CS695 (Spring 2024-25)

virtio inside-out

1. Introduction

The VirtIO Specification defines a communication framework that enables the creation of a direct communication channel between a virtual device in a VM and a corresponding physical device on the host system.

A functional VirtIO device must have two logical components:
Backend VirtIO Device: Implemented in the VMM (i.e. QEMU) in Host user-space.
- Passes the I/O request received from Front-End Driver to the Host Device and returns the results back to the Guest.
- The QEMU implementation of the VirtIO Standard enables developers to define emulations (virtual backends) using functions defined in <qemu/virtio.c>, <qemu/virtio-pci.c>, <qemu/virtio-rng.c>

Front-End VirtIO Driver: A Kernel module loaded into the guest OS.
- Accepts I/O requests from guest user-space and passes them to the Backend VirtIO device. Finally, it retrieves the results of the request and passes them back to user-space.
- The Linux implementation of the Virtio Standard enables developers to create drivers for paravirtualized devices using the Virtio API, defined in <linux/virtio.c>, <linux/virtio-rng.c> in kernel source.

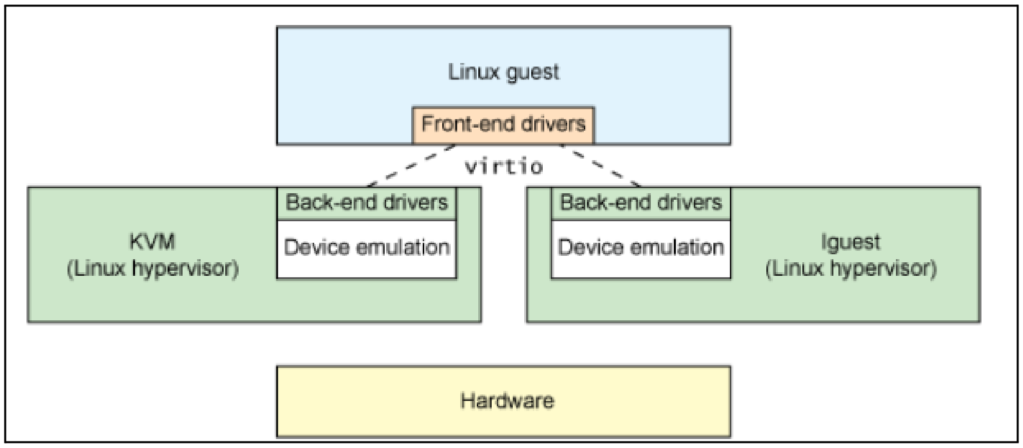


Fig 1. Logical Components of a VirtIO-based IO-virtualization solution.[1]

Central to the VirtIO Standard is the notion of a shared, managed memory region between a guest and QEMU, represented by a data structure called the **VirtQueue.**
- The Linux Kernel and QEMU implement APIs for the management and updation of these queues to implement the driver-device dataplane.

[1] IBM VirtIO Reference https://developer.ibm.com/articles/l-virtio/

2. Virtqueue Architecture

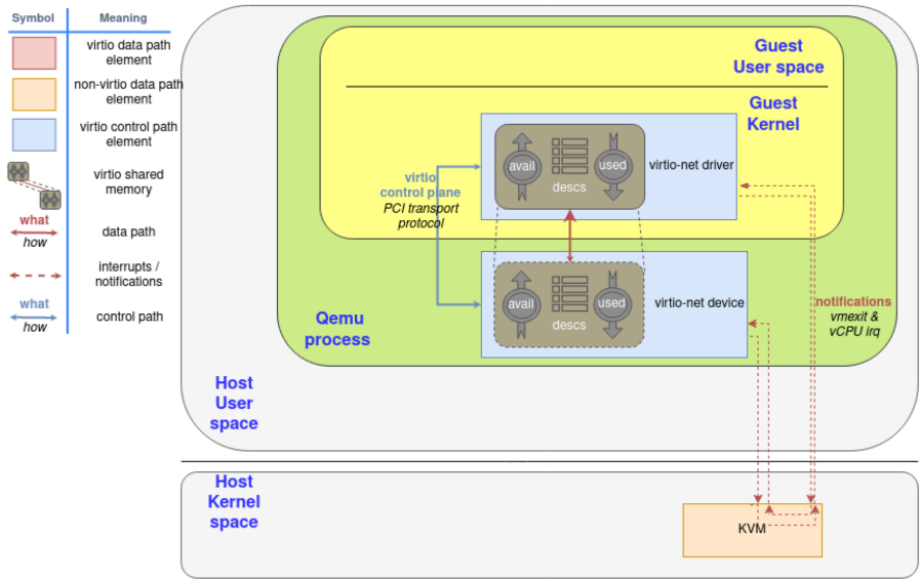
The **VirtQueue** is a nested data structure that implements a `VRing`, located in a shared page in guest physical memory, used to transfer data between device-driver pair i.e. between QEMU (a host user process) and driver (a guest kernel module/process). Many VirtIO devices initialize several VirtQueues. The virtqueues and their semantics are highly device-specific.

