit–guest entrance overhead in your system, improving overall system performance.

Access to floating point register state and performance counters

A QNX hypervisor uses a similar technique to virtualize performance monitoring units (PMUs) and floating point units (FPUs). A guest's CPU is in fact a vCPU: a thread in the `qvm` process instance hosting the guest. This thread, which runs in the hypervisor host, maintains the floating point state for the guest's CPU, as well as the state of the PMU.

The hypervisor host lazily switches the contexts of its threads' floating point states, which generally reduces the number of context switches it needs to do and saves time in the system overall. With this design, however, if a hypervisor guest attempts to reference its floating point state (i.e., the floating point state of a vCPU thread in its

hosting `qvm` process instance), the physical FPU registers may contain data belonging to another thread in the hypervisor host.

In this case, the guest must exit so that the hypervisor can context switch the FPU registers, loading the values for that guest's vCPU thread. As with other guest exits, this operation requires a guest exit–guest entrance cycle along with its overhead. Reducing the frequency of a guest's attempts to reference its floating point state therefore reduces overhead and can improve performance.

Reprogramming timers

Depending on the platform, programming of timers by a guest may indirectly trigger guest exits.

For example, on ARM platforms, the hypervisor doesn't need to trap a guest attempt to program a *virtual* timer; the programming takes place in the software of the VM hosting the guest. However, the hypervisor must trap any guest attempts to program a *physical* timer, which will cause a hardware interrupt and the guest to exit.

Thus, a guest OS that uses high-precision timers, continuously reprogramming timer hardware, will incur the overhead of the guest exits required to handle the reprogramming.

Page updated: August 11, 2025

Guest IPIs

Reducing the frequency of guest-issued interprocessor interrupts (IPIs) can improve the performance of the guest and of the overall system.

The cost of an IPI between physical CPUs (pCPUs) typically is less than a microsecond when initiated by an operating system running directly on the hardware. While this overhead isn't significant, any excessive use of IPIs may affect system performance.

Just like any OS running running directly on hardware with multiple pCPUs, a guest OS running in a VM with multiple vCPUs may need to issue IPIs. From the perspective of the OS issuing the IPI, the IPI behavior is exactly the same regardless of whether the OS is running directly on hardware or as a guest in a VM.

However, the time overhead cost of an IPI issued by a guest OS in a VM is an order of magnitude greater that the cost of an IPI issued by an OS running directly on hardware. This cost is similar to the cost of a guest exit-entrance cycle, which typically can take ten microseconds, sometimes longer.

Since a guest OS runs in a VM and its CPUs are in fact vCPUs (threads in the hosting `qvm` process instance), when a guest issues an IPI, the IPI source is a vCPU thread, and each IPI target is another vCPU thread.

The relatively high cost of guest-issued IPIs is due to the work required by the hypervisor to prepare and deliver these IPIs; that is, to the work that must be done by software rather than by hardware to deliver the IPI from its source vCPU thread to its target vCPU thread or threads.

NOTE:
Just as an IPI issued by a system running directly on hardware can target multiple pCPUs, an IPI issued by a guest in a VM can target multiple vCPUs.

Preparing the IPI for delivery

The hypervisor tasks described below prepare a guest-issued IPI for delivery. They are the same regardless of the board architecture or the state of the target vCPU thread.

When the guest OS issues an IPI, the hypervisor must:

1. Trap the request in the source vCPU thread.
2. Write the hosting VM's interrupt controller registers (virtual ICRs) with the information that will be needed to deliver the guest-issued IPI to all target vCPUs.
3. Compile this information into a list of target vCPUs and interrupt numbers.
4. For each target vCPU, inject the relevant interrupt number. That is, prepare the data and logic that will cause that target vCPU to see that it has an interrupt pending when it resumes.

Delivering the IPI

From this point forward, the work required to deliver a guest-issued IPI to a vCPU is the same as for delivering any interrupt to a vCPU, regardless of the interrupt's source. This work differs according to the state of each target vCPU thread. Also, some boards support posted interrupts, which can reduce overhead.

Target vCPU thread isn't executing

If a target vCPU isn't executing guest code (the guest is stopped on an HLT or WFI), then the host will see the target vCPU thread as stopped on a semaphore. In this case, assuming that the source vCPU thread prepared the IPI for delivery:

- Normal QNX OS scheduling policies apply and the target vCPU thread will begin executing again when its turn comes.
- When the vCPU thread begins executing, it picks up the guest-issued IPI from its VM's virtual ICRs.

Target vCPU thread is executing

If a target vCPU thread is executing guest code:

- The hypervisor must trigger an IPI on the pCPU where the guest code is executing; if posted interrupts aren't supported by the hardware or aren't enabled, the interrupt on the pCPU forces a guest exit and immediate re-entrance.
- When the guest re-enters, the target vCPU thread will pick up the pending guest-issued IPI from its VM's virtual ICRs.

However, note the following:

- If a target vCPU thread is executing vdev code, no forced guest exit is needed. Virtual devices are part of the hypervisor; when execution passes from the virtual device back to the guest, the target vCPU will pick up the pending guest-issued IPI from its VM's virtual ICRs.
- If the hardware supports posted interrupts (as some x86 boards do) and this capability is enabled, then no guest exit is required to deliver a guest-issued IPI to its target vCPU(s). The hypervisor's IPI on the pCPU triggers delivery of the guest-issued IPI, and the guest learns about the interrupt without having to exit.

Reducing the frequency of guest-issued IPIs

The costliest tasks in the preparation and delivery of a guest-issued IPI to its target vCPU(s) are:

- Compiling the list of target vCPUs. There is no mechanism for circumventing this task.
- The guest exit–entrance cycle. This cost can be reduced by enabling posted interrupts, if the hardware supports this capability.

Given the high cost of guest-issued IPIs, even on boards with posted interrupt support, reducing the frequency of guest-issued IPIs can improve both guest and overall system performance. This reduction can often be achieved by managing which vCPUs a guest application runs on; in this case, by binding the relevant processes in the guest to a single vCPU:

- If the threads of a multithreaded process in the guest frequently interact, bind the process to a single vCPU (i.e., a single `qvm` process vCPU thread).
- If several processes are communicating frequently, bind them to the same vCPU.

With these configurations, the application will behave as though it is running on a single-CPU system and won't issue IPIs that require hypervisor intervention to deliver.

For a Linux guest, use the `taskset` command (see your Linux documentation). For a QNX guest, use the `on` command with the `-C` option. For example, the following command:

```
on -C 1 foo
```

binds the program `foo` to CPU 1 (from the guest's perspective), which is in fact a `qvm` process vCPU thread (see the `on` utility in the QNX OS *Utilities Reference*).

In some cases, if you are running a single-threaded application, it may even prove advantageous to run that application in its own guest OS running in a VM on its own dedicated pCPU.

NOTE:
The `set` option's *exit-on-halt* argument allows you to control if a WFI (ARM) or HLT (x86) instruction causes a guest exit (see "Configuration variables" in the "VM Configuration Reference" chapter).

Page updated: August 11, 2025

Interrupts

Superfluous interrupts can significantly compromise guest and system performance.

In a hypervisor system, the guest configures the interrupt controller hardware, but it is the hypervisor that manages the hardware to fulfill the needs of the guest. Thus, when a hardware device asserts an interrupt for a guest, the hypervisor must always intervene. This means that on an interrupt, the guest must exit at least once to allow the hypervisor to examine the interrupt and decide what to do with it.

Even if the device that asserts an interrupt is configured as a pass-through device (see "Pass-through devices" in the "Understanding QNX Virtual Environments" chapter), it is the hypervisor that manages the interrupt controller hardware. The hypervisor must mask the interrupt, pass it up to the guest by updating the vCPU thread with the interrupt information, and finally, on the physical EOI unmask the interrupt.

The hypervisor's intervention in the interrupt delivery to the guest, even for pass-through devices, prevents interrupt storms from compromising the system behavior, and allows for a proper cleanup of pending interrupts in the case of a guest failure.

Since interrupts trigger guest exits, eliminating actions that trigger superfluous interrupts is one of the most effective ways to improve guest and overall system performance. However, since interrupts are essential to any functioning system, they can't be eliminated. What you can do, though, is design and configure your system to handle interrupts efficiently.

NOTE:
Guest IPIs are no exception; the hypervisor also handles them as interrupts (see "Guest IPIs" in this chapter).

Virtualization support in the hardware

If the hardware provides no special support for interrupts in a virtualized system, the hypervisor host looks after everything. It traps each interrupt and updates the appropriate vCPU thread structure with the interrupt information. This software-only method is available on all hardware and is the default method if no other methods are offered by the hardware. It requires at least one guest exit and incurs the greatest overhead.

Many ARM and x86 platforms provide virtualization support that reduces the work required to deliver an interrupt to a guest in a virtualized system. When planning your system, consider the performance benefits of these hardware features.

To take advantage of hardware assistance with interrupts you need:

- hardware support for the feature (e.g., posted interrupts on x86 platforms)
- BSP support to include, enable, and configure the required components and features (e.g., GICv3 with hardware assist)
- VM configuration that includes and configures the required features
- guest configuration that includes and configures the required features

ARM

On supported ARM platforms, the hypervisor provides two techniques for reducing the guest exits required to handle interrupts.

Interrupt enable

If the hardware allows, the hypervisor can set the guest's IRQ requested bit even if interrupts are disabled in the guest. Setting this bit doesn't cause a guest exit.

See your ARM hardware documentation for more information about implementing this hardware feature.

Generic Interrupt Controller (GIC) hardware assist

Many ARM platforms with GIC hardware provide optional hardware assist capabilities for interrupt handling.

GIC hardware assist doesn't eliminate guest exits on physical interrupts, but it does help with the guest manipulations of the virtual GIC state in response to guest interrupt delivery (pass-through or from a vdev).

x86

On supported x86 platforms, hardware assistance for handling interrupts in a virtualized system currently includes LAPIC hardware assist and posted interrupts.

Local component Advanced Programmable Interrupt Controller (LAPIC) hardware assist

LAPIC hardware assist helps reduce the number of guest exit-guest entry cycles required to handle an interrupt.

Systems that use LAPIC provide data structures with information about the LAPIC. To reduce the number of guest exits required to handle an interrupt delivery to a guest, your device driver can turn on the bit associated with the hardware-assisted LAPIC, then signal the guest to run.

This method can be used only if the hardware supports posted interrupts (see below), or if the guest's vCPU is already stopped (i.e., the vCPU thread scheduled to receive the interrupt is already in the STOPPED state). If the hardware doesn't support posted interrupts and the vCPU isn't already stopped, the vCPU must be stopped, which requires a guest exit and re-entry.

This method affects only the vCPU targeted by the interrupt. The VM's other vCPUs can continue to run.

Posted interrupts

If the hardware supports posted interrupts, this capacity can be used to eliminate the need for a guest to exit to receive an interrupt.

As always, the hypervisor must examine the interrupt to determine its destination, which requires a guest exit-entrance cycle. With posted interrupt support, however, when it has determined what to do with an interrupt, the hypervisor doesn't need to force the guest to exit so that the guest will notice an interrupt waiting in its virtual LAPIC:

1. The guest can continue to run.
2. The hypervisor instructs the hardware to update the guest's virtual LAPIC state with the pending interrupt.
3. The guest notices the change to its virtual LAPIC state just as it would notice the change to its LAPIC state if it were running in a non-virtualized environment.
4. The guest responds to the interrupt.

Suppose a guest with a single vCPU is performing some task such as calculating the value of *pi*. A hardware device asserts an interrupt for the guest. The hardware forces the guest to exit so the hypervisor can examine the interrupt. The guest re-enters and continues running to calculate *pi*.

Remember that the `qvm` process instance for the VM hosting the guest has at least two threads: the thread for the vCPU and the `qvm` main thread. With posted interrupt support, the above could be achieved as follows: the main thread instructs the virtualization support hardware to post the interrupt to the guest; the hardware updates the guest's virtual LAPIC, delivering the interrupt to the guest without forcing another guest exit.

Page updated: August 11, 2025

Overhead in a virtualized environment

There are many sources of overhead in a virtualized environment; finding them requires analysis both from the top down and from the bottom up.

Sources of overhead in QNX hypervisor systems include:

Guest exits
: These occur whenever a virtualization event requires that the guest leave the native environment (i.e., stop executing). Such events include the hypervisor examining and passing on interrupts, and looking after privileged operations such as updating registers. You can reduce these events by configuring your guest OSs so that they don't unneccesarily trigger exits (see "Guest exits" and "Guest-triggered exits").

Interrupts
: Whether they are initiated by the hardware or, indirectly, by a guest request, individual interrupts in a hypervisor system require a guest exit so the hypervisor can manage them. This means the cost of interrupts can be significantly greater than in a non-virtualized system. You can reduce the cost of interrupts by eliminating superfluous interrupts (e.g., by using VIRTIO devices where appropriate) and by using hardware virtualization support to reduce the cost of the necessary interrupts (see "Interrupts" and "Virtual I/O (VIRTIO)").

Hypervisor overload
: Poor VM configuration can force the hypervisor to trigger guest exits to manage competing demands for resources. You can often resolve many overload problems by modifying your VM configuration (see "vCPUs and hypervisor performance").

NOTE:
A good general rule to follow when tuning your hypervisor system for performance is that for a guest OS in a VM, the relative costs of accessing privileged or device registers and memory are different than for an OS running in a non-virtualized environment.

Usually, for a guest OS, the cost of accessing memory is comparable to the cost for an OS performing the same action in a non-virtualized environment. However, accessing privileged or device registers requires guest exits, and thus incurs significant additional overhead compared to the same action in a non-virtualized environment.

For instructions on how to get more information about what your hypervisor and its guests are doing, see the "Monitoring and Troubleshooting" chapter.

Measuring overhead

Finding sources of overhead requires analysis both from the top down and from the bottom up.

Top-down analysis

1. Run a system in a non-virtualized environment, and record a benchmark ("N" for native).
2. Run the same system in a VM, and record the same benchmark information ("V" for virtual).

   Usually benchmark N will show better performance, though the opposite is possible if the VM is able to optimize an inefficient guest OS.
3. Assuming that benchmark N was better than benchmark V, adjust the virtual environment to isolate the source of the highest additional overhead. If benchmark V was better, you may want to examine your guest OS in a non-virtualized environment before proceeding.

When you have identified the sources of the most significant increases to overhead, you will know where your tuning efforts are likely to yield the greatest benefits.

Bottom-up analysis

1. Run one or more guests in your hypervisor system.
2. Record every hypervisor event over a specific time interval ("T").
3. Use the data recorded in T to analyze the system.

The hypervisor events include all guest exits. Guest exits are a significant source of overhead in a hypervisor system (see "Guest exits" and "Guest-triggered exits" in this chapter).

SoC configuration

QNX hypervisors include a QNX OS microkernel with support for virtualization. As with all QNX microkernel systems, the bootloader and startup code pre-configure the SoC, including use of the physical CPUs (e.g., SMP configuration) and memory cache configuration. The hypervisor doesn't modify this configuration; however, you can do so to provide performance gains.

For more information about how to change the bootloader and startup code, see *Building Embedded Systems* in the QNX OS documentation, and your Board Support Package (BSP) *User's Guide*.

Multiprocessing

The QNX hypervisor supports both symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP). Also, you can configure threads in the hypervisor host to run on specific physical CPUs (pCPUs). This includes vCPU threads and other threads in your `qvm` processes. You can pin these threads to one or several pCPUs by using the `cpu cluster` option when you assemble your VMs.

For more information about QNX OS support for multiprocessing, see the "Multicore Processing" chapter in the QNX OS *Programmer's Guide*.

Page updated: August 11, 2025

Performance Tuning

This chapter describes common sources of overhead in a QNX hypervisor system, and strategies for reducing this overhead and optimizing system performance.

A VM defines its guest's hardware environment. Once its guest is running, however, a VM is at the mercy of the guest's operations, much like hardware is at the mercy of the code it must execute.

The hypervisor must always respond to the needs of guests in its VMs. If a guest is not tuned for optimal performance in a hypervisor VM and for the board on which the hypervisor is running, the hypervisor can't do anything about the situation. For example, if a guest makes requests that generate numerous spurious interrupts (and, therefore, guest exits), the hypervisor can only attempt to handle these interrupts, in the same way that hardware must respond to the requests of a non-virtualized system.

Page updated: August 11, 2025

vCPUs and hypervisor performance

Perhaps counter-intuitively for someone accustomed to working in a non-virtualized environment, more vCPUs doesn't mean more power.

vCPUs and hypervisor overload

Multiple vCPUs are useful for managing guest activities, but they don't add processor cycles. In fact, the opposite may be true: too many vCPUs may degrade system performance.

A vCPU is a VM thread (see cpu in the "VM Configuration Reference" chapter). These vCPUs appear to a guest just like physical CPUs (pCPUs). A guest's scheduling algorithm can't know that when it is migrating execution between vCPUs it is switching threads, not pCPUs. This switching between threads can degrade performance of all guests and the overall system. This is especially common when VMs are configured with more vCPUs than there are pCPUs on the hardware.

Specifically, if the hypervisor host has more threads (including vCPU threads) ready to run than there are pCPUs available to run them, the host's scheduler must use the thread priorities and apply its scheduling policies (round-robin, FIFO, etc.) to decide which threads to run. These policies may employ preemption and time slicing to manage threads competing for pCPUs.

Every preemption requires a guest exit, context switch and restore, and a guest entrance. Thus, inversely to what usually occurs with pCPUs, reducing the number of vCPUs in a VM can improve overall performance: fewer threads will compete for time on the pCPUs, so the hypervisor will not be obliged to preempt threads (with the attendant guest exits) as often.

In brief, fewer vCPUs in a VM may sometimes yield the best performance. You can run fewer vCPUs than there are pCPUs, including just one vCPU.

Multiple vCPUs sharing a pCPU

When configuring your VM, it is prudent to not assume that the guest will always do the right thing. For example, multiple vCPUs pinned to a single pCPU may cause unexpected behavior in the guest: timeouts or delays might not behave as expected, or spin loops might never return, etc. Assigning different priorities to the vCPUs might exacerbate the problem; for example, the vCPU might never be allowed to run.

In short, when assembling a VM, consider carefully how the guest will run on it.

For more information about scheduling in a hypervisor system, see "Scheduling" in the "Understanding Virtual Environments" chapter.

Page updated: August 11, 2025

Virtual I/O (VIRTIO)

VIRTIO is a convenient and popular standard for implementing vdevs; using VIRTIO vdevs can reduce virtualization overhead.

QNX hypervisors support the VIRTIO 1.1 and later specifications. The benefits of VIRTIO include:

- Implementation – no changes are required to Linux or Android kernels; you can use the default VIRTIO kernel modules for Linux or Android guests.
- Efficiency – VIRTIO uses virtqueues that efficiently collect data for transport to and from the shared device.
- Serialized queues – the underlying hardware can handle requests in a serialized fashion, managing the priorities of requests from multiple guests (i.e., priority-based sharing).
- Guest performance – even though the queues to the shared hardware are serialized, the guest can create multiple queue requests without blocking; the guest is blocked only when it explicitly needs to wait for completion (e.g., a call to flush data).

For more information about VIRTIO, including a link to the latest specification, see *OASIS Virtual I/O Device (VIRTIO) TC* on the Organization for the Advancement of Structured Information Standards (OASIS) web site; and Edouard Bugnion, Jason Nieh and Dan Tsafrir, *Hardware and Software Support for Virtualization* (Morgan & Claypool, 2017), pp. 96-100.

Implementing a VIRTIO vdev

For this discussion, we will examine an implementation of VIRTIO using `virtio-blk` (block device support for filesystems). Note that:

- QNX hypervisors support `virtio-blk` interfaces that are compatible with VIRTIO 1.1 or later; so long as the guest supports this standard, this implementation requires no changes to the guest.
- The VIRTIO specification stipulates that the guest creates the virtqueues, and that they exist in the guest's address space. Thus, the guest controls the virtqueues.

The hypervisor uses two mechanisms specific to its implementation of the `virtio-blk` device:

- a file descriptor (fd) – this refers to the device being mapped by `virtio-blk`. The `qvm` process instance for the VM hosting the guest creates one fd per `virtio-blk` interface. Each fd connects the `qvm` process instance to the QNX `io-blk` driver in the hypervisor host.
- I/O vector (iov) data transport – the vdev uses an iov structure to move data from virtqueues to the hardware driver

For more information about what to do in a guest when implementing VIRTIO, see "Discovering and connecting VIRTIO devices" in the "Using Virtual Devices, Networking, and Memory Sharing" chapter.

Guest-to-hardware communication with virtqueues

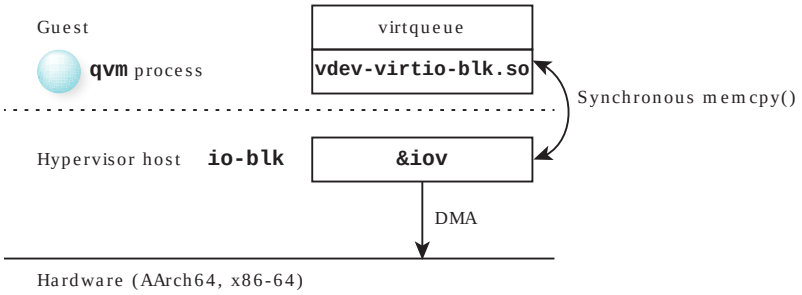
Assuming a Linux guest, the `virtio-blk` implementation in a QNX hypervisor system works as follows:

1. A VM is configured with a `virtio-blk` vdev.

   This vdev includes a `delayed` option that can be used to reduce boot times from a cold boot (i.e., from the time that the host system starts booting). It lets the VM launch and the guest start without confirming the presence of the hardware device. For example, the guest can start without the underlying block device specified by the `virtio-blk hostdev` being available (see vdev virtio-blk in the "Virtual Device Reference" chapter).
2. The `qvm` process instance hosting the guest reads its configuration file and loads the **vdev-virtio-blk.so**.

   This vdev's `hostdev` (host device) option specifies the mountpoint/file path that will be opened and passed to the guest as **/dev/vda**. For example, the `hostdev` option can pass the path **/dev/hd1t131** to the guest as **/dev/vda**. You can specify multiple `hostdev` options for a guest.
3. The `qvm` process hosting the guest launches, and the guest starts.
4. After the guest opens the locations specified by the vdev's `hostdev` options, file descriptors refer to each of these locations. The `qvm` process instance owns these file descriptors, but the microkernel manages them.

Figure 1Data is passed from virtqueues in the guest to the hardware. The guest and the `qvm` process share the same address space.



The guest can now read and write to hardware. For example, on a Linux application read:

- The `io-blk` driver pulls blocks either from disk cache or from hardware DMA into iov data structures.
- The iov data structures are copied to virtqueues using synchronous *memcpy()*, as shown in the figure above.
- The Linux `virtio_blk` kernel module provides virtqueue data to the application.

Similarly, on a Linux application write:

- The Linux `virtio_blk` kernel module fills virtqueues.
- On a flush or kick operation, the `qvm` process instance uses synchronous *memcpy()* to copy the virtqueue data directly to the iov data structures in the multithreaded `io-blk` driver.
- The `io-blk` driver writes to disk (DMA of iov directly to disk).

For both reading and writing, the Linux filesystem cache may be used.

Page updated: August 11, 2025

Design Safe States

The QNX Hypervisor supports two Design Safe States (DSSs).

CAUTION:
When you are running the non-safety QNX Hypervisor variant and not adhering to the safety manual, there is no guarantee that the hypervisor will enter into one of its DSSs following an undefined condition in a VM or the hypervisor itself.

If an internal or external detection mechanism alerts the hypervisor of a condition it is not designed to handle in any other way, the hypervisor should do one of the following:

VM DSS (local DSS)

If an undefined condition is confined to a VM, the hypervisor terminates that `qvm` process instance (e.g., with a SIGSEGV signal). Terminating the hosting `qvm` process instance terminates its guest.

The hypervisor host continues to run normally after it terminates a `qvm` process instance. You can design your system to take appropriate action after moving a guest to its local DSS; for example, you can reconstruct the VM and reboot the guest.

Hypervisor host DSS (global DSS)

Since the QNX Hypervisor comprises `qvm` integrated with the QNX OS microkernel, any abnormal conditions that cause the QNX OS microkernel to move to its DSS are viewed as causing the hypervisor to move to its global DSS. That is, if the undefined condition isn't confined to a VM, the hypervisor shuts down.

WARNING:
The hypervisor host kernel initiates a shutdown sequence on the CPU where the undefined condition is found. The last thing the kernel's shutdown sequence does is to trigger the kernel reboot callout, which must be provided in the BSP. This callout must ensure that it places the entire system in a safe state. Typically, this means rebooting the system, which will reset all processors and peripherals.

Page updated: August 11, 2025

Checking whether an OS is in a virtualized environment

The hypervisor provides a mechanism for an OS to determine if it is running in a virtualized environment and in particular, a QNX hypervisor environment.

ARM platforms

On ARM platforms, check the `model` property in the Flattened Device Tree (FDT) that describes the underlying system. If this property is set with `QVM-v8A`, then the OS is hosted in a QNX hypervisor VM (i.e., it is a guest OS).

You can obtain the VMID through the vdev API; for details, refer to the *qvm_guest_vmid()* entry in the *Virtual Device Developer's API Reference* which is available in our GitLab repository at https://gitlab.com/qnx/hypervisor.

x86 platforms

For x86 platforms, see your hardware documentation to learn which CPUID register bit your OS needs to check to know if it is running in a hypervisor, and where in the register it needs to look for the VM ID string. The ID string for QNX hypervisor VMs is “QNXQVMBS”.

Below is an example of C code that a QNX OS might use to check if it is running in a QNX hypervisor:

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <arpa/inet.h>

int cpuid(uint32_t id, uint32_t *regs) {
    asm volatile ("cpuid"
        : "=a" (regs[0]), "=b" (regs[1]), "=c" (regs[2]), "=d"
(regs[3])
        : "a" (id), "c" (0));
    return 0;
}

static const char * const qnxstr = "QNXQVMBS";

int main(int argc, char *argv[])
{
    uint32_t regs[4];

    cpuid(1, regs);
    if ((regs[2] & (1 << 31)) == 0) {
        puts("We are not running in a hypervisor");
        return 1; /* not a hypervisor */
    }

    /* It is a hypervisor, but is it OUR hypervisor? */
    cpuid(0x40000000, regs);
    regs[1] = htonl(regs[1]);
```

Linux and Android

You can adapt the C code above for Linux and Android on x86, or run the `cpuid` utility.

Page updated: August 11, 2025

QNX Hypervisor: Protection Features

This chapter explains the main features used by the QNX Hypervisor to protect itself and its guests. Many of these features are part of the design, while others are configurable.

Page updated: August 11, 2025

Handling a qvm termination

You can register to be informed if a `qvm` process instance terminates, so you can take appropriate action in response to the termination.

See "Shutting down guests" in the "Booting and Shutting Down" chapter for more information about shutting down guests.

Controlled termination

You can write a minimalist vdev that can initiate appropriate actions when its `qvm` process instance is about to terminate; for example, because its guest has shut down or become unresponsive. The samples below are from a fictitious `vdev qvmterm` that includes only a control function, which registers to receive a callback only if its `qvm` process instance terminates.

The vdev's control function is implemented as follows:

```
qvmterm_control(vdev_t *vdp, unsigned const ctrl, const char *const
arg) {
  struct term_state *state = vdp->v_device;

  switch (ctrl) {

    case VDEV_CTRL_TERMINATE:
    // The qvm is terminating.
    // Do any cleanup necessary for the vdev: end any worker threads
    // it created, close its connections to backend drivers, etc.
    // In this example, one worker thread was created by this vdev from
    // the vdev_thread_create() function when the guest was configured
    // (VDEV_CTRL_GUEST_CONFIGURED message block).
    // End the thread.
    pthread_cancel(state->thread);
    pthread_join(state->thread, NULL);
    break;
  }
```

The vdev populates its factory structure (`qvmterm_factory`) with the control function and the safety variant requirement but nothing else:

```
qvmterm_register(void) {
  static struct vdev_factory qvmterm_factory = {
    .next = NULL, // patched
    .control = qvmterm_control,
    .vread = NULL,
    .vwrite = NULL,
    .option_list = NULL,
    .name = NULL, // patched
    .factory_flags = VFF_NONE,
    .acc_sizes = 1u << 1,
    // Make sure we have local data space for our internal structure.
    .extra_space = sizeof(struct qvmterm_state),
    // For the safety version of vdev-qvmterm, we require safety.
    // For the non-safety version we do not require safety.
    //.safety = VDEV_SAFETY_SELECTED,
  };
  vdev_register_factory(&qvmterm_factory, QVM_VDEV_ABI);
}
```

For more information about writing vdevs and for vdev source code samples, see the *Virtual Device Developer's Guide* and *Virtual Device Developer's API Reference* in the QNX Hypervisor GitLab Repository at https://gitlab.com/qnx/hypervisor.

qvm crash

To manage an urgent `qvm` process instance termination in response to an unknown state, you can run a hypervisor host process outside the `qvm` that handles the termination. This process includes code to perform the system cleanup and hardware shutdown tasks that would be looked after in the VDEV_CTRL_TERMINATE message block in the fictional `vdev qvmterm` described above.

For a list of `qvm` exit codes, see "qvm process exit codes" in the "Monitoring and Troubleshooting" chapter.

Page updated: August 11, 2025

DMA device containment

The hypervisor uses an IOMMU/SMMU Manager to ensure that no pass-through DMA device can access host-physical memory to which it has not been explicitly granted access.

NOTE:
The `smmuman` service is a QNX OS component. It is described here for the convenience of hypervisor users. For more complete information, see the *SMMUMAN User's Guide*.

DMA devices and IOMMUs/SMMUs

A non-CPU-initiated read or write is a read or write request from a Direct Memory Access (DMA) device (e.g., GPU, network card, sound card). An IOMMU (SMMU on ARM architectures, VT-d on x86 architectures) is a hardware/firmware component that provides translation and access control for non-CPU initiated reads and writes, similar to the translation and access control that second-stage page tables provide for CPU-initiated reads and writes.

Hardware limitations

On some boards, the IOMMU/SMMU hardware doesn't provide the `smmuman` service the information it requires to map individual devices and report their attempted transgressions.

For example, on some ARM boards, the IOMMU doesn't provide the `smmuman` service the information it requires to identify individual PCI devices. Similarly, on x86 boards, the VT-d hardware can't identify or control individual MMIO devices that do DMA. Additionally, some boards might not have enough session identifiers (SIDs) to be able to assign a unique SID to every hardware device, so multiple hardware devices may have to share an SID.

These sorts of limitations mean that on these boards, the `smmuman` service may not be able to map individual devices. In a hypervisor system, the service may be able to map devices passed through to a guest to the memory allocated to that guest's VM, and report attempts by these devices to access memory outside the VM's memory. But it may not be able to report attempts by individual devices to access memory outside their allocated memory but inside the VM's allocated memory.

Responsibilities of the `smmuman` service

The `smmuman` service works with the IOMMUs or SMMUs on supported hardware platforms to:

- Manage guest-physical memory to host-physical memory translations and access for non-CPU initiated reads and writes (i.e, for DMA devices).
- Ensure that no pass-through device can access host-physical memory outside its mapped (permitted) host-physical memory location.

Using `smmuman` in a hypervisor system

In a QNX hypervisor system, the `smmuman` service runs in the hypervisor host domain and is launched at host startup.

When launched, the `smmuman` service learns which IOMMUs/SMMUs on the board control which DMA devices. Later, `qvm` process instances can ask `smmuman` to inform the board IOMMUs/SMMUs of the memory locations that DMA devices are permitted to access.

With the information about DMA devices obtained at startup, during operation the IOMMUs/SMMUs will know to reject any attempts by DMA devices to access memory outside their permitted locations.

During operation, `smmuman` monitors the system IOMMUs/SMMUs. If all DMA devices are behaving as expected, `smmuman` does nothing except continue monitoring. However, if a DMA device attempts to access memory outside its permitted locations, the IOMMU/SMMU hardware rejects the attempt, and `smmuman` records the incident internally.

Clients may use the `smmuman` API provided in `libsmmu` to obtain the information and output it to a useful location, such as the system logger (see `slogger2` in the QNX OS *Utilities Reference*). You can then use this information to learn more about the source of the illegal memory access attempt, and troubleshoot the problem.

`smmuman` and `qvm`

The `qvm` process instances in a QNX hypervisor system are `smmuman` clients:

- If `smmuman` *is not* running, `qvm` process instances refuse to include in their VMs any pass-through devices they know are DMA devices, and refuse to start (see the `loc` option's "d" and "n" access type attributes in the "Configuration" chapter).
- If `smmuman` *is* running and reports that it doesn't know how to handle a DMA pass-through device, the `qvm` process will report a problem and exit.

There is one exception to the above rules of behavior: unity-mapped guests. A guest is unity-mapped if all guest-physical memory allocated in the guest's VM is mapped directly to the host-physical memory (e.g., `0x80000000` in the guest maps to `0x80000000` in the host).

If a guest is unity-mapped, if `smmuman` isn't running or doesn't know how to handle a device, the `qvm` process instance hosting the guest will report the problem as expected, but it will also start.

WARNING:
Unity-mapping may be useful in some limited circumstances for testing and experimentation. However, unity-mapping circumvents the memory abstraction fundamental to a virtualized environment, and should not be used in a production system.

Information about IOMMU/SMMU control of DMA devices

Information about which IOMMU/SMMU controls which DMA device may be available in different locations, depending on the board architecture and the specific board itself (e.g., in ACPI tables, in board-specific code). To obtain this information, `smmuman` queries the firmware, and accepts user-input configuration information.

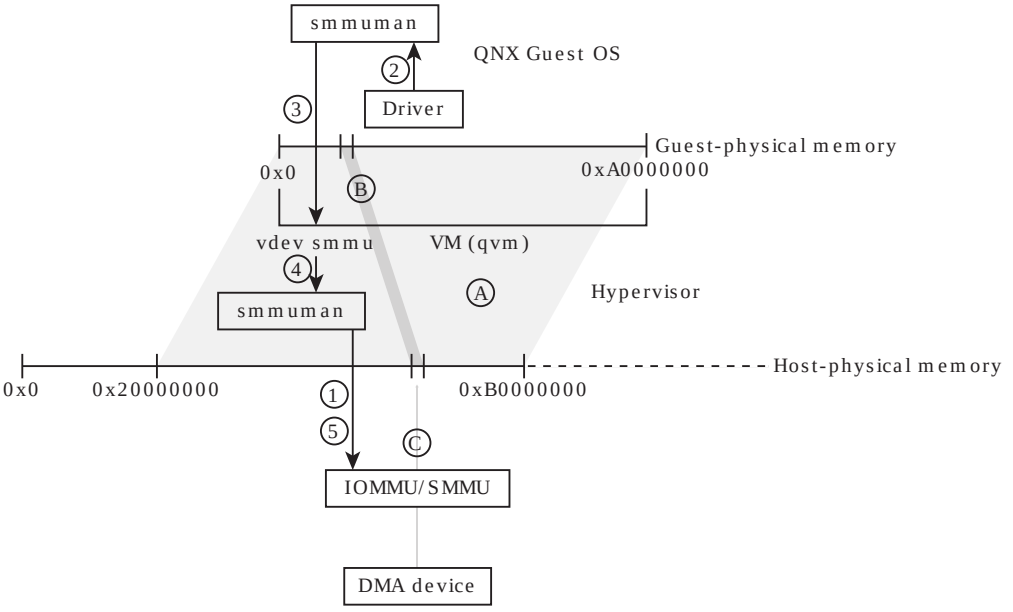
By default, the user-input information overrides the information obtained from the board, so that it can replace corrupt or otherwise unreliable information on the firmware. However, the user may request that `smmuman` keep the board-obtained information and merge it with the user-input information.

The `smmuman` service in QNX guests

If you are implementing pass-through devices anywhere on a system with a QNX guest, you should implement the `smmuman` service in the hypervisor host *and* in your QNX guest. Your guest will use the smmu vdev, which implements for a guest the IOMMU/SMMU functionality needed in a VM (see `vdev smmu` in the "Virtual Device Reference" chapter).

The figure below illustrates how the `smmuman` service is implemented in a hypervisor guest to constrain a pass-through DMA device to its assigned memory region:

Figure 1The `smmuman` service running in a guest uses the `smmuman` service running in the hypervisor.



For simplicity, the diagram shows a single guest with a single pass-through DMA device:

1. In the hypervisor host, the VM (`qvm` process instance) that will host the guest uses the `smmuman` service to program the IOMMU/SMMU with the memory range and permissions (A) for the entire physical memory region that it will present to its guest (e.g., `0x20000000` to `0xB0000000`).

This protects the hypervisor host and other guests if DMA devices passed through to the guest erroneously or maliciously attempt to access their memory. It doesn't prevent DMA devices passed through to the guest from improperly accessing memory assigned to that guest, however.

2. In the guest, the driver for the pass-through DMA device uses the SMMUMAN client-side API to program the memory allocation and permissions (B) it needs into the `smmuman` service running *in the guest*.
3. The guest's `smmuman` service programs the `smmu` vdev running in its hosting VM as it would an IOMMU/SMMU in hardware: it programs into this vdev the DMA device's memory allocation and permissions.
4. The `smmu` vdev uses the client-side API for the `smmuman` service running in the hypervisor host to program the memory allocation and permissions (B) requested by the guest's `smmuman` service into the board's IOMMU/SMMU.
5. The host's `smmuman` service programs the pass-through DMA device's memory allocation and permissions (B) into the board's IOMMU/SMMU.

The DMA device's access to memory (C) is now limited to the region and permissions requested by the DMA device driver in the guest (B), and the guest OS and other components are protected from this device erroneously or maliciously accessing their memory.

For the following reasons, the pass-through DMA device's memory allocation and permissions (B) can't simply be allocated when the guest is started, and must be allocated after startup:

- The VM hosting the guest doesn't know what memory mappings the guest will use for its DMA devices.
- A driver in the guest may change its memory mappings.
- A guest's DMA device driver (e.g., a graphics driver) may dynamically create and destroy memory regions while the guest is running.

WARNING:
You should also use the `smmuman` service running in the hypervisor host to program the board IOMMU/SMMUs with the memory ranges and permissions for the entire physical memory regions that every VM in your system will present to its guest. This way, any pass-through devices owned by these guests won't be able to access memory outside their guests' memory regions.

For more information about how to use the `smmuman` service in a QNX guest, see the *SMMUMAN User's Guide*.

Linux guests don't support the `smmuman` service; however, you should still use the `smmuman` service in the hypervisor host to program the IOMMU/SMMU with the memory range and permissions for the entire physical memory regions that the hypervisor host will present to each Linux guest.

Running `smmuman` in guests on ARM platforms

To run the `smmuman` service, guests running in QNX hypervisor VMs on ARM platforms must load **libfdt.so**. Make sure you include this shared object in the guest buildfiles.

NOTE:
The `libfdt` library is certified for internal use only. You must not use this library except where directed by QNX and only in context of that direction.

Page updated: August 11, 2025

Protection strategies

The QNX Hypervisor implements multiple strategies to protect itself and its guest OSs, and to mitigate the undesired effects of faults.

Because it uses the QNX OS microkernel, the QNX Hypervisor can rely on the safety-related techniques employed by the microkernel.

Types of interference

Interference comes in many types and differs according to whether the components of a system are actively cooperating or are meant to be completely independent. The following is an incomplete list. One component may:

- Rob another component of system resources (file descriptors, mutexes, flash memory, etc.). By periodically using and not releasing a file descriptor, one process could eventually consume all of the system's file descriptors and prevent a crucial process from opening a file in the flash memory when it needs to.
- Rob another component of processing time, preventing it from completing its tasks. By performing a processor-intensive calculation or by entering a tight loop under a failure condition, a process could prevent a critical process from running when it needs to.
- Access the private memory of another component. In the case of read access, this may be a security breach that could lead to a safety problem later; in the case of write access, this could immediately create a dangerous situation.
- Corrupt the data shared with another component, causing the other component to behave in an unexpected and potentially unsafe manner.
- Create a deadlock or livelock with another component with which it is cooperating. In either case, the system makes no forward progress, allowing a dangerous situation to arise through inaction. The circumstances that give rise to deadlocks and livelocks are generally subtle and, because of their temporal nature, can seldom be detected or reproduced by testing.
- Break its contract with a cooperating component by “babbling” (sending messages at a high rate or repeating messages) or sending messages with incorrect data.

Use of privilege levels

The QNX Hypervisor uses the hardware-supported privilege levels to ensure that the system is protected from errant code (see “Manage privilege levels”).

Security

The QNX Hypervisor inherits the security features of the QNX OS. These features include mechanisms that secure the hypervisor host domain. For example, a device driver running in the host domain may be made more secure by setting access control lists and forcing authentication (PAM).

Security policies can be applied to the VM managers (the `qvm` processes). In this way, the hypervisor host can integrate with the system designer's security policy and accommodate vdev-specific security needs. For hypervisor guests, security policies managed by the `secpol` utility can constrain the procmgr abilities required and the path permissions created by host-domain processes (and hence those of any loaded vdevs). But, these policies can't in any way constrain the VM as defined by its configuration (***.qvmconf**) file. For instance, a security policy can't prevent a `qvm` process instance from passing through a particular device or memory region to a guest.

For more information about security policies, refer to the `secpol` entry in the QNX OS *Utilities Reference*, and the *System Security Guide*.

Temporal isolation

The QNX Hypervisor uses the following components to ensure temporal isolation between the hypervisor host and the guests, and between the VMs and their guests:

- any temporal partitioning that is configured in the host
- any third-party solutions for configuring QNX OS scheduling policies
- the OS's priorities and scheduling to ensure proper management of conflicting requests for CPU resources

Spatial isolation

The guest software running inside a VM is separate from the host microkernel, so a bug in the guest, or a malicious corruption of one, can't hinder correct operation of the host microkernel.

Guests' access to memory is through intermediate stage tables (for ARM, Stage 2 page tables; for x86, Extended Page Tables (EPT)). Memory allocation and device assignment are static; they are defined in the device tree, validated at startup, then loaded. They can't be changed after the system has started (see "Memory").

The hypervisor implements a SMMU manager (`smmuman`) to ensure that the hardware SMMUs trap and fail attempts by pass-through Direct Memory Access (DMA) devices to access memory outside their allocated memory regions. The `smmuman` service tracks these violations and retains this information until queried for it (see "DMA device containment" in this chapter).

Data isolation

In a hypervisor system, the hypervisor host configuration and the VM configuration assign physical devices to either the hypervisor host or a VM. Startup of `qvm` process instances fails if a device is assigned to more than one VM or to a VM and the hypervisor host. Thus, a single entity (hypervisor host or guest OS) will have exclusive control over each device. Data from one device is passed to an entity that doesn't own the device only when that entity expressly requests it.

Managing unresponsive servers

The hypervisor can use the `server-monitor` utility to watch designated servers and take action if these servers don't handle an unblock pulse due to a timeout. That is, the hypervisor can protect itself from a server becoming unresponsive.

When `server-monitor` is implemented in a system, it watches a list of servers and takes a specified action if any of them fails to respond to unblock pulses within a specified time interval. Thus, if a client is unable to unblock a server by sending it an unblock pulse, `server-monitor` can remedy the situation by taking appropriate actions. These actions range from doing nothing, to sending a signal to the server to get it to respond, to rebooting the system (see `server-monitor` in the QNX OS *Utilities Reference*).

Watchdogs

Although describing the design and implementation of watchdogs is outside the scope of this document, you can implement your own watchdogs within a VM by using the provided watchdog vdevs (see "Watchdogs" in this chapter, and "`vdev wdt-ib700`" and "`vdev wdt-sp805`" in the "Virtual Device Reference" chapter).

Page updated: August 11, 2025

Watchdogs

QNX hypervisors provides virtual watchdog devices that you can use in a VM just as you would a hardware watchdog on a board in a non-hypervisor system.

WARNING:
It is your responsibility to determine how best to use hardware and software watchdogs for the hypervisor host as well as for guests. This is of particular importance in safety-related systems.

If you need advice about how to implement watchdogs, please contact your QNX representative.

Watchdogs in the hypervisor host

The QNX Hypervisor provides the same support for a watchdog as the QNX OS microkernel, of which it is a superset. If a hardware watchdog exists, you can use a board-specific utility to kick it, and use the High Availability Manager (HAM) or the System Launch and Monitor (SLM) service to manage the host in the event that a watchdog detects an anomaly in the host's behavior (see `ham` and `slm` in the QNX OS *Utilities Reference*).

To learn more about your board-specific watchdog kicker utility, see your BSP *User's Guide*.

Watchdogs in a guest

The watchdog vdevs implemented with QNX hypervisors emulate a hardware watchdog in a VM. A watchdog kicker utility running in the guest enables its virtual hardware watchdog, then writes at specific intervals to guest-physical registers monitored by the watchdog vdev to inform the watchdog that the guest is running. This is referred to as "kicking" the watchdog.

If the watchdog kicker fails to write to the registers within the required delay, the watchdog vdev can trigger an appropriate action, such as forcing the `qvm` process instance to exit. This triggering of a follow-up action is known as the *watchdog bite*.

Since the `qvm` process is a QNX OS process, you can trap its exit codes just like for any OS process. If you're implementing a watchdog service for your guests, you should configure your hypervisor host to trap the `qvm` process exit codes so you can decide what the host will do in the event that a `qvm` process instance terminates unexpectedly (see "qvm process exit codes" in the "Monitoring and Troubleshooting" chapter). These actions can range from simply logging the error and waiting for user intervention, to using the HAM to attempt to restart the `qvm` process and its guest OS.

For information about how to cause the guest to dump in response to a watchdog bite, see "Getting a guest dump during a crash" in the "Monitoring and Troubleshooting" chapter.

CAUTION:
When a watchdog vdev terminates a `qvm` process instance in an orderly manner, this termination necessarily also stops the guest from executing. From the guest's perspective, this is *not* an orderly termination.

Implementing a watchdog in a guest

To implement a watchdog service in a QNX guest running in a QNX hypervisor VM, you need to:

- Include the appropriate watchdog vdev (`wdt-sp805` for ARM or `wdt-ib700` for x86) in the configuration for the VM that will host the guest (see vdev wdt-sp805 and vdev wdt-ib700 in the "Virtual Device Reference" chapter).
- Implement a watchdog kicker in the guest, and configure it to kick the watchdog vdev (for QNX guests the kicker is provided in the BSP, see "QNX guest watchdog kicker (wdtkick)" below).

  For Linux and Android guests, see "Implementing a watchdog in a Linux or Android guest" below.
- Use the HAM and/or SLM utilities in the host to manage the `qvm` process in the event that a watchdog detects an anomaly in a guest's behavior and its hosting `qvm` process instance has exited. For example, the SLM could restart the `qvm` process instance with the same VM configuration it had before it exited (see `ham` and `slm` in the QNX OS *Utilities Reference*).

NOTE:
The watchdog vdevs emulate a subset of the functions provided by hardware watchdogs. Refer to the chip documentation for information about your hardware watchdog (e.g., the SP805 documentation at the

ARM Documentation Center).

Starting and stopping watchdogs and their kickers

QNX hypervisor BSPs include scripts for starting and stopping watchdogs and their kickers in QNX guests. These scripts are located in the BSP's **scripts** directory:

`watchdog-start.sh`
: Starts the watchdog vdev in the VM (`qvm` process instance), and starts the kicker utility (`wdtkick`) in daemon mode in the guest.

`watchdog-stop.sh`
: Stops the kicker utility in the guest, then immediately writes to the guest-physical register the value needed to stop the watchdog vdev in the VM before the vdev notices that the kicker has stopped.

QNX guest watchdog kicker (`wdtkick`)

The `wdtkick` utility is board-specific for QNX guests (see `wdtkick` in the QNX OS *Utilities Reference*). It is shipped in the BSPs for supported boards, and is located at **src/hardware/support/wdtkick**.

The buildfiles for the QNX guests made available with the hypervisor include commented-out sections with configurations for `wdtkick`. To use this utility, you can uncomment these sections and rebuild your guest.

Below are examples of `wdtkick` configurations for ARM and for x86 guests.

`wdtkick` in an ARM guest (SP805 emulation)

Kick the watchdog every second (1), and set the timeout period to three (3) seconds, which in fact translates to six (6) seconds because the timer on ARM platforms counts down twice and asserts the reset only on the second timer expiry:

```
wdtkick -v -a 0x1C0F0000 -t 1000 -E 8:3 -W 0:0x47868C0
```

where:

- `-v` sets the verbosity
- `-a 0x1C0F0000` sets the base address the watchdog will use in guest-physical memory
- `-t 1000` sets the watchdog kick interval to one second (1000 milliseconds)
- `-E 8:3` sets the offset at which to write in the watchdog the register (8) to enable the timer, and the mask to use when writing
- `-W 0:0x47868C0` sets the offset at which to write in the watchdog register (0), and the value to write there (`0x47868C0`, which specifies 3 seconds at 25 MHz)

NOTE:
Refer to the SP805 specifications and your board manufacturer's documentation for more information about how to configure your watchdog and watchdog kicker.

`wdtkick` in an x86 guest (IB700 emulation)

Kick the watchdog every five (5) seconds, and set the timeout period to 10 seconds:

```
wdtkick -v -a 0x441 -t 5000 -w 8 -W 2:0xA
```

where:

- `-v` sets the verbosity
- `-a 0x441` sets the base address the watchdog will use in guest-physical memory
- `-t 5000` sets the watchdog kick interval to five seconds (5000 milliseconds)
- `-w` sets the width of the watchdog write register to eight (8) bits
- `-W 2:0xA` sets the offset at which to write in the watchdog register (2), and the value to write there (`0xA`, which specifies 10 seconds)

NOTE:
Refer to the IB700 specifications and your board manufacturer's documentation for more information about how to configure your watchdog and watchdog kicker.

Watchdog kicker configuration

You can enter the watchdog kicker configuration via the command line at startup, or you can modify your guest startup to store it in the guest system page's *hwinfo* section (see the “System Page” chapter in *Building Embedded Systems*).

For up-to-date information about your board-specific watchdog kicker utility, see the **wdtkick.use** file included with the BSP.

For information about the watchdog vdevs, see vdev wdt-sp805 (ARM) and vdev wdt-ib700 (x86) in the “Virtual Device Reference” chapter.

Implementing a watchdog in a Linux or Android guest

If you want to use a watchdog in a Linux or Android guest, you must do the following:

1. Enable the watchdog module — the Linux or Android kernel must include the correct watchdog kernel module for your target (IB700 or SP805).

   In most Linux and Android distributions, this module isn't enabled by default. See **menuconfig**, and the Linux kernel configuration documentation's “Device Drivers” section for details on how to rebuild a Linux kernel with the watchdog module enabled.

2. Implement a Linux or Android application or shell script to control the watchdog. Documentation on how to control a Linux watchdog is publicly available. A useful example can be found at www.kernel.org/doc/Documentation/watchdog/watchdog-api.txt.

Page updated: August 11, 2025

Virtual registers (guest_shm.h)

The **guest_shm.h** public header file includes definitions for guests using the shmem vdev.

GUEST_SHM_*

Synopsis:

```
#define GUEST_SHM_MAX_CLIENTS   16
#define GUEST_SHM_MAX_NAME      32
#define GUEST_SHM_SIGNATURE 0x4d534732474d5651
```

Data:

The GUEST_SHM_* constants include the following:

GUEST_SHM_MAX_CLIENTS
: Maximum number of clients allowed to connect to a shared memory region (16)

GUEST_SHM_MAX_NAME
: Maximum length allowed for a region name, in bytes (32)

GUEST_SHM_SIGNATURE
: Signature value to verify that the vdev is present (`0x4d534732474d5651`)

guest_shm_control

Register layout for a region control page

Synopsis:

```
struct guest_shm_control {
  uint32_t    status;
  uint32_t    idx;
  uint32_t    notify;
  uint32_t    detach;
};
```

Data:

The members of `guest_shm_control` include:

status
: Read only. Lower 16 bits: pending notification bitset; upper 16 bits: current active clients (see guest_shm_status below).

idx
: Read only. Connection index for this client.

notify
: Write a bitset here to indicate which clients to notify.

detach
: Write here to detach your client from the shared memory region.

guest_shm_factory

Register layout for shared memory factory page registers

Synopsis:

```
struct guest_shm_factory {
  uint64_t    signature;
  uint64_t    shmem;
  uint32_t    vector;
  uint32_t    status;
  uint32_t    size;
  char        name[GUEST_SHM_MAX_NAME];
  uint32_t    find;
};
```

Data:

The members of `guest_shm_factory` include:

signature

Read only. Is GUEST_SHM_SIGNATURE (see "GUEST_SHM_*" above).

shmem

Read only. The location of the shared memory in guest-physical memory.

vector

Read only. The interrupt number for this shared memory region.

status

Read only. The status of the last creation attempt (see "guest_shm_status" below).

size

The size of the requested named shared memory region, in multiples of 4 KB pages. A write with this value creates a shared memory region, if the region with the specified name and size doesn't already exist.

name

The name of the shared memory region.

find

Whether to find an existing shared memory connection.

guest_shm_status

Status of last request to create a named shared memory region

Synopsis:

```
enum guest_shm_status {
  GSS_OK,
  GSS_UNKNOWN_FAILURE,
  GSS_NOMEM,
  GSS_CLIENT_MAX,
  GSS_ILLEGAL_NAME,
  GSS_NO_PERMISSION,
  GSS_DOES_NOT_EXIST,
};
```

Data:

The `guest_shm_status` enumeration defines these values:

GSS_OK

The region was successfully created.

GSS_UNKNOWN_FAILURE

The region creation failed due to an unknown reason.

GSS_NOMEM

There was insufficient memory to create the region.

GSS_CLIENT_MAX

The region can't be connected to because it is already being used by the maximum permitted number of guests.

GSS_ILLEGAL_NAME

The region creation failed because the specified region name is illegal.

GSS_NO_PERMISSION

The region creation failed because the process attempting to create it has insufficient permissions.

GSS_DOES_NOT_EXIST

An attempt to find a named shared memory region failed.

PCI_VID_* and PCI_DID_*

Vendor IDs and Device IDs

Synopsis:

```
#define PCI_VID_BlackBerry_QNX   0x1C05
#define PCI_DID_QNX_GUEST_SHM    0x0001
```

Memory sharing

Guests in a hypervisor system can use shared memory to pass data to each other or to the hypervisor host.

In a QNX hypervisor system, client applications running in guests can create and manage shared memory, and use shared memory regions to exchange data. Note, however, that these memory regions are ultimately created and controlled by the hypervisor, not by the guest. Host applications may also create shared memory regions or attach to them if permission allows.

The **hypervisor-shmem-examples-*.tgz** archive available with QNX hypervisors includes source code for sample memory-sharing programs: **ghstest.c** for a QNX guest and **hhstest.c** for the hypervisor host.

To write hypervisor host modules that can share data with guests, you need to use the Virtualization API (**libhyp.a**). This is described in the *Virtualization API Reference* that's not included with the QNX hypervisor documentation. To obtain this additional documentation and support for writing host modules, contact your QNX representative.

How shared memory works

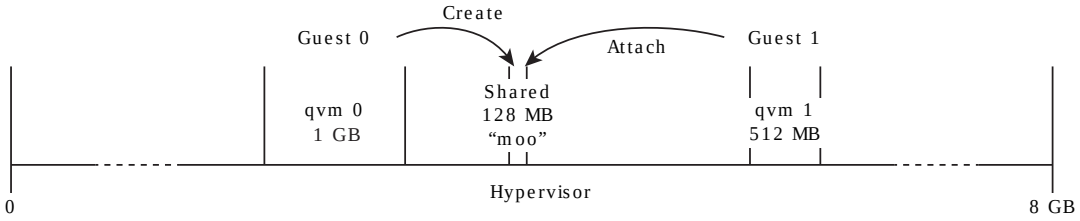
To use shared memory, a client application in a guest or in the hypervisor host needs:

- a mapping of the shared memory region
- a hardware interrupt it can use to signal other users of shared memory regions that this region has been updated

The hypervisor provides the shmem vdev, which implements setting up the shared memory mapping and the interrupts you need to use shared memory. This vdev provides additional functionality to simplify using shared memory, including:

- shared memory region names (a QNX hypervisor system may include multiple named shared memory regions)
- selective signaling (e.g., signal Guest 0, but not Guest 3)
- signal knowledge (the ability to know which guests have been signaled)

Figure 1A 128 MB memory allocation (“moo”) shared by Guest 0 and Guest 1



The figure above illustrates memory sharing between two guests. Guest 0 attempts to attach to a 128 MB shared memory area (“moo”) first. Since no such area exists at the specified location, the attempt to attach creates the area and allows the guest to attach to it. Guest 1 can simply attach to the same name to share data with Guest 0.

NOTE:
All connections to a shared memory region are peers. That is, there is no distinction between the guest that creates a shared memory region and the guest that attaches to it. Simply, the first attempt to attach to a shared memory region creates the region. As far as the guest is concerned, it simply attaches to the region.

This design avoids ordering problems where the system designer would have to make sure that one guest always comes up and creates the shared memory region before another guest tries to attach to it.

Page updated: August 11, 2025

Configuring the VMs

The shmem vdev can be added to a VM configuration, just like any other vdev.

To include the shmem vdev in a VM that will host a guest using shared memory services, simply add the vdev to the `qvm` configuration. Make sure that it's in the `qvm` configuration for every VM with a guest that will need to use the shared memory services. For example:

```
# qnxcluster.qvmconf example
system cluster
ram 1024M
cpu
cpu
load /emmc/QNX_cluster.ifs
vdev ioapic
        loc 0xf8000000
        intr apic
        name myioapic
vdev ser8250
        intr myioapic:4
vdev timer8254
        intr myioapic:0
vdev mc146818
vdev shmem
vdev pckeyboard
```

In the configuration above, the `vdev shmem` line adds the shmem vdev. This a PCI device. But if you specify its `loc` and `intr` properties, the guest will see it as an MMIO device at the specified location. For example, this configuration:

```
vdev shmem
  create TEST1,0xf0000  # name is TEST1 with size of 0xf0000
  loc 0x10000000    # location of factory page
  intr myioapic:10  # hardware interrupt used for signaling
```

causes the `qvm` process to create a shared memory region "TEST1" when it starts and assembles the VM. The configuration assumes that a vdev ioapic called "myioapic" has been specified.

NOTE:
Since the `qvm` process creates the shared memory region when it starts up (rather than when a guest attempts to attach to it), you can ensure there's enough underlying physical memory on the host system for the region.

For more information about `qvm` configuration files and how to use them to assemble a VM, see "Assembling and configuring VMs" in the "Configuration" chapter.

Using the `allow` and `deny` options

You can use the shmem vdev's `allow` and `deny` options to create a restrictions list of shared memory regions the guest may and may not access (see vdev shmem in the "Virtual Device Reference" chapter).

The hypervisor enforces the access and denial properties specified in the restrictions list at runtime. This enforcement ensures that code in a guest can't access specified shared memory regions without explicit permission to do so, or create vulnerable shared memory regions that another guest in the hypervisor system might find and mine for information.

Page updated: August 11, 2025

Factory and control pages

The QNX hypervisor shared memory implementation uses *factory pages* and *control pages*.

Factory and control pages store the shmem vdev's virtual registers. A driver in the guest can access these registers in guest-physical memory, and interract with them as it would physical registers, reading from and writing to them at specified offsets.

Factory and control pages are the same size as QNX hypervisor kernel pages (4 KB).

Factory pages

Including shmem vdevs in the configuration for a VM (see "Configuring the VMs") causes the underlying `qvm` process to create a factory page for the VM. Typically, there is only one shmem vdev per `qvm` configuration file, hence one factory page per VM.

A factory page contains information about the shared memory, including a field with the guest-physical address for each shared memory region's control page. This field changes value as the guest creates or attaches to different shared memory regions.

A factory page may be located anywhere in unallocated memory that the guest running in the VM can access. For example, assuming no other memory has been allocated yet, if you allocate 192 MB of RAM (`ram 192M` set in the `qvm` configuration), you can place the factory page outside this allocated RAM at `0x10000000` (256 MB). This address is a guest-physical address, *not* an actual physical address in hardware. The shmem vdev virtualizes the factory page for the guest. No other device may use this location.

When an application in the guest attempts to create a shared memory region, it describes the region to the hypervisor by writing to the shmem vdev's registers in the factory page. The hypervisor uses the factory page to create the new shared memory region and its control page, updating the vdev's registers.

Registers are described in the data types defined in the **qvm/guest_shm.h** header file (see "Virtual registers (guest_shm.h)" in this chapter).

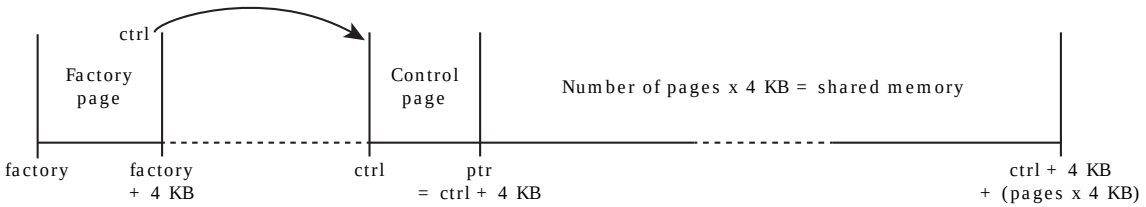
Control pages

A shared memory control page contains registers to which shmem vdev client applications can write to manage their relationship with a shared memory region and its other users (e.g., inform other users of a change to the region's contents, detach from the region).

The first request from a guest for a new shared memory region sets up the requested region, plus a control page for it. Every shared memory region has its own control page that prefixes the region; there are as many control pages as there are shared memory regions.

After a guest creates a shared memory region or attaches to it, the guest uses a field on the system page to locate the region's control page. When guests need to communicate with other guests, they write to the control page.

Figure 1A factory page with its pointer to the control page for a shared memory region.



The diagram above illustrates how a VM's factory page points to the control page that is prefixed to a shared memory region.

Page updated: August 11, 2025

Using the shared memory (shmem) vdev

The shmem vdev provides a simple mechanism for sharing memory regions between guests, or between guests and the hypervisor host.

For configuration information for this vdev, see vdev shmem in the "Virtual Device Reference" chapter.

shmem vdev driver

The shmem vdev appears to client application code running in a guest or in the hypervisor host as a device with a mapped register set in guest-physical or host-physical memory and the ability to raise interrupts.

This client code is simply a driver for the shmem vdev, rather than for some physical device (see "Virtual devices" in the "Understanding QNX Virtual Environments" chapter). The client running in a guest or in the host can use the shmem vdev register set to connect to an existing named shared memory region, or to create a new such region and connect to it.

Accessing a shared memory factory page

Two methods are available for specifying a shared memory factory page in the VM configuration: MMIO and PCI (see "Factory and control pages"). To locate a factory page, clients in a guest or the host must use the same method that was used to specify the factory page:

MMIO
: The factory page is directly mapped. Use the guest-physical address and the interrupt specified by the shmem vdev's `loc` and `intr` options in the VM configuration.

PCI
: (Default setting) The hypervisor assigns the factory page its location and interrupt. The factory page appears as a PCI device. The guest must run a PCI server, and clients can use the PCI server API to access the factory page and get location and interrupt information (see the "API Reference" chapter in the *PCI Server User's Guide*).

When your client application in the guest has obtained the factory page's base address in guest-physical memory, it can use *mmap()* to map the address into a virtual address. The relevant registers are in the `guest_shm_factory` data structure, which is defined in the **guest_shm.h** public header file.

Creating and connecting to a shared memory region

The shmem vdev is the factory; the client writes to virtual registers in the factory page, which causes the vdev to create shared memory regions if necessary.

That is, the factory page client (i.e., the shmem vdev driver) writes a name and size to the factory page. This action prompts the shmem vdev: if the specified shared memory region doesn't already exist, the shmem vdev creates it along with its control page, returning their location in guest-physical memory to the client; or, if the region already exists, the vdev simply returns the location. This create-on-first-use behavior allows guests to be started in any order; whichever guest is first to attempt to access a shared memory region creates the region.

When accessing a named shared memory region, the client application needs to do the following with the `guest_shm_factory` data structure for the factory page:

1. Write the shared memory name to `guest_shm_factory.name`, and the required size of the shared memory to `guest_shm_factory.size`; if the shared memory region of the specified name and size doesn't already exist, the shmem vdev will create it.
2. Check `factory.status` for the status of the named shared memory region.
3. Read `factory.shmem`, which holds the base guest-physical address of the `guest_shm_control` data structure. This structure provides the register layout for the shared memory region's control page, and the shared memory region itself.

Page updated: August 11, 2025

Starting and using guests

Below are instructions for booting guests in your hypervisor's VMs.

When the hypervisor has completed its startup, it is ready to host guests. You can't start a guest until the hypervisor has finished configuring the VM hosting this guest. You can configure your hypervisor's startup activities to automatically launch `qvm` processes that will load and launch guests after VM configurations are complete, or to wait for further input before starting any guests.

You know that your hypervisor has completed its startup successfully when you can perform any of the activities described in "Viewing hypervisor activity".

Prerequisites

To run a guest on the hypervisor, you need the following on your target board:

- a running hypervisor
- the IFS image (QNX OS) or kernel image (Linux) with the guest you want to run
- a configuration file for the guest; at a minimum, this file specifies the guest's assignments of RAM, CPUs, and virtual devices, and the path and name of the file with the guest's IFS

In addition to the above, Linux guests may require an initial RAM disk (**initrd**, **initramfs**) file.

Starting a guest

CAUTION:
When you start a `qvm` process and launch a guest, the terminal console will switch to the guest you have started. You may find it useful to get the IP address for the hypervisor host, then open a second terminal console via SSH into the hypervisor host domain before you start a guest (see "Determining where terminal consoles are connected" below).

To start a QNX guest OS, in the command line of your terminal program connected to the hypervisor:

1. Go to the directory with the `qvm` configuration file for the VM in which you will run your guest.
2. Run the `qvm` utility, pointing it to the configuration file for the guest you want to run.

For example, from the directories with the guests, run these commands for each guest type:

- QNX OS 8.0 guest: `qvm @qnx80/qnx80-guest.qvmconf`
- Linux guest: `qvm @linux/linux-guest.qvmconf`

The at sign (`@`) in front of the filename in the `qvm` command line designates a path to a `qvm` configuration file.

When it receives the instruction, the hypervisor should:

1. Launch a new instance of the `qvm` process for the VM that will host the guest OS.
2. Parse the configuration file (e.g., **qnx80-guest.qvmconf**) for the guest OS.
3. Load the IFS with the guest (e.g., **qnx80-guest.ifs**).
4. Start the guest OS.
5. Connect the terminal program to the guest, so that output from the guest OS boot goes to the terminal on the development host system; this is the default configuration, which you can change.

Things to try

Below are some things you can try in order to get to know your virtualized system.

Determining where terminal consoles are connected

Determining which terminal consoles are connected to the hypervisor host or to the guests can be difficult for a novice hypervisor user.

If you know a little about how your system is configured, you can easily determine what your terminal console is showing, however. Running `uname -a`, and `pidin info` or `ps` should give you the information you need. For example, a `pidin info` command on a terminal connected to a QNX guest configured to run on two vCPUs would show two CPUs, but the same command on a terminal connected to the hypervisor host would show all CPUs on the board, which may be, say, four.

The hypervisor host and QNX guests each have an **/etc/os.version** file. You can open these files to view information about the host or the guests.

Viewing thread activity

To view information about thread activity in a `qvm` process, use `pidin` in the command line of your terminal program connected to the hypervisor host, or the QNX Momentics IDE tool connected to the hypervisor host.

The `qvm` thread activity tells you about vCPUs, but not about the various services and applications each guest is running. To view thread activity within a guest from the IDE, you can create a new QNX target and enter the IP address of the guest. In this case, you must be running the `qconn` service on the guest, because this is the service that supports the IDE. If your configuration doesn't start this utility, you must start it manually (see qconn in the QNX OS *Utilities Reference*).

Virtual networking

To start virtual networking, on the command line for a console connected to a QNX guest, run the following script:

```
/scripts/network-start.sh
```

When the script completes, you can get an IP address for your guest, ping the guest, and perform other networking activities such as using SSH to access this guest, mounting an NFS share on it, etc.

You can do the equivalent on a Linux guest as well.

Block devices

To start a block device, on the command line for a console connected to a QNX guest, run the following script:

```
/scripts/block-start.sh
```

Try using the block device as you would use this type of device in a non-virtualized system.

The block device is mounted to the **/RAM disk** path on the guest. The VM is configured to use the `devb-ram` device on the hypervisor host. You can change the `qvm` configuration for the VM hosting your guest so that your guest's block device uses another device, such as a USB key.

Page updated: August 11, 2025

Shutting down guests

There are different methods for shutting down guests in a QNX hypervisor VM.

WARNING:
To avoid killing a driver in the midst of an I/O operation, and possibly leaving the hardware in an unrecoverable state, use a SIGTERM or SIGINT signal to terminate a `qvm` process.

For instructions on how to shut down the hypervisor, see "Shutting down the QNX hypervisor". For information on how to handle a VM termination, see "Handling a qvm termination" in the "QNX Hypervisor: Protection Features" chapter.

Internal (controlled) guest shutdown

The guest (more specifically, an application running on the guest OS) can initiate its own shutdown. For example, an application in a QNX guest could use the *shutdown_system()* function to shut down or reboot the guest. In short, any action that ends up calling the reboot kernel callout will work.

From the command line, use `shutdown` in a QNX guest, or `reboot` in a Linux or Android guest.

If the guest shuts itself down or attempts to reboot, the hypervisor ends the `qvm` process for the VM hosting the guest, and frees all resources used by this process, meaning it releases them to the host. This includes those resources used by the guest (see "Handling a qvm termination").

The host can't see into the guest, so if the host needs to perform actions *before* the guest shuts itself down, the guest must explicitly inform the host that it's shutting down. A simple way to do this is to:

1. Use the shared memory vdev (vdev shmem) to set up a small area of memory shared between the guest and the host (see "Memory sharing" in the "Using Virtual Devices, Networking, and Memory Sharing" chapter).
2. When it shuts down, the guest writes a pre-defined value to a location in this shared memory.
3. When it reads this value in the shared memory, the hypervisor host starts performing whatever tasks it needs to complete (e.g., masking interrupts to the guest, quiescing any physical devices) in response to the guest's shutting down. An explanation of quiescing physical devices is given in "Quiescing devices during guest shutdown".

If the guest has physical devices allocated to it, the guest should inform any other guests with which it might be sharing these devices that it is shutting down and thus, the devices will no longer be available.

External (uncontrolled) guest shutdown

Whenever possible, use a controlled shutdown from within a guest (`shutdown` in a QNX guest, or `reboot` in a Linux or Android guest). If you cannot do so, then an uncontrolled guest shutdown is possible as described here.

From your host system terminal, you can use a utility such as `slay` to deliver a signal to the `qvm` process for the VM hosting the guest (see `slay` in the QNX OS *Utilities Reference*):

- **SIGTERM or SIGINT**: Send a SIGTERM or SIGINT signal to the `qvm` process instance to emulate pressing the power button. For example: `slay -s SIGTERM qvm`. This makes `qvm` ask the guest nicely to terminate.

  On x86, this triggers the ACPI power button notification mechanism, which will allow the guest to cleanly shut down if it takes notice of it. Linux and Android guests (the latter if set up properly) have ACPI handling in place and will notice, but QNX guests won't. If you send a SIGTERM or SIGINT but the guest doesn't shut down, `qvm` assumes there's no handling installed in it. Sending a second signal will unconditionally terminate the guest.

  On ARM, there's no standard for power button notification so unless a guest-specific vdev is written, no handling is performed. Thus, sending a SIGTERM or SIGINT immediately terminates the guest with no notification.

- **SIGQUIT**: Send a SIGQUIT signal to the hosting `qvm` process instance to terminate it without notifying the guest regardless of any power-off handling in it. This emulates immediately cutting the power. For example: `slay -s SIGQUIT qvm`.

If you terminate a VM (`qvm` process instance), you must put into a quiescent state any hardware devices that were passed through to a guest in the VM. You can do this with a custom vdev. For information about writing custom vdevs, see the *Virtual Device Developer's Guide* in the QNX Hypervisor GitLab Repository at https://gitlab.com/qnx/hypervisor.

CAUTION:

If your guest is terminated in an uncontrolled manner, this guest may behave in an undefined manner on subsequent startups (e.g., it may have a corrupt filesystem).

Watchdogs

Guests can be configured to use a watchdog vdev in their hosting VMs. If a guest fails to kick its watchdog in time, the watchdog may trigger a SIGQUIT to terminate the hosting `qvm` process instance immediately. This is a common way of using a watchdog, but it is not the only possible action (see “Watchdogs”).

Page updated: August 11, 2025

Booting the QNX hypervisor host

You can boot and run the QNX hypervisor host without booting any guests. This helps you perform basic troubleshooting of your target system.

NOTE:
Many board vendors provide separate installation guides and User's Guides for their SoC solutions, which explain how to boot and run the hypervisor on their boards. The information in this *User's Guide* applies to all vendor solutions using the hypervisor version described herein unless restrictions or recommendations on options are provided.

On the target system, the hypervisor features must be enabled by providing the `-Q enable` option to the startup program, as described in "Configuring the hypervisor host". When you boot the target board, the QNX OS microkernel should then start with hypervisor mode enabled.

You can confirm that your target system supports running a QNX hypervisor by running the `qvm-check` utility. You can then look at the host's activity (see "Viewing hypervisor activity"), and boot the guests (see "Starting and using guests").

Page updated: August 11, 2025

Shutting down the QNX hypervisor

The following methods can be used to shut down the QNX hypervisor:

- If the QNX hypervisor encounters an undefined state, it will move to its Design Safe State (DSS); for details, see "Design Safe States" in the "QNX Hypervisor: Protection Features" chapter.
- All mechanisms available for shutting down or rebooting the QNX OS (e.g., `slay -s SIGTERM qvm`) are available to the QNX hypervisor. These mechanisms are available to applications running in the hypervisor host domain but *not* to guest OSs or their applications.

For instructions on how to shut down hypervisor guests, see "Shutting down guests".

Page updated: August 11, 2025

Quiescing devices during guest shutdown

When a guest shuts down, in either a controlled or uncontrolled shutdown, the underlying `qvm` process should try to quiesce any physical devices used by the guest. This ensures that the guest driver doesn't continue writing to physical memory, which can leave the hardware in an unexpected or even damaged state.

When a guest shuts down and hence, its device drivers go away, the host hardware can't be expected to stay in the same state because the devices may be operating asynchronously (e.g., doing DMA writes, generating interrupts). To protect the integrity of the hardware, the `qvm` process in which the guest runs can be designed to quiesce any such device (i.e., make it go dormant) when the guest terminates.

To follow this design, the `qvm` process can include a vdev that quiesces devices and configure this vdev so its callback function is run when the `qvm` process terminates, before the resources of driver processes are released. The callback function must do any required cleanup, including quiescence. This entails turning off any active device as quickly as possible so that when the `qvm` process goes away, the device no longer:

- writes to physical memory (very dangerous)
- generates interrupts (less dangerous because the hypervisor can acknowledge the IRQ and ignore it)

For information on defining a vdev's control function to register a callback that runs during process shutdown, see "Handling a qvm termination".

NOTE:
A guest *can* try to quiesce a device; it's just that the hypervisor can't depend on the guest doing so. For the non-safety QNX Hypervisor product, it is recommended but not required to provide a vdev to quiesce physical devices during shutdown.

Page updated: August 11, 2025

Starting VMs

When you start a VM, you can either enter all of the configuration information in the command line, or point the `qvm` process at a configuration file.

Starting a VM with a configuration file

The recommended method for starting a VM is to point the `qvm` process at a `qvm` configuration file. You must first navigate to the directory containing the configuration file that you want to use, before launching the `qvm` process, as follows:

```
% cd /guests/qnx-guest-1/

% qvm @config
```

where *config* is the name of the configuration file within the current directory (e.g., **qnx80.qvmconf**).

The `qvm` process opens the file and parses its contents, which define the VM (see "Assembling and configuring VMs").

NOTE:
You must launch the `qvm` process from within the directory containing the configuration file, or you'll get an error that the IFS can't be found. The reasons behind this requirement are that you can move the guest around and still launch it with the same command, and use a relative path for `load` in the ***.qvmconf** file, which reduces maintenance of this file.

The at sign (`@`) in front of the filename in the command-line instruction designates a file as the `qvm` configuration file.

Starting a VM without a configuration file

A `qvm` configuration file is *not* required; these files are a convenience, and recommended, especially for complex VMs. However, you can start a `qvm` process instance without a configuration file. Just enter the configuration information in the command line. For example, the following command starts and configures a Linux guest with a PL1011 vdev on an ARM platform:

```
qvm cpu sched 8 ram 0x80000000,128m load /vm/images/linux.img \
    cmdline "console=ttyAMA0 earlycon=pl011,0x1c090000 debug
user_debug=31 loglevel=9" \
    initrd load /vm/images/ramdisk.img \
vdev pl011 loc 0x1c090000 intr gic:37 hostdev /dev/ser2
```

Configuration information for a `qvm` process entered through the command line uses the same syntax as `qvm` configuration files.

NOTE:
The `system` option isn't specified in the command line configuration shown above because this configuration is used for testing a temporary configuration.

Initializing a VM without launching the guest

You can instruct a VM to initialize the guest's environment, which entails validating all the VM's configuration settings, and then exit before launching the guest. For information on doing so, see the `dryrun` option in the "VM Configuration Reference" chapter.

Page updated: August 11, 2025

Booting and Shutting Down

This chapter describes how to boot and shut down the hypervisor host and guests in a QNX hypervisor system.

Page updated: August 11, 2025

Using Virtual Devices, Networking, and Memory Sharing

This chapter describes how guests can discover and connect to vdevs and how they can use important hypervisor capabilities, such as networking and memory sharing.

Page updated: August 11, 2025

Discovering and connecting VIRTIO devices

Multiple methods are available for connecting a device driver in a guest to its corresponding hypervisor host device. QNX Hypervisor uses the VIRTIO standard for implementing virtual devices (vdevs) in the host.

A device driver running in a guest can use one of two methods to connect to a VIRTIO vdev in the host:

- PCI discovery
- Direct memory mapping

The method you use will depend on how your guest is configured, and how easy or difficult you want to make changing the device location. For example, in a production system, direct memory mapping (where you write in your configurations the memory locations of your devices) may be acceptable or even preferable, but you may require a more flexible approach during your project's development phase and therefore may want to use PCI discovery.

NOTE:
The QNX OS *Utilities Reference* has information about drivers such as `devb-ahci` and `devb-virtio` and their options.

PCI discovery

The PCI discovery method is most common in guests designed for x86 platforms, but it can also be used by guests for ARM platforms. PCI hardware isn't required to support this discovery mechanism, because in a hypervisor, guest PCI is completely virtual.

The following steps explain how PCI mapping can be used to enable a driver in a guest to discover and connect to a VIRTIO device. The example uses the `virtio-blk` vdev, but it could also use other VIRTIO vdevs such as `virtio-console` or `virtio-net`.

Using `virtio-blk` and a QNX guest, PCI discovery can be done as follows:

1. In the hypervisor host, start the driver for the AHCI SATA interfaces (`devb-ahci`).
2. Configure the VM hosting the guest to map the location of the AHCI SATA driver (located at **/dev/hd1t178**) through VIRTIO to the guest; for example:

   ```
   vdev virtio-blk
       hostdev /dev/hd1t178
       name virtio-blk_qvm178
   ```

   See vdev virtio-blk for more information about configuring this vdev in the VM.
3. Start the QNX guest.
4. In the guest, run `pci-server` (see `pci-server` in the *Utilities Reference*); for example:

   ```
   pci-server -bus-scan-limit=8
   ```
5. In the QNX guest, use `pci-tool` to query the PCI device (see `pci-tool` in the *Utilities Reference*); for example:

```
[x86 guest QNX 8.0]% pci-tool -v

B000:D00:F00 @ idx 0
        vid/did: 1c05/0002
                BlackBerry QNX, n/a QVM PCI host bridge
        class/subclass/reg: 06/00/00
                Host-to-PCI Bridge Device

B000:D01:F00 @ idx 1
        vid/did: 1af4/1042
                <vendor id - unknown>, <device id - unknown>
        class/subclass/reg: 01/80/00
                Other Mass Storage Controller

B000:D02:F00 @ idx 2
        vid/did: 1af4/1041
                <vendor id - unknown>, <device id - unknown>
        class/subclass/reg: 02/80/00
                Other Network Controller

B000:D03:F00 @ idx 3
        vid/did: 1c05/0001
                BlackBerry QNX, n/a QVM guest shared memory factory
        class/subclass/reg: 05/80/00
                Other Memory Controller

B000:D04:F00 @ idx 4
        vid/did: 1af4/1042
```

Note that the disk controller appears with a VID/DID (vendor ID/device ID) of `1af4/1042`; this is the VIRTIO standard reference for a mass storage device. Other devices such as memory devices or network devices will also show up as PCI devices.

In a Linux guest, you can use `lspci`. You may see the VID/DID as `1c05/0042`; this is the BlackBerry VIRTIO vendor ID. The Linux kernel module understands that this is a block device.

6. In the QNX guest, start the VIRTIO driver for block devices (`devb-virtio`); for example:

   ```
   devb-virtio
   ```

   The driver will scan the PCI space to find the device with the `1af4/1042` vendor ID/device ID and mount it as a block volume; for instance, the above command will create **/dev/hd0** in the guest.

7. If the block volume is already formatted as a QNX6 power-safe filesystem, you can mount it; for example:

   ```
   mount -tqnx6 /dev/hd0 /mydisk
   ```

Direct memory mapping

The following steps explain how a device can be mapped in the host, and the mapping passed on to a driver in a guest. The example uses the `virtio-blk` vdev, but it could also use other VIRTIO vdevs such as `virtio-console` or `virtio-net`.

Using `virtio-blk` and a QNX guest, direct memory mapping can be done as follows:

1. In the hypervisor host, start the driver for the appropriate interfaces (e.g., `devb-ahci`).
2. Configure the VM with the path of the host file to use for the contents of the device (see vdev virtio-blk in the "Virtual Device Reference" chapter).

   Use a location in guest-physical memory that isn't used by any other driver in the guest, and an interrupt that isn't used by any other driver or service in the guest. In this example we use `0x1c0d0000` and `41`:

   ```
   vdev virtio-blk
       loc 0x1c0d0000
       intr gic:41
       hostdev /dev/hd1t178
       name virtio-blk_qvm178
   ```

3. Start the QNX guest. No PCI server is needed.
4. Start the VIRTIO block device driver (`devb-virtio`) in the guest, using the driver's startup options to map in the memory and interrupt you specified in the `virtio-blk` vdev configuration; in our example the options would be as follows:

   ```
   devb-virtio virtio smem=0x1c0d0000,irq=41
   ```

The guest driver should now find the device in the host through the VIRTIO device.

NOTE:
You can use FDTs to expand direct mapping to support dynamic discovery of devices. For example, on an ARM platform you can define an FDT overlay to provide the guest with information about virtual devices.

For more information about using FDTs in a QNX hypervisor system, see “ACPI tables and FDTs” in the “Configuration” chapter.

Page updated: August 11, 2025

qvm-check

Report if the system supports QNX hypervisor operation

Syntax:

```
qvm-check
```

Runs on:

QNX OS

Options:

None

Description:

Run the `qvm-check` utility on a target system to learn if the features needed to run a QNX hypervisor have been enabled. This utility returns one of the following values:

- 0 — the system is set up to run a QNX hypervisor
- 1 — the system is *not* set up to run a QNX hypervisor
- 2 — the results are inconclusive; see the output for more information

When there's a problem (and hence, a non-zero value is returned), `qvm-check` also prints a message describing the problem.

Page updated: August 11, 2025

Utilities and Drivers Reference

This chapter describes the utilities and drivers delivered with the QNX hypervisor.

Page updated: August 11, 2025

Common vdev options

There are common options that apply to more than one type of vdev.

These common options are described below. If, however, an option is also described in the reference for an individual vdev, refer to that description.

`hostvector` ***vector_number***

Applies only to PIC vdevs. This option sets the associated host vector number of the first line of the current vdev. This number is used by *InterruptAttach()* and *InterruptAttachEvent()* (see the QNX OS *C Library Reference*). The PIC input lines that follow are sequentially assigned to host vector numbers.

This option is needed only if any of the PIC guest interrupts will be marked as pass-through. If `hostvector` isn't specified, its value defaults to the previous PIC device's `hostvector` number, plus this previous device's number of lines. If this is the first PIC specified, `hostvector` defaults to 0 (zero).

`intr` ***guest_intr***

The guest interrupt of the current vdev is set to *guest_intr*, in two parts, separated by a colon (e.g., `intr gic:40`).

For PCI devices that use an interrupt, if you don't set a number for *guest_intr*, this number is automatically assigned. For non-PCI devices (e.g., MMIO devices), you must set this option.

For more information, see "Guest interrupts" in the "Configuration" chapter.

`loc` ***location_spec***

Make the current vdev appear to the guest at *location_spec*, which can be in memory, in I/O space, or on the PCI bus, depending on the vdev. The *location_spec* argument is complex, consisting of multiple fields that depend on which of these resource types is being configured. This option is mandatory for non-PCI (e.g., MMIO) devices.

`log` ***filter*[,*filter*,...] *output_dest***

The message types to output and their output destination. This option lets you override the VM's global logging for the current vdev, so you can see certain message types related to the vdev's activity.

The option's argument has two tokens: The first is a comma-separated list of the severity levels of the messages to be output. The possible severity levels are:

- `fatal`
- `internal`
- `error`
- `warning`
- `info`
- `debug`
- `verbose`

The second token is the destination, which is one of: `stdout`, `stderr`, `slog`, and `none`. This last setting allows you to disable outputting certain message types.

You may specify as many `log` options as you like, but if you specify conflicting destinations for a given severity level, the latter (rightmost) option takes precedence. For example, if you have the following:

```
log fatal,internal,error,warn stdout
log warn none
```

The second option setting cancels the logging of warnings that was set by the first setting.

The `log` and `logger` options work similarly, so for more details about the syntax and semantics of this option, see the logger entry in the "VM Configuration Reference" chapter.

`name` ***name***

An optional name used by the vdev for further configuration. Currently this option is relevant only for peer vdevs, such as `virtio-net`, and for PIC devices. For certain PIC devices, the name may be referenced in *guest_intr* specifications to say that the device is connected to a given PIC input line.

See "VM configuration file example" for details on how the name option relates to specific PIC hardware. See `vdev virtio-net` in this chapter for more information about using the `name` option when configuring peers in a network.

vdev 8259

Emulate minimal functionality of an Intel 8259 PIC

Synopsis:

```
vdev 8259 loc baddr
```

Option:

`loc` ***baddr***

The base address of the registers (default space is `io:`).

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description

x86 only. The `8259` vdev emulates minimal functionality of an Intel 8259 programmable interrupt controller (PIC). It can't be used as a virtual PIC device, but its presence is required for guest OSs to be able to boot.

Page updated: August 11, 2025

vdev gic

Emulate an ARM standard Generic Interrupt Controller

Synopsis:

```
vdev gic options
```

Options:

`gicc` ***guest_address***
: Provide the guest-physical address of the virtual GIC's CPU registers.

`gicd` ***guest_address***
: Provide the guest-physical address of the virtual GIC's distributor registers.

`gicr` ***guest_address***
: Provide the guest-physical address of the virtual GIC's redistributor registers.

`host-gicd` ***host_address***
: Deprecated. Use the *host-paddr-gicd* configuration variable, as desribed in the "VM Configuration Reference" chapter.

`host-gich` ***host_address*[/*cpu*/*spacing*]{,*host_address*[/*cpu*/*spacing*]}**
: Deprecated. Use the *host-paddr-gich* configuration variable, as desribed in the "VM Configuration Reference" chapter.

`host-gicr` ***host_address***
: Deprecated. Use the *host-paddr-gicr* configuration variable, as desribed in the "VM Configuration Reference" chapter.

`host-gicv` ***host_address***
: Deprecated. Use the *host-paddr-gicv* configuration variable, as desribed in the "VM Configuration Reference" chapter.

`its` ***address***
: Present the guest with virtualized Interrupt Translation Service (ITS) hardware at the guest-physical address specified by *address*.

  ITS hardware isn't useful without LPIs, so if you specify the `its` option, you must also specify the `num-lpis` option.

`its-prealloc` ***num_dts*[, *num_itts*]**
: Pre-allocate memory for the ITS data structures to avoid allocation at runtime. The first argument specifies the number of device table entries while the second argument, if given, specifies the number of interrupt translation table entries.

`loc` ***address***
: Provide the guest-physical address of the GIC distributor registers.

`msi` ***address*, *base_intr*,*num_intrs***
: Create a Message Signaled Interrupt (MSI) device frame at the physical address specified by *address*.

  The *base_intr* argument specifies the first interrupt number the frame allows, and *num_intrs* specifies the number of interrupts allowed. Thus, the highest interrupt allowed is `base_intr + num_intrs - 1`.

  NOTE:
  When presenting a vdev or passing through a device requiring MSI or MSI-X, you must make an MSI mechanism available to the guest that uses this vdev or device.

  You don't need PCI to use the MSI feature; it is supported for non-PCI devices on an ARM host that has ITS hardware.

`num-lpis` ***number***
: Set the number of Locality-specific Peripheral Interrupts (LPIs) that will be made available to the guest. The *number* argument must be a value from 8192 to 65536.

For a list of options available to all vdevs, see "Common vdev options" earlier in this chapter.

Description:

ARM only. Implicit (see "Implicit vdevs").

The `gic` vdev emulates an ARM standard Generic Interrupt Controller. This release supports only version 3 (GICv3). Because this is an implicit vdev, you need to add configuration information for it only if you want to use non-default behavior or provide information to be used in case information is not available from the board or is incorrect.

If none of `gicc`, `gicd`, or `loc` is specified, the `qvm` process assumes the ARM foundation model addresses. For GICv3 these addresses are:

```
gicc=0x2c000000, gicd=0x2f000000, gicr=0x2f100000
```

If some addresses are specified but others missing, the `qvm` process places the `gicc` registers after the `gicr` registers. For more information, see "*asinfo*" in the "System Page" chapter in *Building Embedded Systems*.

Page updated: August 11, 2025

vdev hpet

Provide the guest with an HPET device

Synopsis:

```
vdev hpet options
```

Options:

`frequency` *value*

Use the given frequency instead of the default 14318180 Hz.

Currently, the vdev uses an integer divisor to match with the hypervisor's internal counter, which will introduce a skew.

`intr` *intr_def*[,*intr_def*[,*intr_def*[,...]]]

Define timers and assign interrupt lines to them. For complex reasons and to support legacy operations, a Linux guest will expect the first timer to be on IRQ 0, and a QNX guest will expect it to be on IRQ 2. The second timer should be set to IRQ 8 in both cases, and any additional timer can use any other interrupt.

If you specify multiple interrupts (timers), you can:

- Specify the `intr` option once, followed by multiple definitions; for example:

  ```
  vdev hpet intr ioapic:2,ioapic:6
  ```

- Specify a new `intr` option for each definition; for example:

  ```
  vdev hpet intr ioapic:2 intr ioapic:6
  ```

- Mix the two forms; for example:

  ```
  vdev hpet intr ioapic:2,ioapic:6 intr ioapic:10
  ```

`legacy-free`

Clear the "legacy" bit (LEG_RT_CA) in the main ID register. Specifically, clearing or not clearing the bit determines where Timer0 interrupts may be routed.

If you specify this option, the guest can't make any assumptions about the routing of Timer0 and Timer1.

If you don't specify this option, the guest can assume that Timer0 will be routed to IRQ2 and Timer1 to IRQ8. Note the following, however:

- Not specifying the `legacy-free` option doesn't enforce the assumed routing (Timer0 to IRQ2, Timer1 to IRQ8). You must ensure that your configuration specifies the correct routing.
- A Linux guest will always assume that Timer0 is routed to IRQ0, because this is usually the case. Though this routing warrants a redirection entry in the Multiple APIC Description Table from ACPI, Linux ignores the absence of this redirection entry.
- The above means that if you don't specify the `legacy-free` option, the first two `intr` instances in the `hpet` vdev configuration should be as follows, for QNX:

  ```
  vdev hpet intr ioapic_name:2,ioapic_name:8
  ```

  and for Linux:

  ```
  vdev hpet intr ioapic_name:0,ioapic_name:8
  ```

`loc`

The base address of the HPET registers.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

x86 only. The `hpet` vdev provides a High-Precision Event Timer (HPET) device to the guest. It is exposed to the guest through an ACPI device. If this device is provided in the VM (`qvm` process instance), the guest may use HPET services. For example, for a QNX guest, the `startup-apic` command in a boot image will search for and use the virtualized HPET timer if it exists.

Currently, only the first HPET vdev is exposed, and there is no way for the guest to discover the address of any additional HPET vdev.

This vdev supports up to 32 timers. The guest can't change the IRQ assignment of each timer, but Message Signaled Interupts (MSIs) are supported.

Page updated: August 11, 2025

vdev ioapic

Emulate an Intel ioapic PIC

Synopsis:

```
vdev ioapic options
```

Options:

`hostvector` ***vector***

Associate the first guest interrupt of the I/O APIC with the host interrupt number specified by *vector*; associate the second guest interrupt with *vector* +1, etc.

If a guest interrupt is marked as a pass-through target and no host vector is explicitly provided, this association provides the host interrupt vector number to use for the guest interrupt.

`intr apic`

Downstream interrupt to raise.

`loc` ***addr***

Base address of the device registers.

`name` ***name***

Name to use for upstream interrupts referencing this PIC.

For a list of options available to all vdevs, see “Common vdev options” at the beginning of this chapter.

Description:

x86 only. The `ioapic` vdev emulates an Intel I/O APIC programmable interrupt controller (PIC).

Page updated: August 11, 2025

vdev mc146818

Emulate the Motorola 146818 real-time clock chip

Synopsis:

```
vdev mc1416818 loc baddr reg regnum,value
```

Options:

`loc`
: The base address of the device registers (default space is `io:`).

`reg` *`regnum,value`*
: Set the NVRAM *reg_num* register to *value*.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

x86 only. The `mc146818` vdev emulates the Motorola 146818 real-time clock chip.

Page updated: August 11, 2025

vdev pci-dummy

Provide a read-only, inactive PCI device

Synopsis:

```
vdev pci-dummy options
```

Options:

`clone` ***host_pci_dev***

Use the specified PCI device as a template for the vdev, and copy the values of the named registers from the host into what the guest will see.

`reg` ***offset/size=value***

Set a specific configuration space register. For example:

```
vdev pci-dummy reg 0/4=0xFFFF
```

sets the configuration space register at offset 0 and size 4 (bytes) to `0xFFFF`.

Permitted sizes are 1, 2, 4 and 8 bytes.

`reg` ***name=value***

Equivalent to the `reg` *offset*/*size* option above, but uses one of the known names (e.g., Vendor_ID). The size is implied by the name, so you don't have to specify it.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

x86 only. The `pci-dummy` vdev provides a read-only, inactive PCI device. Such a device may be needed to expose a specific pair of PCI identifiers to help the guest identify a hardware platform.

The following registers are considered: Vendor_ID, Device_ID, Revision_ID, Class_Code[0], Class_Code[1], Class_Code[2], Header_Type, Sub_Vendor_ID, Sub_System_ID.

Page updated: August 11, 2025

vdev pci-host-bridge

Create a PCI host bridge device

Synopsis:

```
vdev pci-host-bridge [ecam addr]
```

Option:

ecam ***addr***

Set the location of the PCI ECAM space to *addr*. If this option isn't specified, the `qvm` process instance will automatically choose a location, and inform the guest via ACPI or FDT.

For a list of options available to all vdevs, see “Common vdev options” at the beginning of this chapter.

Description:

Implicit. The `pci-host-bridge` vdev creates a PCI host bridge device.

Page updated: August 11, 2025

vdev pckeyboard

Emulate an IBM PC keyboard controller

Synopsis:

```
vdev pckeyboard loc baddr
```

Option:

`loc`

The base address of the device registers (default space is `io:`).

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

x86 only. The `pckeyboard` vdev provides minimal emulation of an IBM PC keyboard controller.

This emulation is provided because for historical reasons; x86 systems expect the device to be present. The vdev provides the minimum emulation needed for a QNX guest to reboot in a hypervisor VM.

Page updated: August 11, 2025

vdev pl011

Emulate a PL011 serial device

Synopsis:

```
vdev pl011 options
```

Options:

batch ***timeout***

Instead of writing one character at a time, batch the output until the number of microseconds specifed by *timeout* has passed with no new characters coming in.

delayed ***seconds*** **|forever**

Delay opening the host device until it is first referenced by the guest.

If the device isn't found in the host, keep trying to find the device for the number of seconds given by *seconds* before timing out (e.g., `delayed 5`). If *seconds* is set to `0` (i.e., `delayed 0`), try only once to find the device. If it is set to "forever" (i.e., `delayed forever`), never time out and just keep trying to find the device.

hostdev [<|>]*host_device_name*

hostdev >"|*command_string*"

The first form means use the *host_device_name* as the source and/or destination of characters for the vdev.

For a given `hostdev` option, you can use one of `<` (input) or `>` (output) so the host device specified by *host_device_name* provides input or receives output.

To specify separate input and output devices, you can use multiple `hostdev` options:

```
hostdev <myhostinputdevice
hostdev >myhostoutputdevice
```

Or, you can use neither `<` nor `>` to indicate the same device is used for both input and output:

```
hostdev myhostdevice
```

The second form means redirect the vdev's output to a process created by executing *command_string*. The command string must be immediately preceded by the pipe character (`|`) and then surrounded by double quotes, as shown here:

```
hostdev >"|/guests/tee /guests/pipe_test.log"
```

Here the vdev's output is sent as input to the `/guests/tee` program which takes the pathname of a log file, **/guests/pipe_test.log**, as an argument. The command you name can do whatever you want; for example, it could perform logging with file size limits. For further explanation, refer to "Logging output" in the `vdev virtio-console` reference.

name ***device_path***

Specify a device path that you can write commands to in order to control the size and number of log files created for storing the vdev's output. For example, you can issue a command to reopen a log file (i.e., truncate it to zero bytes) after you noticed it was getting too big. For information about performing this log rotation, refer to "Logging output" in the `vdev virtio-console` reference.

sched ***priority***

Send pulses indicating available input on the host device with a priority level of *priority*.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

ARM only. The `pl011` vdev emulates a PL011 serial device.

vdev progress

Report when a guest accesses an address in guest-physical memory, or attempts to execute specific instructions

Synopsis:

```
vdev progress options
```

Options:

`cpuid` ***instr***
: x86 only. Generate a report whenever the guest executes a `CPUID` instruction with the EAX equal to the value specifed by *instr*.

`full`
: Generate a full register dump on each reference.

`loc` ***addr***
: Report when the guest reads or writes the guest-physical address specified by *addr*.

`smc` ***instr***
: ARM only. Generate a report whenever the guest executes an `SMC` instruction with the immediate value specified by *instr*.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

The `progress` vdev can generate reports when any of the `loc`, `smc`, or `cpuid` conditions described above occurs.

For example, the following generates a report when a guest accesses the specified location in its guest-physical memory:

```
vdev progress
        loc 0xffff0000
```

Similarly, the following generates a report when a guest executes an `SMC` instruction with a value of `0x01`:

```
vdev progress
        smc 0x01
```

The hypervisor outputs reports to whatever destination is specified by the `logger` option's `info` filter (see `logger` in the "VM Configuration Reference" chapter).

Page updated: August 11, 2025

Virtual Device Reference

This chapter presents the virtual devices (vdevs) delivered with QNX hypervisors, and describes how to configure these vdevs.

NOTE:

- For general information about configuring VMs which include vdevs, including syntax, see "Assembling and configuring VMs" in the "Configuration" chapter.
- For information about options available to all vdevs, see the next section, "Common vdev options".
- For information about options available to specific vdevs, see the remaining sections in this chapter.

Implicit vdevs

An *implicit vdev* is a vdev that is in the `qvm` process code, so it is present in the VM assembled by that process even if you don't specify it. You need to specify such a vdev only if you want to use non-default values for its options. Implicit vdevs are marked as such in the individual vdev descriptions in this chapter.

Syntax

All vdevs specified in a VM configuration are preceded by the `vdev` option, which identifies the component as a virtual device. Its mandatory *name* argument specifies the device type:

```
vdev name
```

For example, the following creates a vdev that emulates an Intel 8254 timer chip:

```
vdev timer8254
    intr myioapic:0
```

NOTE:

The `qvm` configuration parsing and validation allows colons (":") in the *name* argument, because they may be used with some vdevs. However, this character is not supported for Programmable Interrupt Controller (PIC) vdevs (e.g., the `gic` or `ioapic` vdev). If you use a colon in a PIC device name, you will get an error when you attempt to refer to it in the configuration for another vdev (e.g., in an `intr` option).

Multiple listings of a vdev

If you specify the same vdev more than once in a VM configuration, the shared object (**vdev-*.so**) file that implements the vdev is still loaded by the associated `qvm` process instance only once. In this case, then, the shared object must manage as many distinct device instances as the guest will need to access for its purposes, just as you would have to connect multiple hardware devices of the same type to support multiple use cases by an OS in a non-virtualized system.

To manage multiple device instances, the vdev must arrange its data to avoid interference between the instances. For information on doing so, refer to the `vdev_s` and `vdev_factory` entries in the *Virtual Device Developer's API Reference*.

Page updated: August 11, 2025

vdev ser8250

Emulate the Intel 8250 UART

Synopsis:

```
vdev ser8250 options
```

Options:

`batch` *timeout*

Instead of writing one character at a time, batch the output until the number of microseconds specifed by *timeout* has passed with no new characters coming in.

`delayed` *seconds* | `forever`

Delay opening the host device until it is first referenced by the guest.

If the device isn't found in the host, keep trying to find the device for the number of seconds given by *seconds* before timing out (e.g., `delayed 5`). If *seconds* is set to `0` (i.e., `delayed 0`), try only once to find the device. If it is set to "forever" (i.e., `delayed forever`), never time out and just keep trying to find the device.

`hostdev [<|>]`*host_device_name*
`hostdev >"|`*command_string*"

The first form means use the *host_device_name* as the source and/or destination of characters for the vdev.

For a given `hostdev` option, you can use one of `<` (input) or `>` (output) so the host device specified by *host_device_name* provides input or receives output.

To specify separate input and output devices, you can use multiple `hostdev` options:

```
hostdev <myhostinputdevice
hostdev >myhostoutputdevice
```

Or, you can use neither `<` nor `>` to indicate the same device is used for both input and output:

```
hostdev myhostdevice
```

The second form means redirect the vdev's output to a process created by executing *command_string*. The command string must be immediately preceded by the pipe character (`|`) and then surrounded by double quotes, as shown here:

```
hostdev >"|/guests/tee /guests/pipe_test.log"
```

Here the vdev's output is sent as input to the `/guests/tee` program which takes the pathname of a log file, **/guests/pipe_test.log**, as an argument. The command you name can do whatever you want; for example, it could perform logging with file size limits. For further explanation, refer to "Logging output" in the `vdev virtio-console` reference.

`intr` *intr*

Signal *intr* for device interrupts.

`loc` *addr*

Set the base address of the device registers to *addr*. Default space is `io:`.

`name` *device_path*

Specify a device path that you can write commands to in order to control the size and number of log files created for storing the vdev's output. For example, you can issue a command to reopen a log file (i.e., truncate it to zero bytes) after you noticed it was getting too big. For information about performing this log rotation, refer to "Logging output" in the `vdev virtio-console` reference.

`sched` *priority*

Set the priority specified by *priority* for pulses indicating that input is available.

For a list of options available to all vdevs, see "Common vdev options" earlier in this chapter.

Description:

The `ser8250` vdev emulates the Intel 8250 UART.

vdev shmem

Provide an inter-guest system shared memory device

Synopsis:

```
vdev shmem options
```

Options:

`allow` *fnpattern*

Allow the guest to create or attach to any shared memory region whose name matches the specified filename pattern (*fnpattern*).

In this string argument, you can use the same wildcards as the shell uses for filename expansion, including asterisks (`*`). You can repeat this option as much as necessary to grant the guest access to shared memory.

If you use this option at all, the configuration will include an implicit `deny *` at the end of the restrictions list, and the guest will be denied access to all shared memory regions not specified by this list. For example:

```
vdev shmem
    allow test*
    allow pluto
```

grants the guest access to any shared memory region whose name begins with `test` and to the region called `pluto`. The guest is denied access to all other regions.

These semantics mean you need to use one (but not both) of the `allow` or `deny` options. For further discussion, see "Restrictions list" below.

`create` *name*,*size*

Pre-create a shared memory object named *name* of *size* bytes.

You can repeat the `create` option to pre-create multiple shared memory regions.

`deny` *fnpattern*

Deny the guest access to any shared memory region whose name matches the specified filename pattern (*fnpattern*).

In this string argument, you can use the same wildcards as the shell uses for filename expansion, including asterisks (`*`). You can repeat this option as much as necessary to restrict the guest's access to shared memory.

If you use this option at all, the configuration will include an implicit `allow *` at the end of the restrictions list, and the guest will be granted access to all shared memory regions not specified by this list. For example:

```
vdev shmem
    deny live*
    deny pluto
```

denies the guest access to any shared memory region whose name begins with `live` and to the region called `pluto`. The guest can access any other shared memory region, however.

These semantics mean you need to use one (but not both) of the `allow` or `deny` options. For further discussion, see "Restrictions list" below.

Alternatively, after using `create` directives to pre-create specific named shared memory regions, you can use a `deny *` directive to deny access to all other regions.

`intr` *intr*

If the `loc` option is specified, you must also specify the value of *intr*; this sets the guest interrupt that is generated when another guest sends a notification of an update to shared memory.

`loc` [*addr*]

The base address of the factory page for the vdev. If *addr* isn't specified, the vdev appears as a PCI device, and the `qvm` process automatically assigns the factory page location and adds this location to the PCI BAR register.

If you specify this option, you must also specify the `intr` option (see above).

`sched` *priority*

Use the priority specified by *priority* for pulses indicating that input is available.

`subst` [*match*],*prefix*

Specify a prefix to add to shared memory region names (see "Using the `subst` option" below).

match

The character string in the shared memory region name to substitute, if found, by the character string specified by *prefix*.

prefix

The character string to substitute for the character string specified by *match*, or to simply prefix to the shared region name if *match* isn't specified.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

ARM and x86. The `shmem` vdev allows you to share memory between guests. Normally this device appears as a PCI device, but if the `loc` and `intr` options are specified, the guest running in the associated VM will see this device as a memory-mapped I/O (MMIO) device at the specified location.

CAUTION:
If a guest uses the `shmem` vdev without considering how much memory the host has available, any memory allocation by the guest will fail when its memory usage exceeds the host's RLIMIT_AS setting specified for the underlying `qvm` process. For more information, see "Memory" in the "Understanding Virtual Environments" chapter.

Restrictions list

You can use the `allow` and `deny` options to define lists of shared memory regions that the guest may or may not access. By limiting access to shared memory and, hence, allocations of shared memory, these options help prevent denial-of-service attacks.

You can repeat either of these options multiple times to specify whatever filename patterns you need to manage the guest's access to shared memory regions. When specifying filename patterns, you can include wildcard characters such as the asterisk (`*`) to indicate multiple shared memory regions for convenience. We recommend defining either an allow or a deny policy for every named region used by the VM, including when using wildcard naming.

NOTE:
If you use the command line to enter `vdev shmem` configuration information that includes `allow` or `deny` options with wildcard entries, you must use the "\" escape character before the asterisk to get the instruction through the shell (e.g., `vdev shmem allow test\*`).

For more information about using the shared memory vdev, see "Memory sharing" in the "Using a QNX Hypervisor System" chapter.

Using the `subst` option

You can use the `subst` option to present the host with different names for shared memory regions for each guest, while presenting the same name to each guest so you can use the same executables in the different guests. For example, each guest might see a shared memory region called `foomem`, but the host would see these regions as `guest0mem`, `guest1mem`, etc.

The following VM configurations produce the result presented above by substituting `guest*` for `foo`:

VM 0

```
vdev shmem
    subst foo,guest0
    create foomem,0xf0000
    ...
```

VM 1

```
vdev shmem
    subst foo,guest1
    ...
```

An equivalent result can be achieved by simply adding the `guest*` prefix, as follows:

VM 0

```
vdev shmem
    subst ,guest0
    create mem,0xf0000
    ...
```

VM 1

```
vdev shmem
    subst ,guest1
    ...
```

NOTE:

- Only one instance of the `subst` option may be specified per shmem vdev.
- If you want the `subst` option to affect the name specified by a `create` option, you must place the `subst` before the `create`.
- The length of the memory region name (name + prefix) must not exceed the value of GUEST_SHM_MAX_NAME (see "Virtual registers (guest_shm.h)" in the "Using Virtual Devices, Networking, and Memory Sharing" chapter).

Page updated: August 11, 2025

vdev smmu

*Provide the required subset of **smmu-*.so** library functionality needed by a `smmuman` service running in a QNX guest*

Synopsis:

```
vdev smmu options
```

Options:

`intr` *num*

The interrupt number to use to inform the guest that the vdev has completed the task requested by the guest.

If you don't specify this option, the `qvm` process automatically assigns an interrupt number.

`loc` *baddr*

The base address of the request page for this device.

If you don't specify this option, the `qvm` process automatically assigns a base address.

For a list of options available to all vdevs, see “Common vdev options” earlier in this chapter.

Description:

The `smmu` vdev provides the following services:

IOMMU/SMMU services

The vdev provides for the guest in a hypervisor VM the same services for guest-physical memory as an IOMMU/SMMU provides for an OS running on hardware. These services are: accepting configuration information passed in by the guest's `smmuman` service, denying DMA device attempts to access memory outside the permitted regions, communicating these denials back to devices, recording these attempts, etc.

Memory mapping

The vdev shares memory mappings for DMA devices passed through to the guest with the `smmuman` service running in the hypervisor host. This ensures that the host `smmuman` service and the board IOMMU/SMMUs can manage access to host-physical memory by pass-through DMA devices.

CAUTION:
On ARM platforms, to run the `smmuman` service, a hypervisor guest must load **libfdt.so**, so make sure you include this shared object in the guest's buildfile. Note, though, that the **libfdt** library is certified for internal use only. You must use this library only where directed by QNX and only in context of that direction.

For more information about the `smmuman` service and how to use it, see the *SMMUMAN User's Guide* in the QNX OS documentation.

Page updated: August 11, 2025

vdev timer8254

Emulate the Intel 8254 timer chip

Synopsis:

```
vdev timer8254 options
```

Options:

`frequency` ***input_freq_in_hz***

Set the input frequency to the device, in Hertz.

`intr` ***intr***

Signal *intr* when a timer expires.

`loc` ***addr***

Set the base address of the device registers to *addr* (default space is `io:`).

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

x86 only. The `timer8254` vdev emulates the Intel 8254 timer chip.

Page updated: August 11, 2025

vdev virtio-blk

Provide the standard VIRTIO interface for block devices

Synopsis:

```
vdev virtio-blk options
```

Options:

`delayed` *`seconds`* `|forever`

Delay opening the host device until it is first referenced by the guest.

If the device isn't found in the host, keep trying to find the device for the number of seconds specified by *seconds* before timing out (e.g., `delayed 5`). If *seconds* is set to `0` (i.e., `delayed 0`), try only once to find the device. If the argument is set to "forever" (i.e., `delayed forever`), never time out and just keep trying to find the device.

`dio disable|enable|read`

Configure the use of Direct I/O when processing the guest's requests. The default setting is `disable`, meaning Direct I/O is not used by default. The `enable` setting activates it for both read and write requests. The `read` setting activates it only for read requests, while write requests are processed with an _IO_WRITE message. With this last setting, which is known as Mixed Mode I/O, the back-end `io-blk` block cache gets used as a writeback cache.

If a Direct I/O access fails at any point (either because it's not supported by the back-end driver or for any other reason), the device will permanently switch back to using _IO_READ and _IO_WRITE messages.

`hostdev [<]`*`host_filename`*

Use the specified host file to store the contents of the device.

A `<` symbol in front of *host_filename* indicates that device contents will be enforced as read-only (e.g., `hostdev </dev/hd0` will make `/dev/hd0` read-only to the guest). With no such symbol, the default read/write access is granted to the guest.

These semantics mean the `hostdev` option for `vdev virtio-blk` is different than the option with the same name for other vdevs. The `<` symbol doesn't specify the input source and the `>` symbol to specify output isn't supported.

`intr` *`intr`*

Signal *intr* for device interrupts. Not required when the vdev appears as a PCI device.

`legacy`

Provide the VIRTIO legacy interface (0.9.5) rather than the 1.0 or later standard versions.

`loc` *`addr`*

Set the base address of the device registers to *addr*. Not required if the vdev appears as a PCI device.

`threads` *`number`*

Set the number of threads for processing requests. This number must be at most 32, and will be truncated to the maximum that the back-end device actually supports if the device has a thread limit smaller than 32.

By default, only one thread is created. Having multiple threads process requests issued by the guest can improve performance for the device. Also, the `threads` option setting affects the number of virtqueues that get created. Having multiple virtqueues can also boost performance. For more information, refer to "Multiqueue support" below.

NOTE:

For better performance, we recommend using a number of threads that is based on the configuration of your hypervisor host system. If the system is single-core, use just one thread. If it's quad-core, use between two and four threads for optimal performance.

For the system to actually benefit from having multiple threads in the VIRTIO device, the driver in the guest must allow requests to be processed asynchronously, and the back-end driver and the back-end device in the host must not serialize the requests.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

ARM and x86. The `virtio-blk` vdev provides an interface for block devices (e.g., SATA drives). Normally this device appears as a PCI device, but if you specify the `loc` and `intr` options, the guest will see it as a memory-mapped I/O device at the specified location.

For `virtio-blk`, packed virtqueues and multiqueue are enabled if the guest supports them. Information about this second feature is given just below.

CAUTION:
Under no circumstances should a block device containing a filesystem be shared between two entities (host or guests), even if one side is read-only. If such sharing is allowed, the two entities cannot synchronize and are unaware of each other's caches, resulting in unpredictable behavior.

Multiqueue support

If the `virtio-blk` vdev is given a `threads` option setting greater than one (1), during startup, the vdev advertises its support for the *multiqueue* feature, which allows a guest to control the mapping of the request-processing threads to virtqueues. If the guest driver then requests the use of multiqueue, the number of virtqueues created is the lesser of the `threads` option setting and the number of vCPUs created in the VM (which is the number of cpu options specified in the configuration file). Creating multiple virtqueues can improve performance when the guest has a heavy I/O load.

If the guest driver does not request multiqueue usage, only one virtqueue is created and all the threads operate on this virtqueue. This setup can still provide better performance that a vdev with just one thread but might not achieve the same performance as when multiple virtqueues are used.

Page updated: August 11, 2025

vdev virtio-console

Provide the standard VIRTIO interface for console devices

Synopsis:

```
vdev virtio-console options
```

Options:

`delayed` *seconds* | `forever`

Delay opening the host device until it is first referenced by the guest.

If the device isn't found in the host, keep trying to find the device for the number of seconds given by *seconds* before timing out (e.g., `delayed 5`). If *seconds* is set to `0` (i.e., `delayed 0`), try only once to find the device. If it is set to "forever" (i.e., `delayed forever`), never time out and just keep trying to find the device.

`hostdev [<|>]`*host_device_name*

`hostdev >"|`*command_string*`"`

The first form means use the *host_device_name* as the source and/or destination of characters for the vdev.

For a given `hostdev` option, you can use one of < (input) or > (output) so the host device specified by *host_device_name* provides input or receives output.

To specify separate input and output devices, you can use multiple `hostdev` options:

```
hostdev <myhostinputdevice
hostdev >myhostoutputdevice
```

Or, you can use neither < nor > to indicate the same device is used for both input and output:

```
hostdev myhostdevice
```

The second form means redirect the vdev's output to a process created by executing *command_string*. The command string must be immediately preceded by the pipe character (|) and then surrounded by double quotes, as shown here:

```
hostdev >"|/guests/tee /guests/pipe_test.log"
```

Here the vdev's output is sent as input to the `/guests/tee` program which takes the pathname of a log file, **/guests/pipe_test.log**, as an argument. The command you name can do whatever you want; for example, it could perform logging with file size limits. For further explanation, refer to "Logging output" below.

`intr` *intr*

Signal *intr* for device interrupts. Not required when the vdev appears as a PCI device.

`legacy`

Provide the VIRTIO legacy interface (0.9.5) rather than the 1.0 or later standard versions.

`loc` *addr*

Set the base address of the device registers to *addr*. Not required if the vdev appears as a PCI device.

`name` *device_path*

Specify a device path that you can write commands to in order to control the size and number of log files created for storing the vdev's output. For example, you can issue a command to reopen a log file (i.e., truncate it to zero bytes) after you noticed it was getting too big. For information about performing this log rotation, refer to "Logging output" below.

`sched` *priority*

Use the priority specified by *priority* for pulses indicating that input is available on the host device.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

ARM and x86. The `virtio-console` vdev provides an interface for console devices. Normally the device appears as a PCI device, but if you specify the `loc` and `intr` options, the guest will see it as a memory-mapped I/O device at the specified location.

This vdev is useful on systems where console interaction to a guest is slow over a regular console or serial terminal. The vdev is a buffering driver; most often it's used when a guest is interacting with the host over an emulated UART. For example, many ARM boards use the PL011 serial port as a terminal for the Linux kernel's output. Using this vdev to buffer PL011 data reduces emulation overhead.

The `virtio-console` vdev requires:

- A device to run on. For example, the PL011 serial device is slow and has slow output because it interacts a lot with the virtualized serial port. By using the `virtio-console` vdev on top of the PL011 device, the console interaction is very fast both on input and output due to the vdev's larger and more efficient buffering.
- A virtual driver running in the guest. Linux, Android, and QNX guests need a VIRTIO driver to be running. For instance, you should configure a Linux guest to include the `virtio-console` kernel module, and specify `console=hvc0` in the `cmdline` option in the `qvm` configuration for the VM.

The vdev offers the following features:

- VIRTIO_CONSOLE_F_MULTIPORT — currently, this is limited to a single port, and the name of the port is the one given to the vdev instance with the `name` option.
- VIRTIO_CONSOLE_F_EMERG_WRITE — the driver can use an emergency write to output a single character without initializing virtqueues or completing feature negotiation. This is handy for providing debugging output early during bootup and for reporting fatal errors (e.g., VIRTIO ring corruption).

Logging output

You can write the vdev output to log files and manage their size and number. There are two ways to do this:

- **Redirect the output to a separate process that manages log files**

  You can define the `hostdev` option to send the vdev output to a separate process. The syntax for doing so is shown in the option's description above. This interface doesn't assume any specific functionality and thus gives you maximum flexibility in generating and managing log files. For example, the other process can:

  - implement a log file rotation scheme (e.g., the Linux utility `logrotate` could be ported to QNX OS and named in the `hostdev` command)
  - generate a single log file and truncate it when it exceeds a certain size
  - generate log files and then compress and/or encrypt them as desired

- **Configure the vdev to offer manual user control of log files**

  You can define the `hostdev` option to specify a log file, and the `name` option to specify the path for receiving commands that manage that file. Suppose you have the following configuration:

  ```
  vdev virtio-console
      hostdev >/guests/virtconsole.log
      name virt_console_name
  ```

  The vdev output will then be stored in the log file **/guests/virtconsole.log** and the vdev can then react to commands sent to a device path that's based on the `qvm` system's name and that ends with **virt_console_name**.

  The user can manually control log file rotation using the following method:

  1. Move the current log file to a new name (e.g., `mv current-logfile saved-logfile`).

     The file continues to accumulate log output even though it has the new name. Note that the new file must be in the same filesystem as the current one.

  2. Send the `0` command to the appropriate device path:

     ```
     echo 0 > /dev/qvm/qnx-guest-1/virt_console_name
     ```

     The `qvm` process closes the original log file and opens a new one with the original name. At this point `qvm` is using the new file for logging and the user can do what they want with the saved one.

     The access permissions for the device path given in *name* are defined using a security policy. For more information, refer to the "Security Policies" chapter of the *System Security Guide*.

  This design avoids race conditions between the vdev process that writes to the file and another process that truncates the file when it gets too big.

Page updated: August 11, 2025

vdev virtio-entropy

Provide a guest with a source of random numbers (entropy)

Synopsis:

```
vdev virtio-entropy options
```

Options:

`delayed` ***num*** **`|forever`**

Delay opening **/dev/random** until the guest's first request.

If **/dev/random** isn't available, either wait for up to the number of seconds specified by *num* before timing out or, if `forever` is specified, never time out.

`intr` ***intr***

Signal *intr* for device interrupts. Not required when the vdev appears as a PCI device.

`loc` ***addr***

Set the base address of the device registers to *addr*. If this option isn't specified, the vdev appears as a PCI device.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

ARM and x86. The `virtio-entropy` vdev provides a guest with access to a source for random numbers (entropy) at **/dev/random**.

For a Linux guest, you must enable the guest's `virtio-rng` Linux kernel module.

Page updated: August 11, 2025

vdev virtio-net

Provide the standard VIRTIO interface for network devices

Synopsis:

```
vdev virtio-net options
```

Options:

`intr` *intr*
: Signal *intr* for device interrupts. Not required when the vdev appears as a PCI device.

`legacy`
: Provide the VIRTIO legacy interface (0.9.5) rather than the 1.0 or later standard versions.

`loc` *addr*
: Set the base address of the device registers to *addr*. If this option isn't specified, the vdev appears as a PCI device.

`mac` *mac_address*
: Use the *mac_address* as the Ethernet address for the vdev in the guest.

`name` *name*
: Set this side of a peer-to-peer networking connection to *name*.

`peer` *path*
: Directly connect to the peer device *path* (usually of the form **/dev/qvm/*vm*/*vnet***). In this case, specifying *mac* and *name* is highly recommended.

  For the *name*, use the name (*system_name*) of the system with the peered device (see system in the "VM Configuration Reference" chapter).

`peerfeats` *mask*
: Specify VIRTIO Network feature bits that should be assumed to be supported by the peer. Standard negotiation isn't possible, so feature bits must be used, as explained in "Configuring the features of a peer connection" below.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

The `virtio-net` vdev provides an interface for network devices. Normally this device appears as a PCI device, but if you specify the `loc` and `intr` options, the guest will see it as a memory-mapped I/O device at the specified location.

For `virtio-net`, packed virtqueues are enabled if the guest supports them.

Configuring access permissions

The access permissions for the node used for a peer-to-peer connection are controlled using a security policy. For more information, refer to the "Security Policies" chapter of the *System Security Guide*. There's no `virtio-net` configuration option for setting access permissions, unlike in previous releases.

Configuring the features of a peer connection

Feature bits are used to specify the features exposed to a peer in a network. They can be defined for each individual `virtio-net` interface (i.e., each peer connection can have its own feature set).

The `peerfeats` option defines the maximum set of features that will be negotiated between the two TCP stacks connected as peers. The available feature bits are specified in *Virtual I/O Device (VIRTIO) Version 1.1*, Section 5.1.3 (see the document repository at oasis-open.org). This maximum set of bits is currently set at `0x7fc3`. The current QNX OS TCP stack supports a subset of the maximum set. This subset is `0x19c3`, or:

```
0001 1001 1100 0011
```

with the least-significant bit (bit zero) on the right.

Of particular interest are the bits that support checksumming and Transmit Segmentation Offload (TSO); both these features are supported in the QNX TCP stack (see the table below).

The default `peerfeats` value for a peer interface is `0x0000`.

You should limit the feature bits to the features supported by the guests. For example, if you are configuring two QNX OS guests to connect as peers, you should specify `0x19c3` as the value for the `peerfeats` option's *mask* in the VM configuration for each guest.

When using the `peerfeats` option:

- If you know the features supported by both the near-end and the far-end peer interfaces, specify them with the `peerfeats` option in the VM configurations for each guest.
- If you don't know the features supported by both peers, you can set the `peerfeats` *mask* to `0x7fc3` and let the connected TCP stacks negotiate the features they will use (check your OS documentation for details).
- If you are configuring a peer connection to a QNX `io-sock` bridge, you *must* set the `peerfeats` option to `0x0` (zero) to disable all features on both peer nodes.

  If one of the peers is in the hypervisor host, configure its interface to disable checksums for TCP and UDP, as follows:

  ```
  ifconfig vp0 -tcp4csum -udp4csum -udp6csum -tcp6csum
  ```

  This is required so that the host bridge does not incorrectly discard or delay packets. Note, however, that you disable checksums only for receiving IP (`-rx`), so you can still send packets with checksums. If the peer node is properly configured, it should have checksums disabled and be able to receive your packets.

NOTE:
If you enable TSO in the interface, you must also use `ifconfig` after the interface starts running to turn the TSO support on. For example, the `ifconfig` startup in your guest might look something like this:

```
ifconfig vt1 tcp4csum udp4csum tcp6csum udp6csum tso4 tso6
```

Guest-to-guest and guest-to-host

The table below lists the recommended feature set for guest-to-guest and guest-to host connections:

Bit	Name	Feature
0	VIRTIO_NET_F_CSUM	Device handles packets with partial (or no) checksum.
1	VIRTIO_NET_F_GUEST_CSUM	Driver handles packets with partial (or no) checksum.

Guest-to-guest only

The table below lists the recommended feature set for guest-to-guest connections only:

Bit	Name	Feature
7	VIRTIO_NET_F_GUEST_TSO4	Driver can receive TSOv4.
8	VIRTIO_NET_F_GUEST_TSO6	Driver can receive TSOv6.
11	VIRTIO_NET_F_HOST_TSO4	Device can receive TSOv4.
12	VIRTIO_NET_F_HOST_TSO6	Device can receive TSOv6.

For more information about setting up peer connections, see "Networking" in the "Using Virtual Devices, Networking, and Memory Sharing" chapter.

Page updated: August 11, 2025

vdev wdt-ib700

Emulate the watchdog found on the IB700 board

Synopsis:

```
vdev wdt-ib700 options
```

Options:

`action [`*dump*`,]`*terminate*`|`*nmi*

Specify what to do when the watchdog timer triggers:

- *dump* – generate a guest dump, if the global option `dump` has been specified for the guest (see "dump" in this chapter, and "Getting a guest dump during a crash" in the "Monitoring and Troubleshooting" chapter)
- *nmi* – send an edge on the interrupt line to which the device is connected
- *terminate* – terminate the guest

`intr` ***intr***

Specify the interrupt number on which to send an interrupt if the watchdog is configured to generate an interrupt.

`loc` ***addr***

The base address of the registers in guest-physical memory (default space is `io:`). The value specified for *addr* must be the same as the value specified for the guest `wdtkick`'s `-a` option.

If this option isn't specified, the configuration assumes the default address: `0x441`.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

x86 only. Provide a watchdog for guests running on x86 platforms.

For more information about watchdogs in a QNX hypervisor system, see "Watchdogs" in the "QNX Hypervisor: Protection Features" chapter.

Page updated: August 11, 2025

vdev wdt-sp805

Emulate an SP805 watchdog

Synopsis:

```
vdev wdt-sp805 options
```

Options:

`action [`*dump*`,]`*terminate*`|`*fiq*

Specify what to do when the watchdog timer triggers:

- *dump* – generate a guest dump, if the global option `dump` has been specified for the guest (see dump in this chapter, and "Getting a guest dump during a crash" in the "Monitoring and Troubleshooting" chapter)
- *fiq* – send an edge on the interrupt line to which the device is connected
- *terminate* – terminate the guest

`frequency` ***hz***

The clock-cycle frequency that corresponds to the *effective watchdog frequency* as described in the SP805 specifications. This value is used to calculate the watchdog timeout. It is specified here in hertz (Hz).

If this option isn't specified, the configuration assumes the default: `25000000` (25 MHz).

`intr` ***intr***

Specify the interrupt number on which to send an interrupt if the watchdog is configured to generate an interrupt.

`loc` ***addr***

The base address of the registers in guest-physical memory. This value must be the same as the value specified for the guest `wdtkick`'s `-a` option.

If this option isn't specified, the configuration assumes the default address: `0x1C0F0000`.

For a list of options available to all vdevs, see "Common vdev options" at the beginning of this chapter.

Description:

ARM only. Provide a watchdog for guests running on ARM platforms.

Emulating the hardware SP805 watchdog device's behavior, this vdev counts down twice before it triggers a SIGQUIT to terminate the hosting `qvm` process instance. Thus, if your watchdog period is set to three seconds, the effective watchdog clock period (when the SIGQUIT will be triggered) is six seconds.

NOTE:
If the watchdog is configured to generate an interrupt, it is up to the guest to configure the interrupt controller to send the appropriate request to a vCPU.

For more information about how an SP805 watchdog calculates the effective watchdog clock period, see the SP805 specifications.

For more information about watchdogs in a QNX hypervisor system, see "Watchdogs" in the "QNX Hypervisor: Protection Features" chapter.

Page updated: August 11, 2025

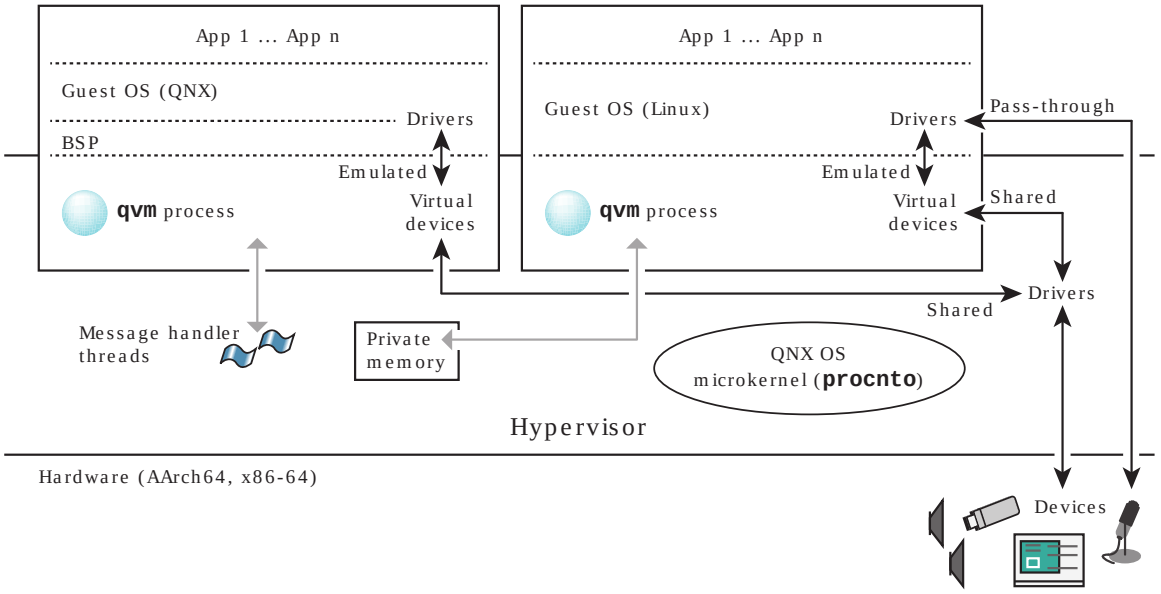
Architecture

A QNX hypervisor comprises the QNX OS microkernel, which contains built-in support for virtualization, and one or more `qvm` process instances.

Two representations of a QNX hypervisor system

The figure below presents a high-level view of the QNX hypervisor architecture and some of the configurations available for accessing both virtual and physical devices. Not all possible guest-device configurations are shown; those shown illustrate only some of the possible configurations.

Figure 1An overview of the hypervisor, showing VMs and some of the ways guests can access virtual and physical devices.



The diagram above presents a static view of a hypervisor system. Unfortunately, it is necessarily misleading, in that it can be interpreted as suggesting that guests actually run *in* VMs or *on* the host system; to make matters worse, when describing a hypervisor system, we often say that a guest is running in a VM.

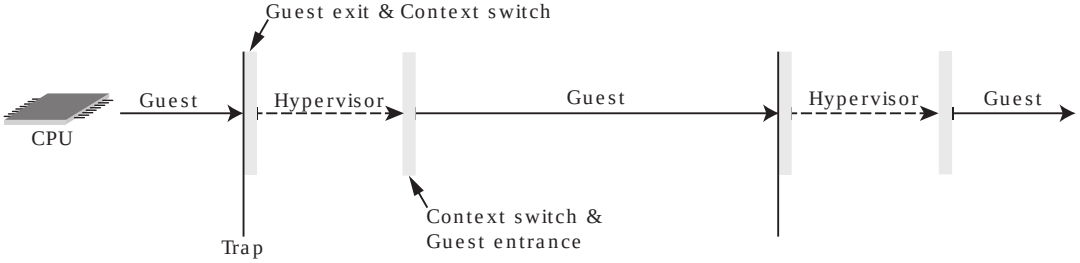
In fact, a guest doesn't actually run in a VM. The hypervisor isn't an intermediary that translates the guest's instructions for the CPU. The VM defines virtual hardware (see "Virtual devices") and presents it and pass-through hardware (see "Pass-through devices") to the guest, which doesn't need to know it is running "in" a VM rather than in an environment defined directly by the hardware.

That is, when a guest is running, its instructions execute on a physical CPU, just as if the guest were running without a hypervisor. Only when the guest attempts to execute an instruction that it is not permitted to execute or when it accesses guest memory that the hypervisor is monitoring does the virtualization hardware trap the attempt and force the guest to exit.

On the trap, the hardware notifies the hypervisor, which saves the guest's context (this is called a *guest exit*) and completes the task the guest had begun but was unable to complete for itself. On completion of the task, the hypervisor restores the guest's context and hands execution back to the guest (this called a *guest entrance*).

The Lahav Line below presents a more dynamic view of the interaction between the hypervisor and one of its guests. For simplicity, it assumes an execution path on a single CPU.

Figure 2A Lahav Line showing how execution in a QNX hypervisor system alternates between the hypervisor and its guests. On a trap, the hypervisor manages the guest exit, saving the guest's context, then restoring it before the guest entrance.



For information about what the hypervisor does to manage the time flow in the guest (caused by the need to have the guest exit), see “Time” in this chapter.

Page updated: August 11, 2025

Devices

A QNX hypervisor provides guests with access to physical devices, including pass-through and shared devices, and virtual devices, including emulation and para-virtualized devices.

About device access

When you configure your QNX virtualized environment (the hypervisor, the `qvm` processes for the virtual machines, and the guests), you need to assign physical devices and virtual devices (vdevs) to the hypervisor and to the guests. To do this properly you need to know not just if a device is a physical or virtual device, but also the type of physical or virtual device, because this determines:

- if the guest or the hypervisor must include a device driver
- if the `qvm` hosting a guest must include the relevant vdev
- if the guest needs to know that it is running in a virtualized environment

NOTE:
In a non-virtualized system, the device driver in an OS must match the hardware device on the physical board. In a virtualized system, the device driver in the guest must match the vdev.

For example, suppose you are using a `pl011` vdev configured as follows: `vdev pl011 loc 0x1c090000 intr gic:37`. You must then tell your guest to use a PL011 device at location `0x1c090000` and interrupt `37`.

You can't pass instructions to your guest to use a UART device, say `earlycon=msm_hsl_uart,0x75b0000`, and expect it to find the PL011 device any more than you could do that in a non-virtualized environment.

For information about configuring the hypervisor host, `qvm` processes, and devices, see the "Configuration" chapter.

Page updated: August 11, 2025

Memory

In a QNX virtualized environment, the guest-physical memory that a guest sees as contiguous physical memory may in fact be discontiguous host-physical memory assembled by the virtualization.

A guest in a QNX virtualized environment uses memory for:

- normal operation (see "Memory in a virtualized environment")
- accessing pass-through devices (see "Memory mapping for pass-through devices")
- sharing information with other guests ("Shared memory")

Note the following about memory in a QNX hypervisor:

- With the exception of shared memory, memory allocated to a VM is for the exclusive use of the guest hosted by the VM; that is, the address space for each guest is exclusive and independent of the address space of any other guest in the hypervisor system.
- If there isn't enough free memory on the system to complete the configured memory allocation for a VM, the hypervisor won't complete the configuration and won't start the VM.
- If the memory allocated to its hosting VM is insufficient for a guest, the guest can't start, no matter how much memory is available on the board.
- You can limit the total memory usage of the VM by defining RLIMIT_AS for the `qvm` process. Defining this limit helps prevent runaway memory usage by a guest, whether it's shared memory or memory exclusive to that guest, including pass-through memory.

  The launcher program `on` provides a way to do this. Refer to the `on` documentation in the QNX OS *Utilities Reference* for details. For instance, a guest that's given 1G of RAM in its configuration file and has modest pass-through memory can be given a limit of 2G as follows:

  ```
  on -L 6:2000000000:2000000000 qvm @myguest.qvmconf
  ```

  You can then verify this memory usage limit with `pidin`:

  ```
  pidin -F "%N %#" -P qvm
  ```
- With the exception of memory used for pass-through devices to prevent information leakage, the hypervisor zeroes memory before it allocates it to a VM. Depending on the amount of memory assigned to the guest, this may take several seconds.

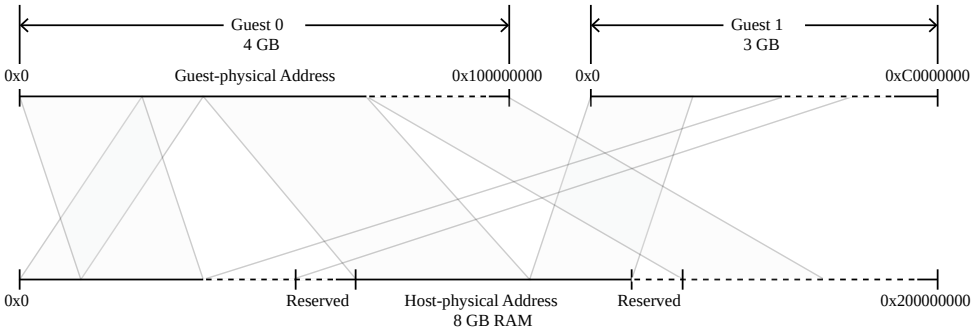
Memory in a virtualized environment

In a QNX virtualized environment, a guest configured with 1 GB of RAM will see 1 GB available to it, just as it would see the RAM available to it if it were running in a non-virtualized environment. This memory allocation appears to the guest as physical memory, but it is in fact memory assembled by the virtualization configuration from discontiguous physical memory. ARM calls this assembled memory *intermediate physical memory*; Intel calls it *guest physical memory*. For simplicity we will use *guest-physical memory*, regardless of the platform (see "Guest-physical memory " in the "Terminology" section).

When you are configuring and accessing memory in a QNX virtualized environment, it is important to remember the following:

- The total amount of memory allocated to guests and any other use may not exceed the physical memory available on the board.
- Memory allocations in the hypervisor host must be in multiples of the QNX OS system page size (4 KB). However, memory smaller than a page can be passed through to the guest. An example with source code for a vdev that passes through small amounts of memory is provided in the QNX Hypervisor GitLab Repository at https://gitlab.com/qnx/hypervisor.
- Guest-physical memory appears to the guest OS just like physical memory would appear to it in a non-virtualized system; for example, on x86 systems it would have the same gaps for legacy devices.
- The apparently physical addresses a guest sees are in fact addresses in the guest-physical memory, which is assembled by the `qvm` process when it creates a VM.
- There is no correspondence between the address a guest sees (guest-physical address) and the physical address to which the guest-physical address translates below the virtualization layer (host-physical address).

Figure 1Illustration of how the memory a guest OS sees as contiguous physical memory is assembled by the virtualization configuration from unrelated regions of physical memory.



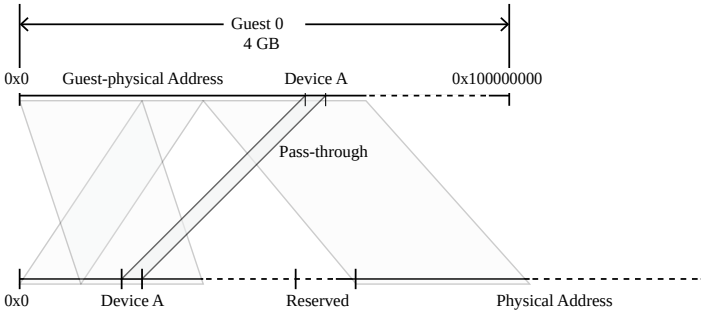
The diagram above presents a simplified illustration of how in a QNX virtualized system the guest memory allocations are assembled from discontiguous chunks of physical memory. To simplify the diagram, the memory allocation for Guest 1 is incomplete, and in the interests of legibility a gap has been added between the guests. Note also that some regions of physical memory may be reserved (e.g., for devices the board architecture requires at specific locations) and can't be allocated to a guest.

When you configure memory for guests, you need to configure the size of the memory allocation, and any platform-specifics, such as gaps for legacy devices (x86) or the RAM start address (ARM). The hypervisor looks after assembling blocks of physical memory into allocations of guest-physical memory for each guest.

Memory mapping for pass-through devices

A guest accessing physical devices as *pass-through* devices needs to map them into the memory regions configured to be accessible to that guest. Note that there is no correspondence between the physical address *seen by the guest* and the physical address *seen from the hypervisor host domain*.

Figure 2Illustration of how the guest-physical address for a pass-through device doesn't correspond to the host-physical address of the device.



For example, Device A may be configured at `0x100` in Guest 0's physical memory. In the hypervisor host, this device may be configured as a pass-through device for Guest 0 at this same location (`0x100`), so when Guest 0 needs to access the device, it looks at `0x100`. However, remember that the guest-physical address that a guests sees translates into some other address in host-physical memory, say `0xC00000100`.

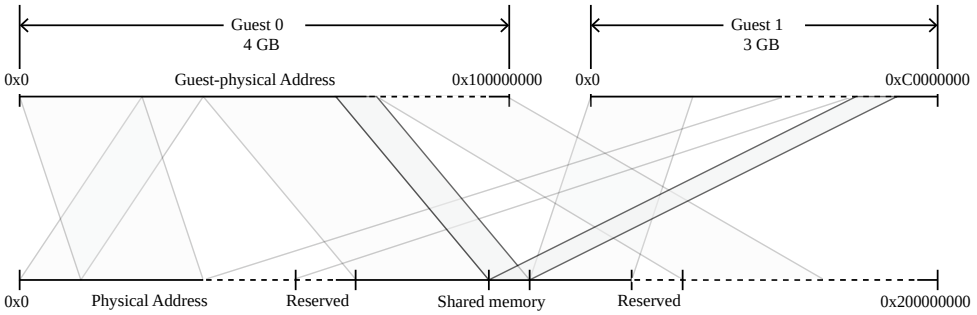
For more information about pass-through devices, see "Pass-through devices" in this chapter.

Shared memory

Portions of physical memory can be allocated to be shared between guests. Guests will use a virtual device such as `vdev shmem` to attach to the same physical address (PA) and use the shared memory area to share data, triggering each other whenever new data is available or has been read.

NOTE:
The PA is actually the host-physical address, not the guest-physical address. The guest-physical addresses will probably differ between the guests.

Figure 3Illustration of memory shared between two guests. Each guest sees the memory as its own.



For information about how to implement memory sharing, see "Memory sharing" in the "Using Virtual Devices, Networking, and Memory Sharing" chapter.

Configuring memory

To allocate memory for a virtual machine (VM), you must allocate the memory required by each part of the system. You don't allocate all of the memory for the VM in one chunk. Rather, you allocate the memory required for the image, then the memory required for your various devices, etc.

Many systems have reserved areas of memory. This is particularly true of x86 systems, which require many vestigial devices at specific locations. These rules apply to guests running in virtual environments, because the OSs are expecting the vestigial devices to be present.

When you allocate memory, it may be efficient to specify only the location of the bootable image in guest-physical memory, and let the `qvm` process pick the location for devices, shared memory, etc.

If a RAM location is to be used for the guest's bootable image, the guest must be configured to look for the image at this address inside its memory. In addition, if you use the `qvm` configuration `load` component to load your bootable image, the address where you load the image must match your guest's configuration. (If you don't specify the address, the `qvm` process looks after this for you.)

For example, for a QNX guest, you might do the following:

- Allocate RAM for the guest image with a start address in guest-physical memory of `0x80000000`:

  ```
  ram 0x80000000,128M
  ```

- Load the bootable image to this location:

  ```
  load 0x80000000,/vm/images/qnx8.ifs
  ```

- Specify this location in the guest's buildfile:

  ```
  [image=0x80000000]
  [virtual=aarch64le,elf] .bootstrap = {
     [+keeplinked] startup-armv8_fm -v -H
     [+keeplinked] PATH=/proc/boot procnto-smp-instr -v
  }
  ```

See "ram" in the "VM Configuration Reference" chapter.

DMA device containment

The hypervisor can use an IOMMU/SMMU Manager such as `smmuman` to ensure that no DMA pass-through device is able to access host-physical memory to which it has not been explicitly granted access.

For more information about the `smmuman` service and how to use it, see "DMA device containment" in the "QNX Hypervisor: Protection Features" chapter, and the *SMMUMAN User's Guide*. For information about using another component to manage IOMMU/SMMU services, see the *smmu-externally-managed* configuration variable description.

Page updated: August 11, 2025

Physical devices

A physical device may be for the exclusive use of the hypervisor host or of a guest, or it may be shared.

Physical devices (or simply *devices*) in a virtualized environment are exactly the same as devices in a non-virtualized environment. They require drivers, and they assert interrupts and receive signals.

Guests running in a QNX virtualized environment may access a physical device directly or through a virtual device, or they may be prohibited from accessing a device. Access to physical devices is controlled by the configuration of the virtualized environment. A physical device may be configured to be:

- exclusively available to a specific guest (see "Pass-through devices")
- shared between guests; not all devices can be shared (see "Shared devices")

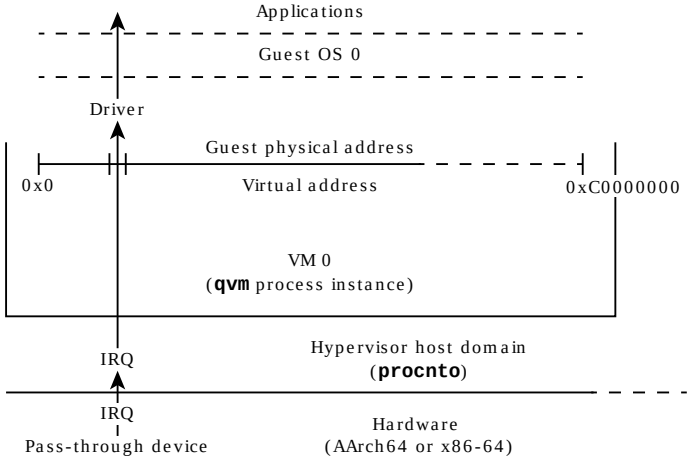
Pass-through devices

In a hypervisor system, a pass-through device is a physical device to which a guest has direct and exclusive access. This type of access to a physical device may be faster than access through a virtual device, or than when access is shared with other guests or the hypervisor host.

To use a pass-through device, the guest must have its own driver for the physical device. No `vdev` is required in the `qvm` process hosting the guest, and no driver is required in the hypervisor itself.

With a pass-through device, the hypervisor host domain knows only that it must route interrupts from the physical device directly to the guest, and pass signals from the guest directly to the device. All interaction is between the guest and the device; the hypervisor's only responsibility is to identify and allow to pass through the interrupts from the device and the signals from the guest.

Figure 1Illustration of a pass-through device in a hypervisor system.



Before the guest is launched, pass-through devices must be configured in the ***.qvmconf** file for the VM that will host the guest (see "pass" in the "VM Configuration Reference" chapter).

CAUTION:
In general, only one resident of a virtualized system is permitted access to a pass-through device at any one time. If the host owns a device that a guest requires as a pass-through device, the host must terminate its device driver before the guest can start a driver for the device in its virtual environment.

Similarly, if one guest owns a device as a pass-through device, it must terminate the device driver in its virtualized space before another guest can use the device in its space.

In short, you should never pass a DMA device through to more that one guest, and only in exceptional designs should you pass a non-DMA device through to more than one guest. If you believe that your design requires that a non-DMA device be passed through to more than one guest, contact your QNX representative.

Passing through clock-dependent devices

Clock-dependent devices (e.g., eMMC) may require additional work to be passed through, because the guest doesn't have access to the clocks. Strategies for passing through these devices include:

- Modify the guest's device driver so that it doesn't attempt to modify the clocks.

- Create a vdev that intervenes when the guest's device attempts to manipulate the clocks.
- If the clock registers are appropriately spaced, pass the appropriate subset of these registers to the guest.

For more information, contact your QNX representative.

Passing through complex devices

Some devices (e.g., audio and video devices) consist of multiple hardware interfaces, and require very specific configurations in order to operate correctly.

A knowledgeable system designer who knows and understands how to use all the device interfaces, and can define them properly in the VM configuration, should be able to make any complex device work as a pass-through device in a QNX hypervisor system.

Please contact QNX services for more information about the sharing of complex devices between guests.

Shared devices

In a hypervisor system, shared devices are physical devices that can be used by more than one guest, or by one or more guests and the hypervisor itself. There are two methods and models available for device sharing; these include:

- Referred sharing
- Mediated sharing

The hypervisor provides services such as TCP/IP, shared memory, and *virtqueue* support, which you can use for device sharing. The services and configurations you implement depend on the requirements of your system.

NOTE:
QNX hypervisors support vdevs that enable the sharing of audio and graphics physical devices. For more information, please contact your QNX representative.

Referred sharing

With the referred sharing model, the physical device that is to be shared is assigned to a guest, which has full control of the device. If the physical device is accessed as a pass-through device, the guest needs a driver for the physical device (see "Pass-through devices" above). If the physical device is accessed through a virtual device, the `qvm` process hosting the guest must have the appropriate virtual device (see "Virtual devices" in this chapter), and both the guest and the hypervisor host need the appropriate device drivers: for the virtualized device (guest), and for the actual physical device (host).

The other guests that must access the physical device use mechanisms such as TCP/IP, or shared memory and virtqueues to communicate with the guest that controls the device, and pass data as needed.

For example, suppose that two guests are running on a hypervisor system. Guest 0 is a QNX safety-related guest, while Guest 1 is a garden-variety Linux OS guest with no safety requirements. They both need to access a device to output audio: Guest 0 to emit warning sounds, Guest 1 to play music.

We assign the audio device as a pass-through device to Guest 0. The audio virtual device in the `qvm` process for the VM hosting Guest 1 is designed to share data with a virtual device in another VM, and it is configured to know that this other virtual device is in the `qvm` process for the VM hosting Guest 0. With this configuration, all audio device activity for Guest 1 is *referred* to Guest 0 for processing.

When the virtual device in Guest 0's `qvm` process receives interrupts and data for Guest 1, it passes them on to the virtual device in Guest 1's `qvm` process; when this second virtual device receives signals and data from Guest 1 for the audio device, it passes these on to Guest 0 to pass on to its audio device driver and, ultimately, to the physical audio device.

Since Guest 0 controls all input and output from the audio device, it can set priorities to ensure that safety-related processes and threads take precedence over input and output to the garden-variety OS and its applications. Guest 0 can even refuse to cooperate with the other guest or with a device if cooperation might put its own safety-related activities in jeopardy.

CAUTION:
A guest that uses a physical device controlled by another guest depends on that other guest for access to the device; if for any reason the guest controlling the device becomes unavailable, the device also

becomes unavailable. Thus, if a guest is shutting down, it should inform any other guest using any device it controls that the device is now unavailable.

Mediated sharing

The mediated sharing model is similar to the referred sharing model. The difference is that with the mediated model, it is hypervisor host processes rather than a guest that assume the responsibility for controlling the interface with the physical device and for ensuring that configured priorities are respected.

With this model, the device that is to be shared is assigned to the hypervisor itself. Virtual devices in the `qvm` process for each VM connect to a mediator in the hypervisor. This mediator determines which interrupts and data are passed on from the physical device to guests, and which signals and data are passed on from guests to a device driver in the hypervisor and, ultimately, to the physical device.

The mediator also ensures that configured priorities are respected (e.g., ensuring that safety-related activities take precedence), even refusing to cooperate with a guest or device whose activities might be harmful to other guests on the system or the hypervisor itself.

NOTE:
The mediator processes don't have to reside in the hypervisor host. They can reside in one of the guests. With this design, however, a guest failure causes undefined behavior for the mediated sharing.

Page updated: August 11, 2025

Virtual machines

A running hypervisor comprises a QNX OS microkernel and one or more instances of the QNX virtual machine process (`qvm`).

What is a virtual machine?

In a QNX hypervisor environment, a virtual machine (VM) is implemented in a `qvm` process instance. This process is an OS process that runs in the hypervisor host, outside the kernel. Each instance has an identifier marking it so that the kernel knows that it is a `qvm` process.

If you remember anything about VMs, remember that from the point of view of a guest OS, the VM hosting that guest is *hardware*. This means that, just as an OS running on a physical board expects certain hardware characteristics (architecture, board-specifics, memory and CPUs, devices, etc.), an OS running in a VM expects those characteristics. In other words: *the VM in which a guest will run must match the guest's expectations*.

When you configure a VM, you are *assembling* a hardware platform. The difference is that instead of assembling physical memory cards, CPUs, etc., you specify the virtual components of your machine, which a `qvm` process will create and configure according to your specifications.

The rules about where things appear are the same as for a real board:

- Don't install two things that try to respond to the same physical address.
- The environment your VM configuration assembles must be one that the software you will run (the guest OS) is prepared to deal with.

The hardware analogy works in the other direction as well. A VM doesn't need to know what its guest is doing any more than hardware needs to know what an OS is doing. In fact, the VM can't know what a guest is doing ("the guest is a blob"; see the "Terminology" appendix).

For example, a VM can't know why a guest quit (i.e., if the guest quit because a user requested a shutdown or because it encountered a fatal error). If you need to know why a guest quits, you need to rely on the guest to tell you why. If the guest doesn't already have a mechanism to do so, you need to add an appropriate mechanism. For QNX guests, the "Shutdown" screen might provide the required functionality, depending on your requirements.

For more information about configuring VMs, see "Assembling and configuring VMs" in the "Configuration" chapter.

`qvm` services

Each `qvm` process instance provides key hypervisor services.

VM assembly and configuration

In order to create the virtual environment in which a guest OS can run, a `qvm` process instance does the following when it starts:

- Reads, parses, and validates VM configuration (***.qvmconf**) files and configuration information input via the process command line at startup, exiting if the configuration is invalid and printing a meaningful error message to a log.
- Sets up intermediate stage tables (ARM: Stage 2 page tables, x86: Extended Page Tables (EPT)).
- Creates (assembles) and configures its VM, including:
  - allocates RAM (r/w) and ROM (r only) to guests
  - provisions a thread for every virtual CPU (vCPU) it presents to the guest
  - provisions pass-through devices to make them available to the guest
  - defines and configures virtual devices (vdevs) for the hosted guest

VM operation

During VM operation, a `qvm` process instance does the following:

- Traps outbound and inbound guest access attempts and determines what to do with them (i.e., if the address is to a vdev, invoke the vdev code (guest exit, then guest entrance when the vdev code completes); if the address is indeed out of bounds, treat it as such).
- Saves its guest's context before relinquishing a physical CPU.
- Restores its guest's context before putting the guest back into execution.
- Looks after any fault handling.

- Performs any maintenance activities required to ensure the integrity of the VM.

Guest startup and shutdown

A guest in a VM can start just like it does on hardware. From the perspective of the guest, it begins execution on a physical CPU. However, this CPU is in fact a `qvm` vCPU thread. The guest can enable its interrupts, just as it would if it were running in a non-virtualized system.

When the first vCPU thread in a guest's VM begins executing, the VM can know that the guest has booted.

Initiating guest shutdown is the responsibility of the guest. The `qvm` process detects shutdowns initiated through commonly used methods such as PSCI or ACPI. If you want to use a method that the `qvm` doesn't automatically recognize, you can write a vdev that detects the shutdown action and responds appropriately.

See the *Virtual Device Developer's Guide* in the QNX Hypervisor GitLab Repository at https://gitlab.com/qnx/hypervisor for more information about writing vdevs.

NOTE:
If the hypervisor detects an undefined condition in a VM, the hypervisor terminates that VM's `qvm` process instance, which terminates the guest (see "Design Safe States" in the "QNX Hypervisor: Protection Features" chapter).

Manage guest contexts

In a virtualized environment, it is the responsibility of the CPU virtualization extensions to recognize from a guest's actions when the guest needs to exit. When a CPU triggers a guest exit, however, it is the `qvm` process instance (i.e., the VM) hosting the guest that saves the guest's context. The `qvm` process instance completes the action initiated by the guest, then restores the guest's context before allowing it to re-enter (see "Guest exits" in the "Performance Tuning" chapter).

Manage privilege levels

The `qvm` process instance manages privilege levels at guest entrances and exits, to ensure that the guest can run and that the system is protected from errant code.

On a guest entrance, the `qvm` process instance asks the CPU to give the guest the privilege levels it needs to run, but no more. On a guest exit, the `qvm` process instance asks the CPU to return to the privilege levels it requires to run in the hypervisor host.

Only the CPU hardware can change privilege levels. The `qvm` process performs the operations required to have the CPU change privilege levels. This mechanism (hence the operations) is architecture-specific.

Guest access to virtual and physical resources

Each `qvm` process instance manages its hosted guest's access to both virtual and physical resources. When a guest attempts to access an address in its guest-physical memory, this access can be one of the following, and the `qvm` process hosting the guest checks the access attempt and responds as described:

Permitted
The guest is attempting to access a memory region that it owns.

The `qvm` process instance doesn't do anything.

Pass-through device
The guest is attempting to access memory assigned to a physical device, and the guest's VM is configured to know that the guest has direct access to this device.

The `qvm` process instance doesn't do anything. The guest communicates directly with the device.

Virtual device
The guest is attempting to access an address that is assigned to a virtual device (either emulation or para-virtual).

The `qvm` process instance requests the appropriate privilege level changes, and passes execution onto its code for the requested device. For example, a guest CPUID request triggers the `qvm` process to emulate the hardware and respond to the guest exactly as the hardware would in a non-virtualized system.

Fault
The guest is attempting to access memory for which it does not have permission.

The `qvm` process instance returns an appropriate error to the guest.

The above are for access attempts that go through the CPU. DMA access control is managed by the SMMU manager (see “DMA device containment”).

Page updated: August 11, 2025

Interrupts

In a QNX hypervisor system, guests configure their virtual Progammable Interrupt Controllers (PICs), and the hypervisor manages interrupts in accordance with these configurations.

The hypervisor host must intervene to manage any interrupts asserted by the hardware, regardless of whose action triggered the interrupt or who owns the device asserting the interrupt, or of the interrupt's ultimate destination. This means that a guest exit is required so that the hypervisor can look after identifying the interrupt's destination, do any required processing, and pass the interrupt along to its destination.

For information about how to reduce the overhead of handling interrupts in a hypervisor system, see the "Performance Tuning" chapter.

Interrupts for the hypervisor host

Interrupts destined only for the hypervisor host include interrupts from devices owned by the hypervisor, and IPIs between physical CPUs. These interrupts are handled by the QNX OS microkernel interrupt-handling mechanisms, just like interrupts in a non-virtualized QNX OS (see "Interrupts" in *Getting Started with the QNX OS*). There is nothing you need to do with these interrupts that is specific to a hypervisor system.

Interrupts for guests

Handling of interrupts destined for guests is shared between the microkernel (`procnto`) and the VMs (`qvm` process instances) hosting the guests.

The microkernel does the following steps:

- Save the context of the interrupted guest
- Identify the interrupt (using PIC-specific callouts)
- Mask the interrupt (using PIC-specific callouts)
- Inform the relevant `qvm` process instance that the interrupt is available for passing on to the guest

The relevant `qvm` process instance does the following steps:

- Deliver a virtual interrupt to the guest (may use PIC-specific hardware assist routines if these are available)
- Detect that the guest has handled the interrupt
- Unmask the interrupt (see *InterruptUnmask()* in the QNX OS *C Library Reference*)

NOTE:
The `qvm` process doesn't unmask interrupts until the guest has looked after them. This design ensures that if a guest makes an error when handling an interrupt, the error can't affect the hypervisor itself or other guests. In short, the hypervisor is protected from an interrupt storm caused by a guest error.

However, the hypervisor can't protect a guest from itself. If a guest doesn't clear an interrupt condition properly, or if the interrupt rate is too high, the guest will spend its time in its own interrupt-handling code and will not be able to run user threads on any vCPU. This behavior is exactly what would occur if the guest were running in a non-virtualized environment.

Interrupt priorities and destinations

Guest OSs configure their own interrupt priorities and destinations, just as they would if they were running in a non-virtualized environment. The hypervisor presents a virtual GIC (ARM) or APIC (x86) in the VM hosting the guest (see vdev gic and vdev ioapic in the "Virtual Device Reference" chapter).

To the guest these virtual interrupt controllers are indistinguishable from hardware GICs or APICs. The hypervisor delivers interrupts to the vCPUs according to the priorities and the destinations configured by the guest. For example, a QNX guest expects interrupts to go to physical CPU 0, so the hypervisor delivers interrupts to vCPU 0 in the VM hosting the guest.

Interrupts during startup and shutdown

The `qvm` process instance hosting a guest masks interrupts destined for that guest until the guest has successfully booted and has started the appropriate device drivers. Similarly, when a guest begins its shutdown procedure, the hosting `qvm` process instance masks interrupts intended for that guest.

Page updated: August 11, 2025

Scheduling

It is important to understand how scheduling affects system behavior in a QNX virtualized environment.

Hypervisor thread priorities and guest thread priorities

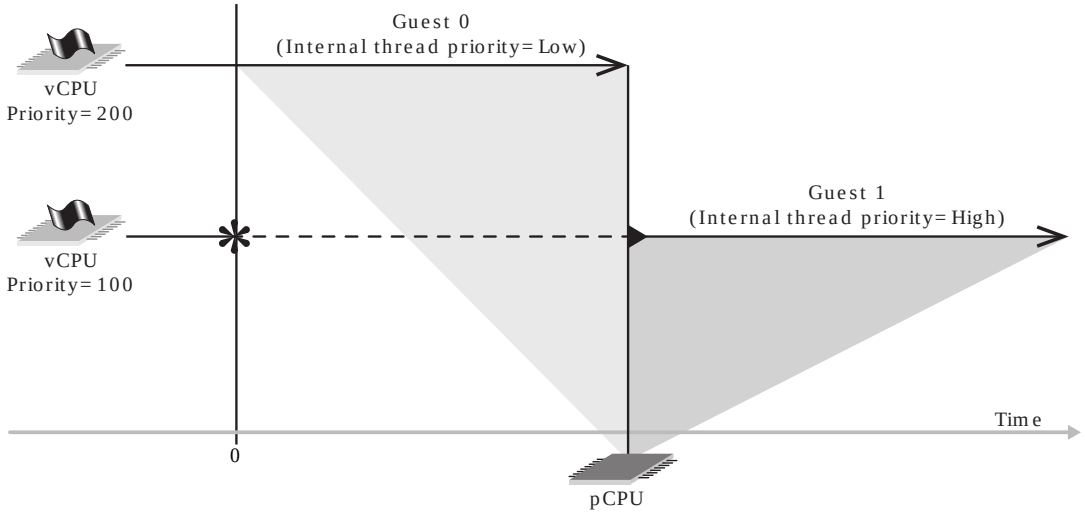
As with any advanced software system, to properly configure scheduling priorities in a QNX hypervisor system requires a thorough understanding of your system's requirements and its capabilities.

To begin with, you should keep in mind the following:

- The hypervisor host has no knowledge of what is running in its VMs, or how guests schedule their own internal software. When you set priorities in a guest OS, these priorities are known only to that guest.
- Virtual CPUs (vCPUs) are scheduled by `qvm` vCPU scheduling threads; these threads exist in the hypervisor host domain.
- Thread priorities in guests have no relation to thread priorities in the hypervisor host. The relative priorities of the `qvm` process vCPU scheduling threads determine which vCPU gets access to the physical CPU (pCPU).

The figure below illustrates that priorities inside guests are not relevant in the host. The shaded areas show which one of two competing vCPU threads gets access to the pCPU.

Figure 1The significant priority for access to a pCPU is the priority of the `qvm` process's vCPU thread.



If two vCPUs compete for a single pCPU, the vCPU thread with the higher priority gets access to the pCPU immediately. The priority of the thread running inside a guest has no effect on this access. In the example shown above:

- The thread *in* Guest 1 has a higher priority than the thread *in* Guest 0.
- The hypervisor thread providing the vCPU for Guest 0 has a higher priority (200) than the hypervisor thread providing the vCPU for Guest 1 (100).
- Therefore, the internal Guest 0 thread gets access to the pCPU and runs from time 0 until it voluntarily gives up the pCPU.
- The result is that the internal Guest 1 thread is blocked until the thread in Guest 0 voluntarily blocks.

Guest exits

The following situations cause a guest to exit:

- a halt in the guest
- a vdev access; this sort of exit doesn't necessarily cause the guest to give up control of the pCPU, because the guest's vCPU thread doesn't necessarily block
- an interrupt on the host, including an interprocessor interrupt (IPI)
- a virtual timer, such as a vdev emulating an Intel 8254 chip
- an instruction, such as a CPUID instruction

For more information about guest exits and their effect on hypervisor system performance, see the "Performance Tuning" chapter. For more information about scheduling in the QNX OS, see "Thread scheduling" in the QNX OS *System Architecture* guide.

Page updated: August 11, 2025

Sharing resources

In a hypervisor system, with the exception of small areas of shared memory, memory and devices are exclusively owned by a single entity (guest or hypervisor host).

Multiple guests running in different VMs may use the same system resource, such as a physical CPU. Management of these resources is entirely the domain of the hypervisor. The system integrator's responsibilities are limited to the (very important) question of configuring the appropriate number of vCPUs for each VM, setting their priorities, and binding (pinning) vCPU threads to specific physical CPUs, if required (see "Multiprocessing" in the "Performance Tuning" chapter).

Other resources, such as memory and physical devices, are assigned to specific VMs by their configurations, which the hypervisor uses to assemble the VMs (see the "Configuration" chapter). These resources become the exclusive property of the guest running in each VM. They can no more be shared between guests than can physical devices on two separate boards.

Sharing memory

When you configure RAM and ROM for your VM, letting the `qvm` process choose the host-physical address for the memory is usually the best strategy, because the process will not allocate the same memory location to more than one VM.

A small shared memory region can be an efficient mechanism for passing data between guests in different VMs, however. If you need to configure a shared memory region, you can do so by using a vdev such as `vdev shmem` to take advantage of the hypervisor's shared memory services. For more information about how shared memory is implemented, and how to use it, see "Memory sharing" in the "Using Virtual Devices, Networking, and Memory Sharing" chapter.

Sharing devices

Like memory, devices can't be shared directly. Virtual devices (whether emulation or para-virtualized) are either implicit (in the `qvm` process code), or they are shared objects configured into the VM; they are hence exclusive to the VM. Physical devices are either assigned directly to a guest as pass-through devices, or are assigned to the hypervisor host, and accessed by a guest through a virtual device.

If an entity other than the owner of a physical device needs to access that device, it must go through the device owner, just as it would if the device were physically on another board. If the device is owned by the hypervisor host, this is called *mediated sharing*; if it is owned by another guest, this is called *referred sharing*. For more information, see "Shared devices" in this chapter.

Page updated: August 11, 2025

Supported architectures and guest OSs

Supported hardware architectures

This release of QNX hypervisor uses a 64-bit kernel and supports the following 64-bit architectures:

- AArch64 on ARMv8 hardware
- x86-64 on x86 hardware

CPU privilege levels

A CPU privilege level controls the access of the program currently running on a processor to resources such as memory regions, I/O ports, and special instructions. A guest runs at a lower privilege than the QNX OS microkernel, and applications running in that guest run at an even lower privilege. This architecture provides hardware-level safety from untrusted software components.

See also "Exception Level (EL)" and "Ring" in the "Terminology" appendix.

PCI support

The QNX PCI vendor ID is 7173 (`0x1C05`). For more information about PCI Vendor IDs, see the PCI SIG website at **https://pcisig.com/**. For more information about the QNX PCI Vendor ID, please contact your QNX representative.

Supported guest OSs

This hypervisor release supports QNX OS and Linux guests for the hardware architectures specified above. The supported guests include:

- QNX OS 8.0
- Linux Ubuntu 22.04

Other guest OSs (e.g., Android, older QNX guests) are possible. For information about supporting these guests, contact QNX Engineering Services.

Guest OSs must be compiled for the hardware architecture on which the hypervisor host is running. For example, AArch64 guests can run on ARMv8 hardware only.

For all target platforms, the hypervisor host domain requires 64-bit hardware and supports 64-bit guests. Guests may run as single-core or multi-core; that is, a guest may run in a VM configured with a single virtual CPU (vCPU) or in a VM configured with multiple vCPUs.

NOTE:
32-bit guests are no longer supported. The QNX Hypervisor supports 64-bit guest OS environments such as Linux that can run 32-bit applications, but only on ARM targets.

The number of vCPUs in a VM affects performance. Adding vCPUs adds vCPU threads to the `qvm` process instance for the VM hosting the guest. Although independent of the number of hardware CPU cores, the number of vCPUs chosen by the system designer is often related to this number of physical cores. Increasing the number of vCPUs can improve performance, but there are circumstances where additional vCPUs may reduce performance. This is explained further in the "vCPUs and hypervisor performance" section.

Page updated: August 11, 2025

A note about nomenclature

Before starting to work with the QNX Hypervisor (QH) ("*the hypervisor*"), it is useful to understand the terminology we use to describe QNX virtual environments, the hypervisor, and its guests, as well as the nomenclature we use for filenames.

Terminology

Please note the following about the terminology used in this guide:

device
: We use *device* to mean a hardware device that exists as a physical component on the system. Contrast *vdev*.

guest
: A *guest* is a guest OS and any applications that are running on this OS; a `qvm` process instance (see below) *hosts* a guest. The guest runs in a VM (see below) created by the `qvm` process instance.

  If we need to distinguish between the guest OS and the applications, we spell it out (e.g., "To stop an application without stopping the guest OS ...").

host
: In most cases, the term *host* refers *not* to a host computer (the desktop or laptop, which you can connect to your target system to load a new image or debug) but to the hypervisor or something in the hypervisor (e.g., "The CPU instructs the guest thread to exit guest mode and continue in *host* mode").

  When the context doesn't make the meaning of *host* clear, we'll use *development host* to refer to the desktop or laptop, and *host domain* or *hypervisor host* to indicate the hypervisor and all of its components.

  When discussing privilege or exception levels, we'll use *hypervisor host* or *hypervisor host domain* (e.g., "Processes running in the hypervisor host domain sometimes run at EL2 on ARM boards, or at VMX root mode on x86 boards").

hypervisor
: A software system consisting of a QNX OS microkernel that has its hypervisor host mode enabled and can include one or more virtual machines.

  There are two products from QNX that you can use to create hypervisor systems: the non-safety QNX Hypervisor (QH) and the safety-certified QNX Hypervisor for Safety (QHS). For simplicity, we use the terms *QNX hypervisor* or *hypervisor* when describing design and implementation details that apply to both the QH and QHS variants.

qvm
: A *qvm* (or `qvm` process instance) is a process in the hypervisor that hosts a guest.

  The hypervisor starts `qvm` process instances; each `qvm` process instance presents a virtual machine (VM) in which a guest can run.

  A `qvm` process runs at times at the permission level of the hypervisor host domain (it is part of the hypervisor) and at times at the lower privilege level of the guest it is hosting.

vdev
: We use *virtual device* (or *vdev*) to mean any device that the hypervisor virtualizes in some way (see "Virtual devices" in this chapter). Examples of vdevs include an interrupt controller (virtualized), or an Ethernet device controller (para-virtualized).

  If we ever refer to a virtual device simply as a device, that is confusing and an error, so let us know about it.

VM
: A *virtual machine* (or *VM*) is presented by a `qvm` process instance to a guest, which runs in the VM; the VM hosts the guest.

NOTE:
See the Terminology appendix for a fuller glossary of terms.

Filenames

We use the following nomenclature to name the files in a QNX virtualized environment:

Prefixes

Prefixes identify where a file is used:

hypervisor

The file includes or is used to build and/or configure the hypervisor host domain. The prefix is followed by a release number.

vdev-

The file is a virtual device.

Suffixes

Suffixes identify the type of file:

.build

The file is a buildfile, for the QNX hypervisor host domain, or for a QNX guest.

For Linux and other non-QNX guests, see the documentation for those types of OSs.

.img

The file is a bootable image. This image may include only the hypervisor host domain, only a guest, or the host domain and one or more guests.

.qvmconf

The file is a configuration file for a VM; it is parsed by a `qvm` process instance to assemble the VM.

Examples

The following illustrates how we compose filenames:

Hypervisor build and image files

Names of buildfiles and image files for the hypervisor host domain are composed as follows: **hypervisor*release*-*board*.*type***, where *release* is the hypervisor release, *board* is the hardware platform, and *type* is the type of file (either **build** or **img**). For example: **hypervisor8.0-fooboard.build** and **hypervisor8.0-fooboard.img**.

Guest (QNX) build and image files

Names of guest buildfiles and images are composed as follows: ***guestos*.*type***, where *guestos* is the guest OS (e.g., **qnx80**, **linux44**), and *type* is the type of file (either **build** or **img**). If your QNX hypervisor system will have more than one instance of the same OS, add a letter to each instance; for example: **qnx80a.build** and **qnx80b.img**.

Linux and other guest OSs use their own nomenclature.

VM configuration files

Names of `qvm` configuration files are composed as follows: ***guest*.qvmconf**, where *guest* is the guest OS (e.g., **qnx80**, **linux44**) that will run in the VM configured by this file. If your QNX hypervisor system will have more than one instance of the same OS, add a letter to each instance; for example: **qnx80a.qvmconf** and **qnx80b.qvmconf**.

Page updated: August 11, 2025

Time

The flow of time in a QNX hypervisor system is more complicated than in a non-virtualized system, and is critical to many functions of the host OS and guest OS. The different pieces of time-related hardware in a hypervisor system interact in complex ways, and emulating them needs to go beyond the functionalities of each separate piece.

To understand the concept of time in this setting, you must consider the following:

- specific reasons why the flow of virtualized time can differ from real time
- the expected OS use cases for time functionality whose correctness must be maintained in a virtualized environment
- characteristics of the physical hardware devices that the virtualized time must account for
- the system design characteristics that support the necessary virtualized time properties

Difference between host and guest time flow

Time flows slightly differently in the guest than in the host because sometimes when a virtual CPU (vCPU) wants to run it gets delayed. This delay can occur in two ways:

- Preemption by another host thread, which could be another vCPU thread. This situation introduces what is called *stolen time*.
- Lengthy emulation. The emulation of even simple functions of any piece of hardware requires many machine instructions, because there is at least one guest exit and a subsequent guest entry. This is inevitably slower than accessing the actual hardware.

These delays mean that time doesn't always behave the same in the host and the guest. There is also the external timeline as experienced by any observer of the target system (i.e., the machine on which the hypervisor runs).

When discussing emulation, you must consider the cost of guest operations. Emulation is typically expensive but sometimes virtualized operation can take a notable additional amount of time compared to the same operation on the host. An operation is considered inexpensive only when there is no discernable cost of doing it in the guest compared to the host.

A hypervisor imposes significant overhead simply by running, and this affects the flow of time in the guest. The next section discusses the various OS use cases for time and the "Virtualization interference on time use cases" section further below discusses how the hypervisor overhead affects each of them.

Expected OS uses of time functionality

Time can be observed and experienced in different ways. Although this discussion considers the expectations about time mostly from the perspective of a guest, the time use cases are also relevant for the hypervisor host, just as they are relevant for an OS running in a non-virtualized environment.

There can be expectations about absolute time usage in a guest, but also expectations about the time usage relative to that in the host.

The host OS or guest OS can use time functionality in the following ways:

1. **Timestamping**

   For any observable event, a value is associated that uniquely identifies the time it happened from the perspective of a given vCPU. That value is a *timestamp*.

2. **Time counting**

   An OS can measure time between two observed events. Typically, an OS makes an equivalence between an amount of time and an amount of work done by a CPU, and uses this assumption in its scheduler. However, stolen time can break this equivalence.

   Accuracy in time counting is critical for the proper accounting of used CPU time and for setting correct deadlines.

3. **Alarms**

   A client process or thread can be notified that a certain amount of time has passed.

4. **Time of day**

   The OS can monitor "wall-clock time". For a guest, this implies it can synchronize with an entity that observes time outside of the guest itself.

5. **Intra-guest synchrony**

The host OS can tell, for a given guest, if an event on one vCPU happened before or after an event on another vCPU.

Here accuracy is important for a guest running inside a VM to set meaningful deadlines for workloads spread over multiple vCPUs.

6. **Inter-guest synchrony**

   An OS can tell if an event on one CPU—a vCPU for a guest, and a physical CPU (pCPU) for the host—happened before or after an event on another CPU. This an extension of intra-guest synchrony to different guests and the host. It is similar in concept to comparing events across a network of computers.

   This feature is useful for the end user and possibly host-side software to synchronize the actions from multiple guests and host processes. For example, in the collation of logs from guests and the host.

Hardware devices for time

The time use cases described above are supported by several pieces of hardware:

- a free-running counter that increments at a fixed frequency
- a timer that generates interrupts at controlled times
- a device linked to an atomic clock through any mechanism

The third component is of great interest for having an accurate time-of-day report, but is usually of little value for the other time use cases. A device of this sort is generally slower to access than the other kinds of devices and therefore in a hypervisor system would be either emulated or passed through to the guest.

For the other two components, their architecture-specific support is discussed in the next two sections. Also discussed is the hardware facility to force a guest exist after a defined amount of time. Although any timer could be used for this purpose, modern CPUs provide a specific facility. The reason is the need to emulate alarms.

x86 hardware

On x86 platforms, the free-running counter is called the Time Stamp Counter (TSC) and is a machine-specific register of Intel CPUs. This 64-bit counter starts at zero (0) and increments at a constant frequency based on the clock speed of the core. In the present day, it is a counter whose frequency never changes through any power transition of the core and is provided by hardware common to all cores on the CPU.

This register's value is accessed using the RDTSC machine instruction.

The Local Advanced Programmable Interrupt Controller (LAPIC) provides a timer that is generally preferred by modern OSs. On recent CPUs, it has a mode where the time at which an interrupt needs to be generated is determined by the TSC. Otherwise the LAPIC timer uses its own counter to generate interrupts.

Additionally, the Virtual Machine Extensions (VMX) provide a virtualized TSC that is simply offset from the host TSC by a value under the control of the Virtual Machine Manager.

Intel CPUs also provide a timer called the *VMX-preemption timer* that forces a guest exit. This timer counts down from a value set at guest entry at a rate that is a fraction of the TSC. When the timer reaches zero, if the guest has not exited for any other reason, the exit is forced.

ARM hardware

The ARMv8 specification mandates a Generic Timer that is accessed using system registers. All cores get a free-running 64-bit system counter from which are derived a physical counter and a virtual counter. The specification defines several timers based on those last two counters:

- the EL1 physical timer
- the EL1 virtual timer
- the EL2 physical timer (EL2 has to be available for hypervisors)

The timers can be used either with an absolute deadline (using the CVAL register, to set CompareValue), or with a relative deadline (using the TVAL register which sets CompareValue to the current counter value plus the written offset). They all have a distinct interrupt local to the core.

Additionally, software running at EL2 can configure an offset on the virtual counter value when used by EL1 and EL0 software. This feature allows a guest to access the virtual timer registers while still having the virtual counter value start at zero when the VM starts. (This is why it is called the virtual counter.)

The EL2 physical timer is the one that the hypervisor is meant to use for forcing a guest exit at a specific time. This is because the EL1 physical timer might be used by the host OS (and sharing would be complex) and the EL1 virtual timer is meant to be passed through to the guest.

The Virtualized Host Extensions (VHE) in ARMv8.1 also define the EL2 virtual timer, which is meant to be used by the host OS in place of the EL1 virtual timer. This makes it easier to provide guests with access to the EL1 virtual timer and in turn makes switching between the host and guest more efficient, which is the goal of VHE.

Features and terminology

The ARMv8 concepts are generic enough in function and name that they tend to be the preferred terminology. What is important is that while there are implementation differences, the Intel TSC and the ARMv8 virtual counter share many qualities that are relevant to a hypervisor:

- they provide a 64-bit free-running counter operating at a known frequency identical to the host
- guest access is inexpensive (in access overhead)
- the value seen by the guest can be offset from the host value

The Intel LAPIC timer and ARMv8 virtual timer also share similar qualities, although the ARMv8 virtual timer requires less emulation:

- the interrupt is local to the CPU
- it can be programmed with a relative or an absolute deadline
- a target value for the virtual counter can be used to trigger the interrupt

The VMX-preemption timer and EL2 physical timer share the following features:

- they can force a guest exit at a scheduled future time
- their resolution is close or identical to the system counter
- they don't interact with any other CPU on the system

To avoid confusion with the ARMv8 devices, a QNX hypervisor VM has guest virtual counters and guest virtual timers. The hypervisor uses the virtual timer to control how long a vCPU can run without a guest exit.

Emulated time-related hardware

Some platforms have legacy hardware that is related to time functionality. Occasionally, it is necessary to emulate such hardware. On x86-64, this includes the 8254, MC146818, and HPET devices. There can be other specialized hardware needing to be emulated: for instance, watchdogs are critical for safe OSs.

While the virtualization extensions and tools offered by modern CPUs help emulate basic time-related functions, the vdevs that emulate such devices must adhere to the QNX hypervisor design. For this purpose, the QNX OS provides a timer API. For information about this interface, refer to the *Virtual Device Developer's API Reference* in the QNX Hypervisor GitLab Repository at https://gitlab.com/qnx/hypervisor.

Virtualization interference on time use cases

A vCPU can be seen as driven by a clock, just like a pCPU. The clock tick provides a timeline to that vCPU.

Whenever a vCPU thread is running, it is either executing guest code or emulating hardware in response to a guest access. In either case, the vCPU's clock is considered running. This clock can be a free-running incrementing counter like the ARMv8 system counter or Intel TSC.

It is assumed that a guest has an inexpensive way of accessing that counter's value for the currently executing vCPU. This assumption means that the timestamping, time counting, and intra-guest synchrony use cases always have as high a precision as the host, at least for the time measurement itself. However, the counter values can still be wrong in other ways: for instance, they might not be identical in all vCPUs at all times.

The hardware makes accessing the clock value inexpensive and fast by having the value observed by the host and the value observed by the guest differ by an offset that is written into a specific register.

The alarms and time of day use cases are relative to an external observer, whether it is the host or something outside the host. Discrepancies in those cases would be observed relative to a secondary time source such as a GPS receiver or the operator's stopwatch.

System design that supports virtualized time

The hypervisor affects the time use cases for a guest running in a VM. QNX Hypervisor is based on these hardware and software design guidelines:

- The hardware supports an inexpensive virtual counter, which means that no guest exit needs to occur for the guest to read that counter.
- The counters on all host pCPUs are synchronized. Using different units on different CPUs would be impractical, so it is expected that the counters have the same value on all pCPUs at all times.

- The system operator or a program in the host domain can translate a value in the guest timeline into a value in the host timeline.
- The host and guests may not go out of synchronization for more than a defined amount of time. This includes all time-related components: counters, timers, and alarm notifications.
- A guest receives information about disruptions in its timeflow; this helps it to better schedule its tasks. The information includes but is not limited to stolen time, known discrepancies, and corrections.
- Intra-guest synchrony is preserved. For a given guest, you can tell if an event on one vCPU happened before or after an event on another vCPU.
- There is no extra cost for a guest being able to tell the order in time between two events happening on different vCPUs compared to the host being able to tell the order of two events on different pCPUs.

Guest time counter

Guests have an inexpensive virtual counter, and whenever one vCPU in the underlying VM is running, the counters on all vCPUs in that VM are considered as running. It is thus possible to have just one counter for the whole VM and make the virtual counters used by the vCPUs match its value. This mechanism, combined with the fact that all host pCPU counters are synchronized, ensures that intra-guest synchrony and inter-guest synchrony (see the above explanations) are preserved by the QNX Hypervisor.

The problem of time-counting

The time counting use case is more challenging to preserve. Although time counting is preserved during a guest entry/exit cycle of a vCPU, it becomes a problem when there is a guest exit. While a vCPU has exited the guest, two things can happen:

- Preemption — the vCPU thread can be preempted by the host to run another thread.
- Emulation — the vCPU can emulate the effects of the instruction that triggered the exit (if the exit was not forced by an interrupt).

Emulation can be complex: for instance, a vdev might have to send a message to a server and wait for a reply. This is quite expensive compared to how long it would take for actual hardware to perform this task.

Is it fair to keep the guest clock running during such operation? The problem is that the guest's scheduler doesn't know that the hypervisor makes the operation expensive. But if the guest clock is kept running, then it is going to bill all those cycles to the thread that caused the exit.

A guest thread that triggers a lot of guest exits can therefore be seen as consuming more CPU time than a thread that doesn't cause an exit. For that reason, it can be tempting to turn off the guest clock during an emulation, and when re-entering the guest, increment the clock by an amount that's fair from the point of view of the guest scheduler.

But as mentioned previously, when one vCPU is running, all of them are considered as running. This means if one vCPU is doing emulation and another is running guest code, then it is not possible to turn off the guest clock. In fact, this could be done only when none of the vCPUs is running. Also, turning off the clock introduces a difference between the flow of time in the host and in the guest.

For these reasons, the guest clock is never turned off. This means that the guest and host timelines are identical, except for a constant offset separating them. As also mentioned previously, the system operator or a host program can translate a value between these timelines.

Exposing stolen time

The information that is most useful for a guest scheduler is how much time each vCPU thread wanted to run but couldn't because of the host scheduling. This is what we call *stolen time*. However, it is not feasible to measure this time accurately. This is because any accurate measurement of stolen time would have to include the time spent in the READY state for other threads that the vCPU thread was blocked on (e.g., for message passing). Implementing a way of measuring this other time would be extremely complex.

Therefore, all the time not explicitly running guest code is considered stolen and reported to the guest as an approximation of the true stolen time. This is still better than not reporting any stolen time, and it helps the guest in scheduling its tasks.

The API that a guest OS uses to read the stolen time is architecture-specific. For a guest running on an ARM platform, this is accessed via the SMCCC machine instruction. For details, refer to the following sources:

- *Arm Paravirtualized Time for Arm-based Systems, v1.0 Arm DEN0057A-b*, available at: https://developer.arm.com/documentation/den0057/latest/.

- *SMC Calling Conventions 1.2 Arm DEN0028C*, available at: https://developer.arm.com/documentation/den0028/c.

On x86 platforms, there is no standard way of accessing the stolen time. Some hardware allows you to read the stolen time via the RDMSR instruction, in which case the register RCX must contain the value `0x44563400ul`.

Alarms

The hypervisor can emulate many different counters and timers for a guest, but they should all use the same underlying clock in the vCPU as described above in "Virtualization interference on time use cases". This design prevents different emulated timers from drifting from each other (i.e., some lagging behind others) in the guest. Regardless of the quality of the vCPU clock, there is no point in adding noise to it.

Time of day

All other use cases of time are relevant on short-term scales (e.g., a few milliseconds), whereas time of day is a long-term experience. If there are enough discrepancies in the frequencies of the various counters, this can introduce time lagging, or *drift*, compared to an external clock.

Although the hypervisor is likely to introduce more drift, all techniques applicable to the host (e.g., an attached device, or network communications like NTP) are also applicable to guests. You can therefore just pass through a clock device or use networking facilities to provide an accurate time of day for a guest.

Effect of Waiting-for-Interrupt state on guest time

In a hypervisor system, the amount of time that a vCPU can stay in the Waiting-for-Interrupt state is bounded by the next scheduled interrupt. A scheduled interrupt is an alarm; that is, a notification of a timer condition. Thus, the emulation of the Waiting-for-Interrupt state must respect this design.

Alarms are handled by the Virtualization Timer while a vCPU thread is executing. One way of dealing with the Waiting-for-Interrupt state is simply to let it run normally and not consider it as a cause of a guest exit. Both the x86 and ARM architectures allow for that through their virtualization. Except for power usage, the guest behaviour is then indistinguishable from simply spinning until an interrupt happens. In the host, the vCPU thread is simply in the RUNNING state.

If the hypervisor makes the guest exit when it enters the Waiting-for-Interrupt state, this is so it can use the pCPU on which the guest was running to run other host threads. The vCPU thread must then stay in a blocked state until an interrupt is injected. However, an alarm can (and most likely will) have been scheduled by the guest, so there is a limit to how long that thread can be in that blocking state.

In QNX OS, you can call *TimerTimeout()* with the relevant options to create a timer with a certain tolerance. Controlling the tolerance of a timer created to emulate the Waiting-for-Interrupt state directly affects the accuracy of any alarms that the guest has scheduled.

As mentioned at the end of the "The problem of time-counting" section, the guest and host timelines are identical except for a constant offset. However, the accuracy of the timeout to emulate the alarm affects the accuracy of the notification sent to the guest.

If the guest doesn't use a regular tick or expose a high-resolution timer, then it is affected by any lack of accuracy of the Waiting-for-Interrupt timer. Even if it has a regular tick, to ensure accuracy, the guest has to run at a frequency significantly lower than that of the host (about twice as slow, like with signal sampling). This is why QNX Hypervisor allows you to set the tolerance of the Waiting-for-Interrupt timer—to balance the needs of the guest with those of the host.

These considerations apply to the Virtualization Timer only if its precision is lower than the system counter. It is assumed that whatever mechanism the hardware offers this timer has the best accuracy possible.

Page updated: August 11, 2025

Virtual devices

A virtual device can emulate a physical device, or it can provide functionality like that provided by a physical device without emulating any specific device.

Virtual devices (or *vdevs*) exist only in virtualized environments. To use a vdev, a guest requires a driver, just as it would require a driver to use a physical device in a non-virtualized environment.

A vdev might never access a physical device, or it might act as an intermediary that both responds directly to a guest and passes requests and responses between a guest and a physical device.

In a hypervisor system, the hypervisor provides vdevs in the `qvm` process instance for a VM. The guest hosted in the VM receives interrupts from each vdev and sends signals to it, exactly as though it were working with a physical hardware device. The guest has no direct communication with any physical device on the system; such a device might not even be present.

The guest requires a device driver for each vdev it will use.

The hypervisor supports the following types of vdev:

- Emulation vdevs
- Para-virtualized devices

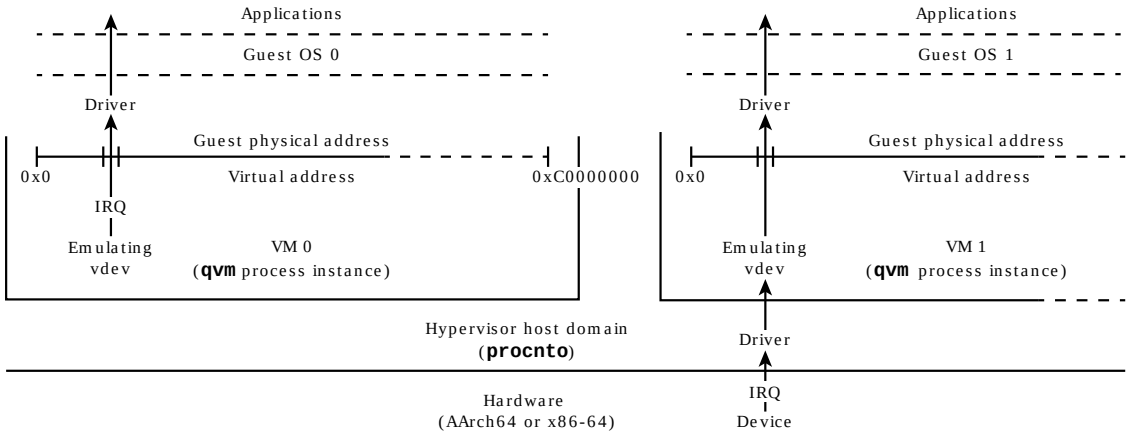
For information about how to write your own vdevs, see the *Virtual Device Developer's Guide* in the QNX Hypervisor GitLab Repository at https://gitlab.com/qnx/hypervisor.

Emulation vdevs

An emulation vdev is a virtual device that emulates an actual physical device for a guest. The following are key characteristics of emulation vdevs in a QNX virtualized environment:

- To use the vdev, a guest doesn't need to know that it is running in a virtualized environment. It interacts with these devices exactly as it would with physical devices, and it has no indication that it is working with a virtual device or that no hardware device might be involved.
- The vdev emulates a physical device. In fact, some vdevs (such as an x86 `pckeyboard`) exist simply because a guest running on an x86 platform expects them to be there.
- The emulated physical device may or may not exist on the system.
- Depending on the type of physical device, the emulation vdev may either handle everything itself (e.g., emulate a timer chip, as does `vdev timer8254`), or it may act as an intermediary between the guest and an actual physical device (e.g., `vdev ser8250`).
- The guest can use the same device driver for the vdev as it would use for the emulated physical device if it were running in a non-virtualized environment.

Figure 1Illustration of emulation vdevs in a hypervisor system.



Para-virtualized devices

Para-virtualized devices are a type of virtual device implemented as software code that runs in the hypervisor layer, but this code doesn't emulate any specific hardware device.

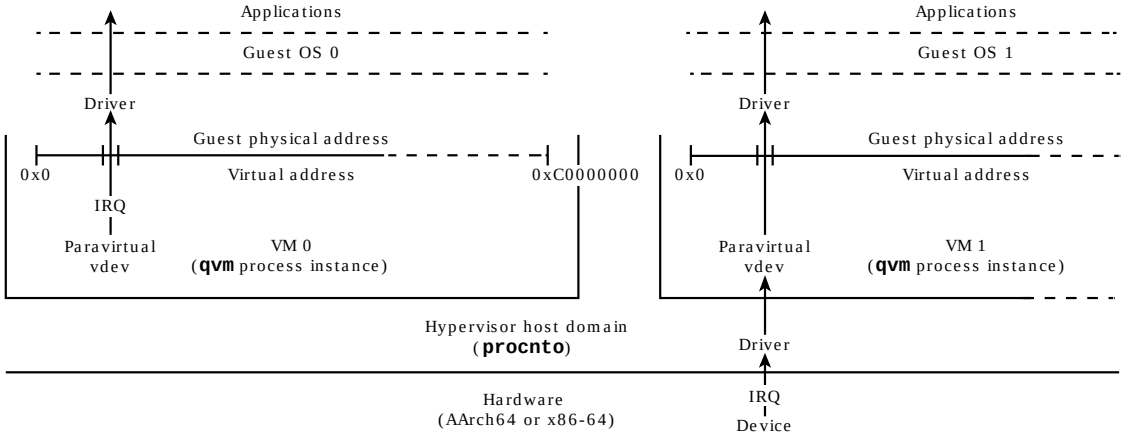
Instead, a para-virtualized device provides the functionality that might be provided by a physical device (or several physical devices) in a non-virtualized environment, but without the constraints of emulating a piece of hardware.

A typical para-virtualized device requires code running in the guest (front-end), and code running outside the guest (back-end); for example, a filesystem device driver running in the guest that connects to a block device driver running in the hypervisor host.

Key characteristics of para-virtualized devices include:

- To use a para-virtualized device, the guest must know that it is running in a virtualized environment. For example, to use `virtio-net`, the guest must know that it is running in a virtualized environment, if only because `virtio-net` has no corresponding physical device (it exists only as a para-virtualized device).
- There exists no physical device that corresponds exactly to the para-virtualized device.
- The guest must have appropriate drivers and interfaces to work with the para-virtualized devices.

Figure 2Illustration of para-virtualized devices in a hypervisor system.



Para-virtualized devices require some initial investment (e.g., to write drivers for guests), but these devices may often be more efficient than vdevs that must emulate a hardware component. For example, emulating a serial port interface that provides a console terminal can result in many guest exits and returns. A `virtio-console` vdev (a para-virtualized serial port interface) provides the same functionality, but with less overhead in the guest.

NOTE:
Para-virtualized devices don't have to be VIRTIO devices; VIRTIO is simply a convenient and currently popular standard.

Page updated: August 11, 2025

Understanding QNX Virtual Environments

QNX hypervisors are designed to meet the expectations of a hypervisor specified by the Popek/Goldberg Theorem.

CAUTION:
Read this chapter and understand its contents *before* you attempt to use the hypervisor. A few minutes spent with this chapter now will save you a great deal of time in the future.

The Popek/Goldberg Theorem

The Popek/Goldberg Theorem specifies that a hypervisor should meet the following three criteria:

Equivalence

Virtual machines (VMs) running in the hypervisor are essentially the same as the underlying hardware. A guest does not need to be aware that it is running in a VM in order to function properly.

The above statement does not preclude the use of para-virtualized devices or other strategies that require virtualization awareness. Such strategies may be used to provide functionality and improve performance.

Safety

With the exception of guest access to pass-through device memory, the hypervisor maintains control of the hardware at all times, regardless of what the guests are doing. It controls guests' abilities to access hardware devices, limits guests' ability to access host-physical memory to their assigned memory regions, has ultimate control over scheduling, manages interrupt routing, and has the ability to terminate a guest, regardless of what the guest may be attempting to do.

Performance

Execution of programs running in VMs is only minimally slower than when running directly on the hardware.

See “The Popek/Goldberg Theorem” in Edouard Bugnion, Jason Nieh and Dan Tsafrir, *Hardware and Software Support for Virtualization* (Morgan & Claypool, 2017).

Page updated: August 11, 2025

cmdline

Pass a string to a guest as though the string had been entered from the command line

Synopsis:

