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

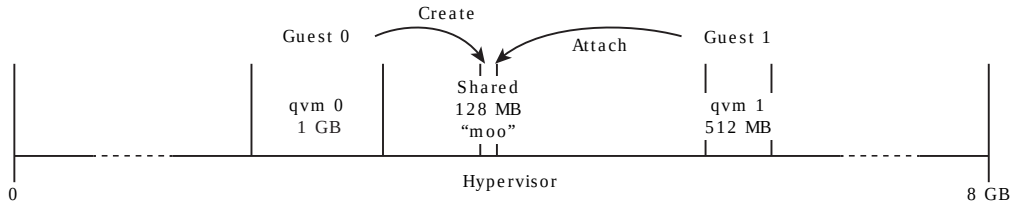
To use shared memory, a client application in a guest or in the hypervisor host needs:

- 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).

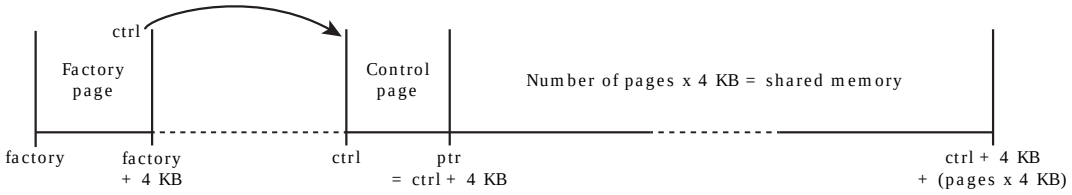
Control pages

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