`VRing`s are of three kinds: Descriptor Ring (Descriptor Area), Available Ring (Driver Area), and Used Ring (Device Area). The entries in the descriptor ring point to actual data buffers. Entries in the available ring point to descriptor ring entries. Analogous to the available ring, used ring entries point to descriptors whose data buffers have been accessed (read/written) by the backend device.

Usage

The `VRing` s can be thought of as metadata structs with flags that enforce read/write permissions for the device and driver. The descriptor ring is fundamental as its entries contain references to the actual shared data buffers in guest physical memory. The Available and Used Rings are more involved in the metadata management.

- The front-end *driver* (in the guest kernel) is the only entity that can add (write) descriptor entries and available ring entries to the descriptor and available rings respectively.
- The backend device (in QEMU) is allowed to read from and write to the data buffer pointed to by a descriptor only if there is an entry in the available ring that points to that descriptor ring entry.
- Opposite to the available ring, only the backend device can configure and add entries to the used ring while the corresponding FE driver can only read from it.



To make a request to a device, the front-end driver will first create a descriptor entry, write to its buffer, add an entry to the available ring that references that descriptor entry, and finally notify the device of an update (i.e. new i/o request) via a virtqueue kick, which is simply a write to a pre-registered mmio region.

- On adding a new entry in the available ring, the backend device can begin processing the request by reading data written to the descriptor buffer by the driver. (Note that processing is done via host hardware, triggered by the path: QEMU VirtIO device backend → host kernel device driver → host device path).
- On completion of request processing, the backend-device will write to the descriptor buffer, and add an entry to the used ring that references the descriptor entry containing the device response, and notifies the guest through a virtual interrupt.

- The driver then follows the used ring entry and this copies back the newly-written device response back into the Guest kernel space (and subsequently into Guest User-Space).

The Linux and QEMU VirtIO APIs enable a developer to implement the above functionality. Writing your own VirtIO device for QEMU requires an understanding of the QEMU and Linux VirtIO APIs, the QEMU Object Model and the QEMU build system.

3. QEMU Build-System Architecture

"The QEMU build system has two stages; first the developer runs the `configure` script to determine the local build environment characteristics, then they run "make" to build the project"

In-Tree vs Out-of-Tree Build: The main difference is the location that the various generated/compiled QEMU files are stored once the build procedure has run successfully. Out-of-tree builds are recommended because they reduce clutter in the main QEMU source, and also allow for several flavours of QEMU to be built (i.e. with/without optional features, etc.)

- Create a new directory in the QEMU root directory, and call the QEMU `configure` script from there. Calling the `configure` script from the root directory will generate an in-tree build.

The configure script both detects your systems characteristics and also allow the developer to specify certain build options (i.e. configure your build). `./configure --help` prints a long list of options, but the most crucial one is setting the target-list which informs the QEMU build system which architectures must be emulated by the built QEMU executable.

- Note that a single QEMU executable can emulate a single architecture only. Therefore the target-list configuration effectively limits the number of executables that are generated by the build process.