```
cmdline commandline
```

Description:

You can pass the string specified by *commandline* to a Linux kernel or to multiboot (ELF) images used for QNX guests.

The `cmdline` option makes command-line input available to the guest in various ways, depending on the guest image format and the guest architecture. How the input is interpreted is the responsibility of the guest.

Example:

The following connects a terminal console and sets up log and debug parameters for a Linux guest running on a virtualized ARM platform:

```
cmdline "console=ttyAMA0 earlycon=pl011,0x1c090000 \
debug user_debug=31 loglevel=9"
```

Page updated: August 11, 2025

cpu

Create a new vCPU in the VM

Synopsis:

```
cpu [options]*
```

Options:

`cluster` *`cluster_name`*

Allow the vCPU to run only on the physical CPUs (pCPUs, or cores) within the specified cluster; this is known as *core pinning*. A cluster is a group of associated pCPUs. The startup code defines the clusters on the system, including their names and CPU mapping. More details about clusters are given in the "Processor affinity, clusters, runmasks, and inherit masks" section of the QNX OS *Programmer's Guide*.

Assigning vCPUs to clusters implies important design choices. If a vCPU is assigned to a cluster with multiple pCPUs, then it is said to be *floating*, which means the vCPU may migrate. Migration is useful on some systems, because with migration a vCPU will move to another core that is free in the same cluster. However, for some realtime guests, assigning a vCPU to one core that isn't shared with other vCPUs can improve realtime determinism.

The default for a vCPU is no restriction to a particular cluster (i.e., no core pinning), which is another case of floating. QNX recommends that you first use floating vCPUs in your design, then move to restrict or prohibit migration as required.

`sched` *`priority`*`[r | f | o ]`

`sched` *`normal_priority,low_priority,max_replacements,replacement_period,initial_budget`* `s`

Set the vCPU's scheduling priority and scheduling algorithm (or policy). The algorithm can be any of the ones supported by QNX OS: round-robin (`r`), FIFO (`f`), or sporadic (`s`). The other (`o`) algorithm is reserved for future use; currently it is equivalent to round-robin.

The default vCPU configuration uses round-robin scheduling for vCPUs. Our testing has indicated that most guests respond most favorably to this algorithm. It allows a guest that has its own internal scheduling policies to operate efficiently.

For more information about sporadic scheduling, see "Configuring sporadic scheduling" below.

Note that the vCPU's priority is independent of the priority of any guest thread in the VM and hence, the thread scheduling in the host isn't affected by the relative importance of different guest threads. For further explanation, see "Controlling guest behavior through vCPUs" in the "Configuration" chapter.

Description:

The `cpu` option creates a new vCPU in the VM. Every vCPU is a thread, so a vCPU can be assigned to a cluster to restrict the pCPUs on which it runs. Similarly, standard (QNX OS) scheduling priorities and algorithms can be applied to the vCPU. Note that vCPU threads are threads *in the hypervisor host domain*.

If no `cpu` option is specified, the `qvm` process instance creates a single vCPU. Determining the optimal number of vCPUs for the VM (and hence, the guest) depends on a few factors, such as performance considerations, as described in "vCPUs and hypervisor performance" in the "Performance Tuning" chapter, and the hardware and OS limits, as explained in the next section.

For information about how priorities for hypervisor threads and guest threads are handled, see "Scheduling" in the "Understanding QNX Virtual Environments" chapter.

Maximum vCPUs per guest

Each hypervisor VM can host only one guest OS, so the maximum number of vCPUs that can exist in a VM is also the maximum number that may be defined for the guest OS running in that VM. This limit is the lower of two values determined by the following factors:

Hardware

On supported AArch64 (ARMv8) and x86-64 platforms, the hardware currently allows a maximum of 254 vCPUs on the board. This number may change with newer hardware.

> **CAUTION:**
> QNX strongly recommends that you don't give a guest more vCPUs than there are pCPUs on the underlying hardware platform, or unpredictable behavior will result.

Specific hardware components may further limit the number of vCPUs per guest. For example, on x86-64 boards, the LAPIC virtualization limits a guest to a maximum of 15 vCPUs. On AArch64 boards, the maximum number of vCPUs per guest is limited by the version of the GIC vdev. For a vdev running in GICv3 mode the maximum number of vCPUs allowed for a guest is eight (8).

Guest OS

Current QNX OSs support a maximum of 64 CPUs. This limit also applies to vCPUs, since a guest makes no distinction between a pCPU and a vCPU. For other types of guests, check the latest documentation for their OSs to learn about the maximum number of CPUs they support.

Configuring sporadic scheduling

NOTE:
Before using FIFO or sporadic scheduling, consult with QNX engineering support.

For sporadic scheduling, you need to specify the following five parameters:

- *normal_priority* – the normal priority value
- *low_priority* – the low priority value
- *max_replacements* – the maximum number of times the vCPU's budget can be replenished due to blocking
- *replacement_period* – the number of nanoseconds that must elapse before the vCPU's budget can be replenished after being blocked, or after overrunning *max_replacements*
- *initial_budget* – the number of nanoseconds to run at *normal_priority* before being dropped to *low_priority*

Examples:

Example 1: pinned vCPU, set scheduling priority and algorithm

The following creates a vCPU that is permitted to run only on the `_cpu-3` cluster, which is a predefined cluster that contains only pCPU 3:

```
cpu _cpu-3 sched 8r
```

Because the vCPU is pinned to a single pCPU, it is not floating and may not migrate. The priority of the vCPU thread is 8, and the scheduling algorithm is round-robin.

Example 2: non-pinned vCPUs, set scheduling priority

The following creates four vCPUs (0, 1, 2, 3), all with priority 10:

```
cpu sched 10
cpu sched 10
cpu sched 10
cpu sched 10
```

The `cluster` option isn't specified, so the default of no core pinning is used. The microkernel scheduler can run each vCPU thread on whatever available pCPU it deems most appropriate.

Example 3: two pinned vCPUs, default scheduling

The following creates four vCPUs (0, 1, 2, 3), with restrictions on pCPUs for only the first two:

```
cpu cluster _cpu-2, _cpu-3  # vCPU 0 may run only on cores 2 or 3.
cpu cluster _cpu-2, _cpu-3  # vCPU 1 may run only on cores 2 or 3.
cpu  # vCPU 2 may run on any pCPU.
cpu  # vCPU 3 may run on any pCPU.
```

For vCPUs 0 and 1, their `cluster` options are set to restrict them to clusters that contain only pCPU 2 and only pCPU 3. They won't migrate to pCPU 0 or 1 even if either is idle. No `cluster` option is specified for vCPUs 2 and 3, so they will use the default of no pinning, meaning they can run on any available pCPU (including 2 and 3).

No scheduling options are configured so the default scheduling priority and algorithm are used.

Page updated: August 11, 2025

`dryrun`

Initialize a VM without starting the guest

Synopsis:

```
dryrun
```

Options:

None

Description:

Specify the `dryrun` option in a VM's configuration to have the `qvm` process instance initialize the complete environment in which a guest will run, then exit before it starts the guest. If the VM configuration isn't valid, the `qvm` process instance logs an error before exiting. Error logging will use the mechanisms configured by the guest (see the `logger` option in this chapter).

This option is especially useful when working with FDTs, and can be used along with the `set` option's *fdt-dump-file* argument (see the "Common variables" section in the "Configuration variables" reference, and "FDTs (ARM)" in the "Configuration" chapter).

Page updated: August 11, 2025

dump

Write a dump file for the guest

Synopsis:

```
dump directory
dump |shell_command
```

Options:

directory

The output directory in the host.

or:

|shell_command

If the first character of the argument is a pipe, the characters that follow it are spawned as a shell command in the host, and the contents of the dump file are written to the shell's standard input.

Description:

On receipt of a SIGUSR2 signal or when triggered by a watchdog (see "Watchdogs" in the "QNX Hypervisor: Protection Features" chapter), write a dump file for the guest hosted in this `qvm` process instance. If the argument does *not* start with a pipe ("`|`"), write the dump file to the directory specified by *directory*. If the argument starts with pipe, spawn a shell command instead.

The dump file

The dump file is called ***prefix-YYYYMMDD-hhmmsss*.gcore** where:

- *prefix* is the name of the VM hosting the guest, as specified by the `system` option; if this option isn't specified (see system in this chapter), *prefix* is set to **qvm-guest**
- *YYYYMMDD-hhmmss* is the host date and time at the moment that the dump is requested

If a file with the name that the `qvm` process composes for the dump file already exists in the specified directory, the `qvm` process attempts to generate a unique name by inserting a hyphen and a number from 1 to 9 (**-1**, **-2**, ... **-9**) before the **.gcore** extension. For example, if the files **qnx8-20170821-120619** and **qnx8-20170821-120619-1** exist, the `qvm` process creates a file called **qnx8-20170821-120619-2**. If the `qvm` process is unable to create a unique filename, it doesn't create a dump file.

The dump file is output in ELF64 format. A `PT_NOTE` segment has the register states for the vCPUs, and `PT_LOAD` segments have the guest memory contents. The format of the `PT_NOTE` segment is described by the **sys/kdump.h** public header file.

To interpret the dump file, you can use GDB and `kdserver` (see `kdserver` in the QNX OS *Utilities Reference*). You can also run `use -i dump_file` to display much more information than usual to assist with troubleshooting. The information displayed in this case includes:

- `pidin` information about the host machine that `qvm` is running on
- information about the version of `qvm` that is running
- the `qvm` configuration being used
- the same output you get when you send SIGUSR1 (QL_QVM_INFO, or information-level, messages)

For more information about this last utility, see the `use` entry in the *Utilities Reference*.

Example:

On receipt of a SIGUSR2 signal, the configuration excerpt shown below will direct the guest's dump file contents to a host shell command that compresses these contents (`gzip`) and then writes the compressed file to `stdout`, which is the file **dump_output.gz**:

```
dump "|gzip >dump_output.gz"
```

NOTE:
The quotation marks around the argument are used so that when `qvm` parses the configuration it reads the full shell command as a single `qvm` argument.

generate

Explicitly request the generation of the specified system information table type in the guest

Description:

See “suppress” in this chapter.

Page updated: August 11, 2025

load

Copy the contents of a file into the guest system address space

Synopsis:

```
[blob_type] load [address,]filepath
```

Options:

address

The location in the guest where *file* is to be loaded. This address is the guest-physical address (the address as seen by the guest, not the host).

If *address* isn't specified, and the `qvm` process recognizes the file content type (*blob_type*), the `qvm` process loads the file content to the location indicated by this type. If the `qvm` process can't identify the type of content, it loads the contents of *file* to the first available location it finds.

The only requirement for the address is that it be in the guest's allocated guest-physical memory (see "Memory" in the Understanding QNX Virtual Environments chapter).

blob_type

The type of content. If *blob_type* isn't specified, the `qvm` process attempts to identify the type of file content and load it appropriately (e.g., if it sees an ELF or Linux image format, it will load the data at the locations indicated by the file contents and configure the guest bootstrap CPU to begin execution at the entry point specified by the file).

If the first file being loaded doesn't have a recognized file format (again, with *blob_type* not specified), the `qvm` process configures the guest bootstrap CPU to begin execution at the first byte of the file.

If *blob_type* precedes a `pass` option, the `qvm` process passes the data of the specifed type to the guest (see "pass" in this chapter).

filepath

The path and name of the file to load.

Description:

The `load` option copies the contents of a file into the guest system's address space. It can perform processing of the file contents, depending on the content type (*blob_type*) specified before the option.

The `qvm` process recognizes the following file content types:

acpi

```
acpi load [addr,]filepath
```

Available only for x86. The contents of *filepath* are loaded into guest memory and provided as an additional ACPI table. If *addr* is specified, the table is placed at the specified guest-physical address (see "ACPI tables and FDTs" in the "Configuration" chapter).

data

```
data load [addr,]filepath
```

The `qvm` process loads the contents of *filepath* into guest memory. If *addr* is specified, the data is placed at that address in guest-physical memory.

The file isn't considered to be a bootable image; the `qvm` process won't configure the bootstrap guest CPU to begin execution at the entry point indicated by the file contents. If the `qvm` process recognizes the file format, it will perform normal load processing for that file format (e.g., if the file is an ELF file, the `qvm` process places the contents according to the ELF segment table).

fdt

```
fdt load filepath
```

Available only for ARM. The file contains a flattened device tree (FDT) binary blob; the `qvm` process will add its automatically generated information to this FDT blob, write it all into the guest memory, and pass the location of this information to the guest OS during bootup (see "ACPI tables and FDTs" in the "Configuration" chapter).

guest

```
guest load [addr,]filepath
```

The `qvm` process will load the contents of *filepath* into guest memory. These contents will be interpreted as being the guest OS boot image.

initrd

```
initrd load [addr,]filepath
```

The `qvm` process loads the contents of *filepath* in guest memory and makes them available to a Linux guest as the initial RAM disk (initrd) filesystem. If *addr* is specified, the contents of the file are placed at the specified guest-physical address.

raw load

```
raw load [addr,]filepath
```

The `qvm` process copies the file byte for byte into the guest, even if its format would normally be handled by the `qvm` process. It performs no processing of the file contents.

Example:

The following loads a QNX IFS into the address space for the guest that will run in the VM defined by the current `qvm` configuration file:

```
load /vm/images/qnx8.ifs
```

Since only the filepath and name are specified, the `qvm` process creating the VM will examine the blob and place it in the location specified in the ELF headers.

Page updated: August 11, 2025

logger

Filter messages to the output, and specify the location where these messages are output

Synopsis:

```
logger filter[,filter,...] output_dest
```

Options:

***filter*[,*filter*,...]**

Output information for the specified message types. Specify one or more types, separated by commas (see below).

output_dest

The output destination for the messages specified by the filters. Specify only one output per `logger` instance. Supported destinations are `stdout`, `stderr`, `slog`, and `none`. Note that `slog` stands for "system logger", which is `slogger2`.

Description:

You may specify from zero to many `logger` options for a VM.

Log messages are filtered by the severity of the condition that caused a message to be emitted. Supported message severity types are:

Filter	Severity
`fatal`	A fatal error was detected; the `qvm` process can't continue.
`internal`	An internal error in the `qvm` process was detected; the process terminates right away.
`error`	An error that may or may not cause the `qvm` process to stop, but does indicate a problem that requires immediate attention, was detected.
`warn`	The `qvm` process can continue, but has encountered a problem that should be addressed.
`info`	Output information is requested by the user (e.g., in response to a SIGUSR1 signal).
`debug`	Provides information useful to QNX when debugging user-reported issues.
`verbose`	Provides users with detailed information useful for debugging their system.

NOTE:
The hypervisor always sends internal error logs to the `slog` output destination. Thus, `logger internal slog` is always on.

Using `logger` filters

Filters are *not* like verbosity levels, where each level outputs increasingly trivial messages. Filters are combined. You must specify the filter for every type of message you want to output. For example, specifying `logger error stderr` outputs *only* errors. It doesn't output errors and fatal errors. To get both, you must specify `logger error,fatal stderr`.

Default configuration

If no `logger` option is specified in the `qvm` configuration file, to ensure that all information, warning, and error-related messages of any type are sent to `stderr`, the following `logger` configuration is assumed:

```
logger fatal,internal,error,warn,info stderr
```

Examples:

An explicit configuration overrides the default configuration. For example:

```
logger error,warn,info slog
```

overrides the default output location (i.e., `stderr`), and directs output to `slog` only.

If you want to output to both `stderr` and `slog`, you must specify both output destinations separately, as follows:

```
logger error,warn,info slog
logger error,warn,info stderr
```

By default, certain log messages are directed to `stderr`. Using the `none` output destination tells a log filter to not log certain messages at all:

```
logger info,verbose none
```

In this case, the `none` destination is specified for `info` and `verbose`, meaning these message types aren't logged.

In the following `logger` option examples, only one output destination is specified per line:

```
logger warn,error,fatal stderr
logger info stdout
logger error,fatal,info,warn slog
logger error,fatal,info,warn stderr
logger internal,debug slog
```

Note that `error,fatal,info,warn` is repeated to direct the output to both `slog` and `stderr`.

Page updated: August 11, 2025

pass

Configure physical devices or data blob types passed through from the host to a guest

Synopsis:

```
[blob_type] pass [loc options] [intr guest_intr[,u][=vector_number]]
```

Options:

blob_type

The *blob_type* setting affects only the `loc` option. If *blob_type* is specified, then all locations specified by the `loc` options that follow (until the next context) provide data of the type *blob_type* to the guest that will be hosted in the VM being configured.

If *blob_type* precedes a `load` option, the `qvm` process copies the contents of the specified file into the guest system address space (see "load" in this chapter).

`intr` *guest_intr*[,u][=*vector_number*]

Pass the `guest_intr` interrupt through to the guest. This argument is associated with a host vector number, either *vector_number* if this value is specified, or the value specified by the `hostvector` vdev option for the PIC identified by *guest_intr*.

The host system uses an interrupt service thread (IST) to deliver *guest_intr* into the guest system. In this release, there's one thread for each passed-through interrupt.

The following attributes specify special behaviors:

u

The `u` attribute permits immediate unmasking of interrupts to a pass-through device, eliminating the need for a guest exit in order to unmask the EOI.

> **WARNING:**
> This attribute may be used only if your system has been configured to meet certain conditions, including:
>
> - Interrupts in the hypervisor host are edge-triggered.
> - Interrupts in the guest are edge-triggered.
> - The guest's device driver won't create new interrupt conditions, including when it is clearing the interrupt.
>
> Incorrect implementation of this feature might create a race condition and result in the loss of the device from the system. Contact your QNX representative before attempting to implement this feature.

`loc [{mem:|io:}]`*region*`,`*length*`,{+|-|r|w|x|c|e|m|d|n|s|t}[,=`*host_location*`]`

The host resource is made directly available to the guest starting at *region* and continuing for *length* bytes in guest-physical memory. The newly accessible region is typically memory, which is optionally prefixed with `mem:` (because this is the default resource type), but on the x86 you can prefix the location with `io:` to pass I/O space through.

The *length* argument is a 64-bit value.

> **CAUTION:**
> When passing through memory, the host-physical starting address of the memory (which gets mapped to *region* in the guest) must be aligned to a page boundary, and the *length* value must be a multiple of the page size on the host system. If not, the guest may experience I/O failures.
>
> These restrictions apply also when using `$asinfo_start{`*name*`}` and `$asinfo_length{`*name*`}`. For more information, see "VM configuration syntax" in the "Configuration" chapter.

The length setting can be followed by access type attributes, which may be a combination of the following symbols:

Argument	Description
+	Attributes following are added to the region (this is the initial state).

Argument	Description
`-`	Attributes following are removed from the region.
`r`	Read
`w`	Write
`x`	Execute
`c`	Cachable
`e`	Report exception
`m`	Usable as system memory (implicitly specifies `+rwxcstn`). If you specify this attribute, you must not specify `d`.
`d`	Device uses DMA. If you're using `smmuman` to manage DMA, you must specify either this attribute or `n`. You may, however, choose to manage DMA without `smmuman`; for more information, read the note about the `d` semantics just below. If you specify `d`, you should specify at least the `r` and `w` access types.
`n`	Device doesn't use DMA. If you're using `smmuman` to manage DMA, you must specify either this attribute or `d`.
`s`	Region can be the source of a DMA request.
`t`	Region can be the target of a DMA request.

If no access types are specified, `rw` is assumed.

The last part of the `loc mem:` arguments string may be an equal sign followed by the host-physical address of the device on the system. For some but not all devices, you can provide the guest with an address that is different from the host-physical address (i.e., the device is at host location A, but the guest sees it at location B).

For example:

```
pass loc mem:0x2000,r=0x1000
```

specifies a read-only location at guest-physical address of `0x2000` for a device at the host-physical address of `0x1000`.

CAUTION:
You can use the `pass loc` option to pass RAM (memory) through to a guest, but the RAM doesn't get zeroed, neither before the guest is given access to it, nor afterwards (e.g., after a VM termination). If you pass through RAM to which another guest system had access, remember that, unless you zero it, this memory may contain vestigial data from a previous guest system.

For the `io:` case, the location is a port number. For example:

```
io:4
```

specifies an x86 I/O device on port 4. You can provide a length and specify access type flags, but no host address is needed.

NOTE:
If you set the `d` attribute to specify that the device uses DMA, the `qvm` process by default expects to use the `smmuman` service to ensure that the DMA configuration is safe and secure. If this service isn't running, `qvm` outputs a fatal error message and aborts without starting the VM.

If, however, the `smmu-externally-managed` variable has been enabled (set to `on`), the `d` attribute has no effect and `qvm` outputs an information message saying so. Enabling this variable tells `qvm` to trust that the DMA configuration normally addressed by `smmuman` has been taken care of in some other way. Note that in this case, the VM configuration must not contain the `smmu` vdev or `qvm` reports a fatal error and aborts.

For more details about the `smmu-externally-managed` variable, refer to the Configuration variables reference.

loc pci:*pci_vid*/*pci_did*

The PCI device identified by the given vendor ID (*pci_vid*) and device ID (*pci_did*) is made available to the guest.

> **NOTE:**
> To support pass-through PCI devices, you must include the **pci_server-qvm_support.so** module in your hypervisor host image. For instructions on doing so, see the pci-server reference in the QNX OS *Utilities Reference*.

loc pci:*pci_bus*:*pci_dev*[.*pci_func*][=pci:*pci_spec*]

A PCI device is made available to the guest and placed at the guest PCI bus *pci_bus*, PCI device *pci_dev*, and PCI function *pci_func* if this field is given.

If no equals sign follows, the device being passed through is host *pci_bus*,*pci_dev*,*pci_func*. If there is an equals sign followed by another `pci:` prefix, then the *pci_spec* after this prefix is either *pci_vid*/*pci_did* or *pci_bus*:*pci_dev*[.*pci_func*], and it specifies the host vendor/device ID or the host bus/device/function to be passed through.

For example:

```
pass loc pci:0:12.0=pci:0x8086/0x1234
```

passes a device with vendor ID `0x8086` and device ID `0x1234` to the guest at bus `0`, device `12`, and function `0` (multi-function). At present, you may choose the device and function numbers in the guest, but the bus number must be `0`.

> **NOTE:**
> To support pass-through PCI devices, you must include the **pci_server-qvm_support.so** module in your hypervisor host image. For instructions on doing so, see the pci-server reference in the QNX OS *Utilities Reference*.

Per the PCI specification, some multi-function device must implement function `0`. If you want to make your PCI devices enumerable with non-zero function numbers, you can use the vdev-pci-dummy instance as a placeholder at function `0`.

See "Graphics device" below for a configuration example that uses this location type.

sched *priority*

Set the priority of the pulses the host system uses to inform the `qvm` process instance of the pass-through interrupts for the current pass-through device (i.e., the one specified after this option in the current `pass` context).

For example, the following sets the pulse priority to 67 for interrupt 89 for the current device:

```
pass
    sched 67 intr gic:89
```

Similarly, the following sets the pulse priority to 44 for interrupts for the current device:

```
pass
    sched 44 loc pci:0:3.0
```

In this last example, no interrupt is specified because for PCI devices interrupts can be implicit (see "Common vdev options" in the "Virtual Device Reference" chapter).

For more information about scheduling priorities, see "Scheduling" in the "Understanding QNX Virtual Environments" chapter, and "Thread scheduling" in the QNX OS *System Architecture* guide.

Description:

The `pass` option specifies that one of the following should be passed through from the hypervisor host domain to the guest:

- a physical device, such as a PCI audio device
- a data blob, such as an initial RAM disk (**initrd**) for a Linux guest

Until the next `pass`, `vdev`, or `reserve` option is encountered in the configuration, all `intr` and `loc` options that follow a `pass` option specify interrupts and locations for this pass-through device or component.

> **CAUTION:**

In general, only one OS may have direct control over a physical hardware resource such as a device or region of memory. In a system that supports virtualization, this means either the host OS or a guest OS (but not both) and requires important considerations when passing physical devices or memory through to a guest.

If the host owns a device that a guest requires pass-through access to, the host must terminate its driver for the device before the guest can start a driver for the device in its virtual environment. Similarly, if one guest owns a device as a pass-through device, it must terminate the device driver in its virtualized space before another guest can use the device in its space.

In short, you should never pass a DMA device through to more that one guest, and only in exceptional designs should you pass a non-DMA device through to more than one guest. If you believe that your design requires the use of a non-DMA device in this way, contact your QNX representative.

The same restriction applies to RAM. When you pass through host-physical memory to a guest, the host must have been configured, via options passed to the startup bootstrap program, to not allocate from that same memory region. Otherwise, it might try to use this memory and, thus, the host and guest OSs might end up corrupting each other's memory, leading to bad behavior or system crashes.

The `qvm` process recognizes the following data types (*blob_type*) specified before a `pass` option:

acpi

```
acpi pass
```

Only available for x86. Any `loc` options following will be interpreted as providing additional ACPI tables to the guest system.

data

Recognized, but not relevant for pass-through.

fdt

```
fdt pass
```

Any `loc` options following will be interpreted as providing an FDT binary blob to the guest system (see "ACPI tables and FDTs" in the "Configuration" chapter).

guest

```
guest pass
```

Any `loc` options following will be interpreted as providing a guest OS boot image to the guest system. For example, if you use this option, you can have the bootloader pre-load a Linux kernel image to RAM at the location specified by `loc`, then point the `qvm` process instance to this location rather than to a file to start its guest.

initrd

```
initrd pass
```

Any `loc` options following will be interpreted as providing an initial RAM disk (**initrd**) image to the Linux guest system.

raw

Recognized, but not relevant for pass-through.

For general information about pass-through devices, see "Pass-through devices" in the "Understanding QNX Virtual Environments" chapter.

NOTE:
On ARM platforms, devices that are clock-dependent (e.g., eMMC devices) can't be passed through (see "Passing through clock-dependent devices" in the same chapter).

DMA status of pass-through devices

When you are implementing a QNX Hypervisor system, you should specify the DMA status of any pass-through device you add to your system; that is, include either `d` (DMA device) or `n` (not DMA device) in the access type following the *length* argument. For example, this configuration:

```
pass loc mem:0x2000,0x1000,rn=0x1000
```

specifies that the device is not a DMA device.

If you're using `smmuman` for DMA configuration checking (which is the default assumption), and you specify neither `d` nor `n` or you specify both for the access type, the `qvm` process instance assembling the VM prints a fatal error message and aborts without starting the VM.

CAUTION:
A DMA device may not be used by more than one OS, whether it's multiple guests or the host and a guest. For example, if you configure a VM to give a guest access to a DMA device that has been already allocated to another VM, when the newly configured VM attempts to access the device, this will either succeed or fail depending on whether an external manager (e.g., `smmuman`) is controlling access to the device. If the access fails, `qvm` aborts and hence, the guest terminates immediately. Therefore, it's important to manage system-wide contention for devices independently of each VM.

Examples:

Linux RAM disk image

The following provides an **initrd** RAM disk image to a Linux system from host system memory:

```
initrd pass loc 0xB1234000,0x4000000,rc=0xB7654000
```

The above creates a RAM disk of `0x4000000` bytes located in the host-physical memory at address `0xB1234000`, accessible by the guest at address `0xB7654000` in guest-physical memory. This region is cacheable (`c`) and read-only (`r`).

NOTE:
It is your responsibility to locate the data in the guest system in accordance with the guest OS's rules. You may prefer to use the `load` option to pass a Linux **initrd** RAM disk to a guest. For example: `initrd load ./myinitrd.gz` (see "load" in this chapter).

Audio device

The following passes through the audio-related devices:

```
pass pci:0:14.0 pass pci:0:23.0
```

Since these are PCI devices, the configuration uses BDF (bus/device/function) numbers rather than memory addresses. The example assumes a board with the specified physical devices at the specified locations.

Graphics device

Assuming that you have a `screen` driver and the other components you need to run graphics on the board, you could do the following to pass through a graphics device:

1. Modify your VM configuration file to create a PCI pass-through device:

   ```
   pass loc pci:0:2.0=pci:0:2.0
   vdev pci-dummy
       clone pci:0:31.0
   ```

2. Start the guest with the new VM configuration.
3. From your desktop host, start Screen in the guest, using the `-c` option to point `screen` to the DRM manager:

   ```
   screen -c /usr/lib/graphics/intel-drm/graphics.conf
   ```

In step 1 we use the `0:2.0` BDF location, which Intel has designated for integrated graphics:

- The value to the left of the equals sign passes the host BDF through to the guest.
- The value to the right of the equals sign ensures that the device will appear at the specified location on the guest's PCI bus.

We specify the BDF value in both places to ensure that the `qvm` process instance presents the same location (`0:2.0`) to the guest, in case code in the guest assumes this is an Intel-designated BDF.

The `pci-dummy` vdev is not strictly required. However, the device driver usually uses this vdev to identify the display adapter, so using the `pci-dummy` vdev allows us to expose the correct vendor and device IDs without providing any further (and unneeded) functionality.

The example assumes a board with the specified physical devices at the specified locations.

See also `loc` above, and `pci-dummy` in the “Virtual Device Reference” chapter.

Page updated: August 11, 2025

ram

Allocate RAM to the VM

Synopsis:

```
ram [start_address,]length
```

Description:

The `ram` option allocates RAM to the VM, at the guest-physical address specified by *start_address* for the number of bytes specified by *length*.

If you don't specify this option, by default the `qvm` process allocates the RAM immediately following the highest RAM/ROM address already allocated in the guest system.

When you specify the *length* argument, you can use either hexadecimal notation (e.g., `0xa0000` for 640 KB), or human-readable abbreviations (e.g., `16M` for 16 MB).

NOTE:
You must allocate RAM before components that refer to the guest memory (see "Exceptions" in the "Configuration" chapter).

RAM that appears to the guest as contiguous is unlikely to be contiguous in host-physical memory (see "Memory" in the Understanding QNX Virtual Environments) chapter.

Example:

The following configuration allocates 128 MB of RAM to the VM, starting at address `0x80000000`:

```
ram 0x80000000,128M
```

The start address for the memory allocation is the address inside the VM: the guest-physical address. If this RAM location is to be used for the guest's bootable image, the guest must be configured to look for the image at this address inside its memory. For instance, assuming the example above is for a QNX guest, you must specify this location in its buildfile:

```
[image=0x80000000]
[virtual=aarch64le,elf] .bootstrap = {
   [+keeplinked] startup-armv8_fm -v -H
   [+keeplinked] PATH=/proc/boot procnto -v
}
```

Allocating memory on x86 boards

Due to the long history of the x86 platform, it is likely that a guest OS will expect the 128 KB of VGA memory to be mapped at `0xa0000`. Though the hypervisor doesn't offer emulation of a VGA device, you should reserve this region in guest-physical memory so that the `qvm` process assembling the VM can't allocate it (see `reserve` in this chapter).

Similarly, the region between `0xc0000` and `0xfffff` has traditionally been known as the BIOS area. You should configure this area, but you should use the `rom` option to remove it from the memory map passed on to the guest.

Thus, to properly virtualize x86 hardware, instead of specifying memory allocations like this:

```
ram 1024M
```

you must reserve the location for the VGA, specify the ROM for the BIOS, and specify the RAM memory, so that your memory configuration for an x86 guest looks something like this:

```
ram 0,0xa0000
reserve loc 0xa0000,0x20000
rom 0xc0000,0x40000
ram 1m,1023m
```

where:

- `ram 0,0xa0000` allocates the first 640 KB of guest-physical RAM to the guest
- `reserve loc 0xa0000,0x20000` reserves the next 128 KB so that the `qvm` process instance assembling the VM can't use this location when it is dynamically allocating resources (see `reserve` in this

chapter).

- `rom 0xc0000,0x40000` allocates the 128 KB BIOS area as ROM
- `ram 1m,1023m` allocates to the guest the remaining portion of the 1 GB of RAM, starting at 1 MB

For cases where the guest doesn't expect to find a component (e.g., legacy device, BIOS) at a specific location, you may specify only the size of the memory allocation and let the `qvm` process decide on the location. However, if the guest expects memory to be available for a specific purpose at a specific location, you must respect these requirements, just as you would for an OS running directly on an x86 board.

See "Configuring memory" in the "Understanding QNX Virtual Environments" chapter.

Mapping PCI pass-through devices on x86

By default, PCI devices on x86 boards are mapped from 2G to 4G in guest-physical memory. Therefore, for guests that require large amounts of memory and that use PCI, avoid allocating this memory range. For example, the following configuration allocates 4.5G of memory for the guest while avoiding the 2G to 4G range:

```
ram 0,0xa0000
rom 0xc0000,0x40000
ram 1M,1500M # Get 1.5 GB. Stops below the 2G boundary.
ram 4096M,3000M # Get 3 GB. Starts above the 4G boundary.
```

Page updated: August 11, 2025

reserve

Reserve specified locations in guest-physical memory and/or guest interrupts

Synopsis:

```
reserve options
```

Options:

`intr` *number*

Prevent the `qvm` process from allocating the guest interrupt specified by *number*.

`loc` *location_spec*

Prevent the `qvm` process from allocating the location specified by *location_spec*. The format of this argument depends on the type of resource that exists in that location:

- `mem:`*`region`*`,`*`length`* The reserved location is a memory area starting at *region* and lasting for *length* bytes.
- `io:`*`port`*`[,`*`length`*`]` The location is an x86 I/O region. The first argument is a port number and this is optionally followed by a length in bytes.
- `pci:`*`pci_bus`*`:`*`pci_dev`*`[.`*`pci_func`*`]` The location is a PCI location. For `reserve loc`, you must use the BDF specification form, and you may not specify a length.

For more information about locations, see "Guest resource types" in the "Configuration" chapter.

Description:

The `reserve` option reserves locations in guest-physical memory and/or guest interrupts to prevent the `qvm` process from allocating them to any component or device.

If a guest OS expects that specific locations will hold some predetermined artifact (such as a table), or that specific interrupts will be used for some predetermined purposes, you should use the `reserve` option to ensure that when the `qvm` process assembles the VM for the guest it doesn't inadvertently allocate these locations and interrupts, and leaves them for you to allocate as your guest requires.

CAUTION:
The `reserve` option prevents the `qvm` process from dynamically allocating the locations and interrupts specified, but it doesn't prevent the `qvm` process from allocating these locations and interrupts if they are explicitly specified in the VM configuration.

The following examples illustrate how this option can be used to reserve different types of resources:

- Guest interrupt 39:

  ```
  reserve intr 39
  ```

- PCI device 1 on bus 0:

  ```
  reserve loc pci:0:1
  ```

- An x86 I/O region of eight bytes at `0xcf`:

  ```
  reserve loc io 0xcf,0x8
  ```

- A memory region of eight bytes at `0x1000`:

  ```
  reserve loc 0x1000,0x8
  ```

Page updated: August 11, 2025

rom

Allocate ROM to the VM

Synopsis:

```
rom [start_address,]length
```

Description:

The `rom` component allocates ROM to the VM, at the guest-physical address specified by *start_address* for the number of bytes specified by *length*.

This option is the same as the `ram` option, except that the guest isn't allowed to write to this memory. Typically, you would use the `load` option to place some initialized data into the region specified by `rom` before the guest starts.

If you don't specify this option, by default the `qvm` process allocates the ROM immediately following the highest RAM/ROM address already allocated in the system.

When you specify the *length* argument, you can use either hexadecimal notation (e.g., `0xa0000` for 640 KB), or human-readable abbreviations (e.g., `1M` for 1 MB).

NOTE:
You must allocate ROM before components that refer to the guest memory (see "Exceptions" in the "Configuration" chapter).

ROM that appears to the guest as contiguous is unlikely to be contiguous in host-physical memory (see "Memory" in the "Understanding QNX Virtual Environments" chapter).

Example:

The following configuration allocates 256 KB of ROM to the VM, starting at address `0xc0000`:

```
rom 0xc0000,0x40000
```

The start address for the memory allocation is the address inside the VM: the guest-physical address (intermediate physical address).

Page updated: August 11, 2025

set

Set the value of a configuration variable

Synopsis:

```
set var val
```

Options:

The `set` option supports the following variable types:

- address – use any of the forms permitted for memory sizes when you specify *val* (e.g., 4K, 0d16384); for more information, see "About notation"
- boolean – any of the following values are equivalent; the first term turns on the feature, the second turns it off: `on`/`off`, `1`/`0`, `yes`/`no`, `true`/`false`
- tristate – the following values are permitted:
  - `on` – the feature is always on
  - `off` – the feature is always off
  - `auto` – the feature's setting is automatically detected by `qvm`
- number – any positive integer that can be expressed with 64 bits; decimal notation (no prefix) and hexadecimal notation (prefixed with `0x`) are permitted; for more information, see "About notation"
- string – a text string, used as is

NOTE:
For descriptions of all variables (i.e., *var* values) supported on a given platform, including the variable types (i.e., possible *val* values), see "Configuration variables".

Description:

Each `set` *var val* argument pair defines a variable setting that applies to an *implicit context*. This is different than the VM configuration file notion of a context, which is a group of related configuration options (for details, see "Contexts" in the "VM configuration syntax" topic). The implicit context to which each variable applies is noted in its description. Supported implicit contexts include:

- global – applies to the VM (`qvm` process instance) currently being configured
- CPU – applies to the vCPU currently being configured
- vdev – applies to the vdev currently being configured

You can display information on the currently permitted variables. For example, in a shell prompt in the host, type:

```
# qvm set ?
```

You should see the variable names along with their types and contexts in a listing like the following:

```
allowed set variables
    legacy-free                    (boolean, global)
    message-block-timeout          (number, global)
    posted-interrupts              (boolean, global)
    virtual-interrupts             (boolean, global)
```

A question mark (`?`) is a shell wildcard character, so you may need to escape it. If a vdev defines its own variables, `?` will list these only after a `vdev` option has loaded the vdev where these variables are specified.

Page updated: August 11, 2025

suppress

Explicitly request the suppression or generation of the specified system information table type in the guest

Synopsis:

```
[system_table_type] suppress|generate
```

Description:

By default a QNX hypervisor always generates an FDT when it runs on an ARM platform, and ACPI information when it runs on an x86 platform. It doesn't currently support ACPI on ARM architectures or FDTs on x86 architectures, so the `generate` property is for future use only.

The *system_table_type* argument specifies the type of system information table to suppress or generate; this can be either `fdt` (ARM) or `acpi` (x86).

Examples:

The following suppresses the FDT system information table in an ARM guest:

```
fdt suppress
```

The following suppresses the ACPI system information table in an x86 guest:

```
acpi suppress
```

Page updated: August 11, 2025

system

Specify the name of the VM being configured by the present configuration file

Synopsis:

```
system system_name
```

Description:

The `system` option specifies the name (*system_name*) of the VM being configured by the present `qvm` configuration file or command-line startup. If specified, it must be the first entry (other than comments) in a `qvm` configuration file or command-line string.

NOTE:
For each VM, the VM-specific configuration file (**guest*.qvmconf**) appears before the common configuration file. The VM-specific configuration files are likely to start with the `system` option, which if it appears must be the first option specified for a VM configuration.

Example:

The following specifies that the name of the VM created with the current `qvm` configuration file is “qnx8a”:

```
system qnx8a
```

If you use the `virtio-net` vdev's `peer` option, the full name of the end of the peer link is formed from the value of *system_name* concatenated with the value specified for the `virtio-net` vdev's `name` option (see vdev virtio-net in the “Virtual Device Reference” chapter).

Page updated: August 11, 2025

`tolerance`

Set how long the host system is allowed to delay notifying the hypervisor of a timer expiry

Synopsis:

```
tolerance microseconds
```

Option:

microseconds

The emulated timer device tolerance, in microseconds. The default is 10 percent of the timer tick size on the host system.

Description:

The `tolerance` option sets the maximum amount of time, in microseconds, that the host system is allowed to delay before notifying the hypervisor of a timer expiry. If the `tolerance` option isn't specified, the tolerance defaults to 10 percent of the host system's timer tick size (which defaults to 1 ms, making the default tolerance 100 µs).

The hypervisor uses host system timers to generate notifications when a guest timer interrupt needs to be delivered in order to support guest system virtualized timer hardware (e.g., the hpet vdev on x86).

The reason that there may be a slight delay between timer expiry and notification delivery is that a QNX hypervisor system allows the system designer to choose the tradeoff between timer accuracy and power consumption (CPU load).

Setting tolerance values smaller than the tick size lets you improve the guest's high-resolution timer (HRT) accuracy, which can be specified to the nanosecond level, because the QNX hypervisor host is always tickless. If you do use smaller values, however, this improves accuracy but also increases the interrupt load and thus may decrease overall performance. This is because it is less likely that multiple timers can be coalesced (aligned to fire) in the same interrupt. You must therefore be sure that any guest expectations assumed in the design of the guest BSP, OS, and applications align with what is being provided by the timer setup in the underlying VM and the host machine. For example, the guest should not expect or require a greater timer accuracy than what either of these components can provide.

For more information, refer to “High-resolution timers” in the “Understanding the Microkernel's Concept of Time” chapter in the QNX OS *Programmer's Guide*.

Page updated: August 11, 2025

unsupported

Specify what the hypervisor should do if it can't successfully complete the emulation of a guest instruction

Synopsis:

```
unsupported   instruction | reference | register   abort | fail |
ignore
```

Description:

Only one type-action pair can be used per `unsupported` option instance. Specify the option for each unsupported type you want to configure.

Types:

instruction
: The hypervisor doesn't support emulation of the instruction. If unspecified, this defaults to `fail`.

reference
: The instruction references a memory (or I/O port on x86) location to which the guest doesn't have access. If unspecified, this defaults to `ignore`.

register
: The guest is attempting to use a system register (ARM) or MSR (x86) that the hypervisor doesn't support. If unspecified, this defaults to `fail`.

Actions:

abort
: Display a message indicating the failure and guest state, then terminate the `qvm` process instance hosting the offending guest.

fail
: Deliver an appropriate CPU exception to the offending guest.

ignore
: Treat the instruction as a no-op, except in the case of guest attempts to read from an unsupported memory or I/O port location, which sets the destination of the instruction to all one bits (`~0`).

If the hypervisor encounters a condition of a type for which no action has been specified, the hypervisor uses its default action for that type.

Example:

The following configuration specifies that:

- if the guest hosted by the current VM attempts to use a register (ARM) or MSR (x86) that the hypervisor doesn't support, the hypervisor will terminate the `qvm` process hosting the guest
- if a guest instruction attempts to access a memory (or I/O port on x86) location, the hypervisor will deliver an appropriate CPU exception to the guest

```
unsupported register abort
unsupported reference fail
```

NOTE:
The configuration doesn't specify the `instruction` type, so the hypervisor will use its default configuration, which for this type is `fail`.

Page updated: August 11, 2025

user

Assign the specified values to the user name, or to the user ID and group IDs of the `qvm` process instance

Synopsis:

```
user username[,gid]* | uid[:gid[,gid]*]
```

Options:

gid

One or more group IDs (separated by commas) to assign to the `qvm` process instance.

uid

The user ID to assign to the `qvm` process instance.

username

The user name to assign to the `qvm` process instance.

Description:

The user is a user in the *hypervisor host*, not in the guest that the VM being configured will host.

If you use the `username[,gid]*` form, the primary group is the one specified for *username* in **/etc/passwd**.

Page updated: August 11, 2025

Configuration variables

You can set configuration variables that affect `qvm` process instance, guest, CPU, and vdev behavior.

Configuration variables are specified using the `set` option, as follows:

```
set var val
```

Depending on the variable, its setting can apply to one of the implicit contexts listed above. For more information about the syntax for setting variables and about the implicit contexts to which they can apply, see the `set` option reference. Here, we describe all supported variables for each platform.

NOTE:
Unless otherwise indicated, each variable can be defined anywhere in the VM configuration file.

Common variables

The following variables are supported on both ARM and x86 platforms:

counter-offset

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: `0xffffffffffffffff`

Specify the difference between host and guest timestamp counter values. This difference is constant throughout the VM's lifecycle. The default behavior, which can also be obtained by setting *counter-offset* to a 64-bit value of all ones (i.e., `0xffffffffffffffff`) is to make the counter of the VM start at 0. Setting this variable to 0 makes the host and guest values match. Setting it to another number allows you to both easily match and distinguish between host and guest values.

debug

- Context: vdev – applies to the vdev currently being configured
- Variable type: boolean
- Default: on

Control whether a vdev is included in the VM status info produced by SIGUSR1.

exit-on-halt

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: on (guest exits on WFI or HLT instruction)

Control if a WFI (on ARM) or HLT (on x86) instruction causes a guest exit. Setting *exit-on-halt* to `off` increases physical CPU usage by vCPU threads but reduces interrupt latency. When the guest is allowed to execute WFI or HLT instructions, the host can't schedule any other thread on the physical CPUs used by the guest while it's executing those instructions.

fdt-dump-file

- Context: global – applies to the entire `qvm` process instance
- Variable type: string
- Default: none

Instruct the `qvm` process instance to write the VM's FDT to the specified file. You can then examine the FDT to better understand any customizations you may need to make to it (see "FDTs (ARM)" in the "Configuration" chapter).

If you set this variable and use the `dryrun` option, you can get the guest's FDT without starting the guest. For example, starting a `qvm` process instance as follows:

```
qvm @mainline-guest.qvmconf set fdt-dump-file /tmp/fdt.dtb dryrun
```

would cause the process instance to write the guest's FDT to **/tmp/fdt.dtb**, then exit with a message like the following:

```
FDT saved to '/tmp/fdt.dtb'
Exiting: dryrun completed
```

Note that the `dryrun` option causes the `qvm` process instance to exit before starting the guest (see dryrun in this chapter).

message-block-timeout

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: 10000 milliseconds (10 seconds)

Set the maximum allowed time, in milliseconds, that a message from the `qvm` process may be blocked before the service sends an unblock pulse to the receiving server. For example:

```
set message-block-timeout 200
```

configures the `qvm` process to send an unblock pulse to any server that doesn't respond to a message within 200 milliseconds.

The *message-block-timeout* variable must be a value from 5 through 10000, or 0 (zero). A 0 makes the timeout infinite (never time out).

Depending on the server's response (or non-response), the `qvm` service may terminate with an error. You can use `server-monitor` to handle situations where a server doesn't respond to an unblock pulse (see `server-monitor` in the QNX OS *Utilities Reference*).

msi-pass-initial-threads

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: 0

Pre-create the specified number of threads for Message Signaled Interrupt (MSI) forwarding. By default, new threads are created for this purpose only when there are no idle MSI forwarding threads that can be used.

slog-buffer

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: 4K

Override the default slog buffer size. Setting this variable lets you increase the buffer size to prevent older entries from being dropped to make room for new entries. For example: `set slog-buffer 8K` increases the buffer size to 8K.

Note that:

- The buffer size must be a multiple of 4K (e.g., 8K, 12K).
- This option must be specified *before* the `logger` option in the VM configuration.

smmu-externally-managed

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: off (the `smmuman` service is expected to be running)

Indicate whether IOMMU/SMMU services are externally managed. By default, the `qvm` process instance expects to use `smmuman` to ensure that the DMA configuration is safe and secure for pass-through devices. Setting this variable to `on` tells `qvm` to trust that the DMA configuration has been taken care some other way.

vdev-message-block-timeout

- Context: vdev – applies to the vdev currently being configured
- Variable type: number
- Default: 10000 milliseconds (10 seconds)

The *vdev-message-block-timeout* variable behaves exactly like the *message-block-timeout* variable but applies only to the vdev currently being configured. This means that the variable must be set within a context beginning with a `vdev` option; for example:

```
vdev lithium-ion-battery
    set vdev-message-block-timeout 5
```

ARM variables

The following variables are supported on ARM platforms only:

fdt-cmdline-policy

- Context: global – applies to the entire `qvm` process instance

- Variable type: string
- Default: `overlay`

Control how the value from the `cmdline` option and the value from the user-provided FDT overlay get merged into the `/chosen/bootargs` property given to the guest. The `overlay` setting lets the user-provided overlay (if it's present) replace this property. This is the default behavior. Alternatively, `prepend` prepends the value from the user-provided overlay to the value given in the `cmdline` option, while `append` appends the value from the user-provided overlay to the value of this option.

force-erratum-l2-tlb-prefetch

- Context: global – applies to the entire `qvm` process instance
- Variable type: tristate
- Default: `auto`

Specify whether to override the autodetection and correction of hardware errata described in:

- SDEN-885749 (Cortex A76) as #1262888
- SDEN-1152370 (Cortex A77) as #1262841

The default behavior (`auto`) is for `qvm` to apply these errata only if it auto-detects that it is needed on the hardware expected by the guest OS (i.e., the hardware that the VM is simulating).

You can set this variable also to `on` to always apply these errata or to `off` to never apply them.

force-erratum-speculative-at

- Context: global – applies to the entire `qvm` process instance
- Variable type: tristate
- Default: `auto`

Specify whether to override the autodetection and correction of hardware errata described in:

- ARM-EPM-048406 (Cortex A53) as #1530924
- SDEN-859338 (Cortex A55) as #1530923
- ARM-EPM-049219 (Cortex A57) as #1319537
- ARM-EPM-012079 (Cortex A72) as #1319367
- SDEN-885749 (Cortex A76) as #1165522

The default behavior (`auto`) is for `qvm` to apply these errata only if it auto-detects that it is needed on the hardware expected by the guest OS (i.e., the hardware that the VM is simulating).

You can set this variable also to `on` to always apply these errata or to `off` to never apply them.

gic-hwassist

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: on

If *gic-hwassist* is set to "true" (default), the `qvm` process instance will try to use the GIC hardware assistance support. If this variable is set to "false", the `qvm` process instance won't try to use this support. You should disable this setting only when advised to by QNX customer support to help diagnose a problem.

host-paddr-gicd

- Context: global – applies to the entire `qvm` process instance
- Variable type: address
- Default: the `gicd` entry in the host's system page *asinfo* section

Set the host-physical address of the GIC distributor registers. This setting is needed on boards where the startup program does not automatically supply the address via the system page.

NOTE:
This and the other *host-paddr-** options are needed only if the startup program has incomplete or incorrect information in the *asinfo* section of the system page. They are not needed on a standard platform with a correct startup program.

host-paddr-gich

- Context: global – applies to the entire `qvm` process instance
- Variable type: string

- Default: the `gich[.cpunum]` entry in the host's system page *asinfo* section

Set the host-physical addresses of the GIC hypervisor control registers. The string should be formed as follows:

host_address`[/`*cpu*`/`*spacing*`]{,`*host_address*`[/`*cpu*`/`*spacing*`]}`

The string components are:

host_address
: The host-physical address of the current physical CPU's GICH registers.

cpu
: If present, the number of the physical CPU whose registers are located at *host_address*.

spacing
: If present, the number of bytes to increase the offset to get the address of each subsequent physical CPU's registers.

host-paddr-gicr

- Context: global – applies to the entire `qvm` process instance
- Variable type: address
- Default: the `gicr` entry in the host's system page *asinfo* section

Set the total size needed for the GIC redistributor registers. Normally, `qvm` determines this size as a multiple of the number of cores on the board. However, some board designs that support varying numbers of cores do not arrange the redistributor registers to form a single contiguous region for all CPUs on the system. For example, a board design that could handle up to 128 cores might organize the redistributors in an array that fits up to 128 entries but the 32-core version could be spread out sparsely across that whole region.

host-paddr-gicv

- Context: global – applies to the entire `qvm` process instance
- Variable type: address
- Default: the `gicv` entry in the host's system page *asinfo* section

Set the host-physical address of the GIC virtual CPU interface registers.

host-ppi-gic-hwassist

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: 25

Set the host interrupt number for the GIC hardware assist maintenance interrupt.

NOTE:
This and the other *host-ppi-** options are needed only if your system does not follow the ARM guidelines for assigning PPIs as explained in rule B_PPI_02 from the *ARM Base System Architecture 1.1D (DEN0094)*.

host-ppi-hcnt

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: 26

Set the host interrupt number for the hypervisor counter interrupt.

host-ppi-pmu

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: 23

Set the host interrupt number for the PMU interrupt.

host-ppi-vcnt

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: 27

Set the host interrupt number for the virtual counter hardware.

its-client-config

- Context: global – applies to the entire `qvm` process instance
- Variable type: string
- Default: none

Define the Interrupt Translation Service (ITS) domains available to the guest. The *its-client-config* variable is a string that must contain a colon-separated list of ITS domain definitions as follows: *`baseloc1,lockloc1[:baseloc2,lockloc2]*`*

NOTE:
Currently, only a single domain definition is allowed.

linux-fdt-free

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: off (the FDT must be located within a specific region of the guest memory)

Control whether to allow the FDT provided to the guest to be located anywhere in guest memory. By default, the FDT must be located in the memory region within 512M of the OS image as required prior to Linux 4.2.

pointer-authentication

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: on (FEAT_PAUTH is enabled)

Specify whether the ARM extension FEAT_PAUTH is enabled in the guest. Note that for this feature to take effect on a vCPU, all pCPUs on which it may run must support this feature.

psci-supported

- Context: global – applies to the entire `qvm` process instance
- Variable type: tristate
- Default: `auto`

Specify whether the host supports the Power State Coordination Interface (PSCI). With the default `auto` setting, `qvm` automatically determines whether the host supports PSCI by checking the host's device tree. However, if the host does support PSCI but does not provide an appropriate device tree entry, this option can be set to `on` to prevent the checking. This option can be set also to `off` to disable the use of PSCI altogether.

sve-enable

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: off (FEAT_SVE is disabled)

Specify whether the ARM extension FEAT_SVE is enabled in the guest. Note that for this feature to take effect on a vCPU, all pCPUs on which it may run must support this feature.

trace-spectre-workaround

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: off

Some optimizations to guest exit mechanisms cause some guest spectre errata workaround requests to not appear in a host trace log. Setting *trace-spectre-workaround* to “true” disables the optimizations, allowing the omitted guest trace-spectre-workaround requests to appear in the host trace log.

trace-vtimer

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: off

Some optimizations to the guest exit mechanisms will cause some guest virtual timer interrupt deliveries to not appear in a host trace log. Setting *trace-vtimer* to “true” disables the optimizations, allowing the omitted guest virtual timer interrupt deliveries to appear in the host trace log.

trace-wfe

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: off

Some optimizations to the guest exit mechanisms will cause some guest executions of a WFE instruction to not appear in a host trace log. Setting *trace-wfe* to "true" disables the optimizations, allowing the omitted guest executions of a WFE instruction to appear in the host trace log.

vmid

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: auto-generated value

Set the VMID for the guest. The value can't be 0. If this variable isn't set, the hypervisor generates a VMID. A vdev can retrieve the value via the *qvm_guest_vmid()* vdev API function.

x86 variables

The following variables are supported on x86 platforms only:

legacy-free

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: off – don't permit arbitrary memory layout (i.e., require ACPI tables, etc.)

The *legacy-free* variable can be used to assemble a VM that hosts an x86 guest with an arbitrary memory layout. Most notably, with *legacy-free* set to "on", the VM can be assembled for a guest that doesn't require memory in the BIOS area (where the ACPI and SMBIOS tables are stored).

If you set this variable to "on", you should also use the `suppress` variable to suppress generation of the ACPI system information table for the x86 guest (see suppress in this chapter).

posted-interrupts

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: on

If *posted-interrupts* is set to "on" (default), the `qvm` process instance will try to use the hardware's posted interrupt support, if such support is available. If this variable is set to "off", the process instance won't try to use this support.

virtual-interrupts

- Context: global – applies to the entire `qvm` process instance
- Variable type: boolean
- Default: on

If *virtual-interrupts* is set to "on" (default), the `qvm` process instance will try to use the hardware's virtual interrupt support, if it is available. If this variable is set to "off", the process instance won't try to use this support. Note that turning virtual interrupt support off also turns off posted interrupt support.

vmx-abort-interval

- Context: global – applies to the entire `qvm` process instance
- Variable type: number
- Default: 0 ms (off)

Periodically check each vCPU to ensure the VMX abort indicator has not been set. A non-zero *vmx-abort-interval* value specifies the interval in milliseconds at which these checks are done. The default setting of zero (0) means these checks are turned off.

NOTE:
This variable should be used only as requested by QNX Customer Support.

vdev

Specify and configure a virtual device to include in the VM

Synopsis:

```
vdev name [options]*
```

Options:

The supported options vary with the individual vdev being configured. More information is given below.

Description:

Provide the virtual device *name* to the guest system that will run in the VM being configured. If *name* is not already known to the `qvm` process instance assembling the VM, `qvm` searchs for a shared library in the host system paths within *LD_LIBRARY_PATH* to load. If necessary, `qvm` adds the **vdev-** prefix or the **.so** suffix to the library filename if either is not present in the *name* argument.

There are many options you can provide following the device name. For example, the `loc` and `intr` options provide the location and interrupt for the device to use, while the `log` option specifies the message types to output and the output destination.

For full details about the different supported vdevs and all of the options you can set for each, refer to the "Virtual Device Reference" chapter.

Page updated: August 11, 2025

VM Configuration Reference

When you specify options to configure a `qvm` process, you are assembling and configuring the components of a virtual machine (VM) in which a guest will run.

NOTE:

- For general information about configuring VMs, including syntax, see "Assembling and configuring VMs" in the "Configuration" chapter.
- For descriptions of vdev configurations, see the "Virtual Device Reference" chapter.
- For information about configuring pass-through devices, see `pass` in this chapter.
- For descriptions of the configuration variables that you can set, see "Configuration variables" in this chapter.

Page updated: August 11, 2025