The configure script also generates Makefile fragments (config-host.mk) etc that specify the GCC compiler to be used. Since the QEMU build procedure also uses Meson, a python-based build system comparable to CMake, the configuration step also creates a virtual Python environment for the Meson code. Meson is invoked to perform the actual configuration step for the emulator build.
The `./configure --help` option displays two kinds of targets, namely <arch>-linux-user and <arch>-softmmu, referring to user-mode and full-system emulation executables respectively.

- Note that QEMU can be built on platforms with KVM and without KVM. On non-hardware accelerated platforms, QEMU performs binary translation of every instruction encountered, via its TCG (Tiny Code Generator) subsystem. TCG ensures that even sensitive instructions are translated to appropriate host architecture instructions without a VM exit. When built with KVM-enabled configuration, the default VM-Exit to KVM path is taken.

User-Mode Emulation: In this mode, QEMU can launch processes compiled for one CPU on another CPU. The emulation is performed using TCG only. The user-mode emulator is of the type qemu-<arch>.
Full-System Emulation: QEMU's system emulation provides a virtual model of a machine (CPU, memory and emulated devices) to run a guest OS. It supports a number of hardware accelerators as well as a JIT through the Tiny Code Generator (TCG) capable of emulating many CPUs. The system emulator is of the type qemu-system-<arch>.

4. QEMU Execution Model

QEMU is invoked from the command-line with various options that define characteristics of the desired virtual machine. A minimal QEMU command-line would have options specifying Machine model, RAM, CPUs, NIC, Boot Disk, and possibly SSH port forwarding.

- Available options can be discovered through the `--help` suffix

```
$>../qemu/build-0/qemu-system-x86_64  \
 -enable-kvm \
 -smp 8  -m 4096 \
 -nic user,model=virtio,hostfwd=tcp::2222-:22  \
 -drive file=ubuntu.qcow2,media=disk,if=virtio \
 -monitor stdio
```

Fig 2. A minimal QEMU command-line invocation.

Upon successful invocation, QEMU runs as a user process on the host machine. The source code entry point is `qemu_init()` defined in <qemu/vl.c>. As part of initialization, QEMU parses the command line options and performs a set of actions, leading to the creation of a massive state object associated with the `MachineClass`. This machine state contains references to heap structs that represent emulated buses, virtual devices, and sets of `MemoryRegion` objects to represent various physical memory mappings visible to the guest. (ex. MMIO regions for PCIe devices).

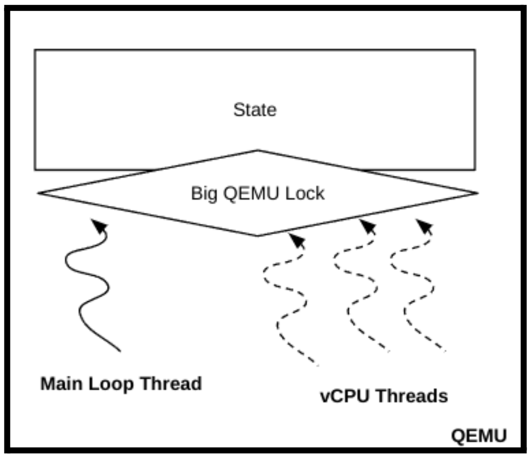


Fig 3. Components of a QEMU process (2016)[2]

Besides initializing MemoryRegions and their read/write callbacks, the initialization procedure also performs several KVM ioctl calls to register relevant machine state (MMIO address ranges, interrupt number mappings) with the KVM kernel module. This procedure terminates with the creation of `pthreads` that represent KVM vCPU threads.

- Each vCPU thread is blocked on the `ioctl(KVM_RUN)`while the CPU executes guest instructions.

At this stage, machine initialization is complete, and QEMU enters its main event loop. The QEMU Main Loop is generally blocked polling sets of file descriptors for events, and triggers pre-registered event callbacks. **The guest machine is now running.**

Depending on the current guest instruction and the QEMU-KVM initial configuration, a VM-Exit to KVM may lead either to:

[2] See blog by QEMU maintainer for the most recent (2024) design.
https://blog.vmsplice.net/2024/01/qemu-aiocontext-removal-and-how-it-was.html

a. Return from KVM_RUN with EXIT_REASON_MMIO or EXIT_REASON_IO to vCPU thread handler.
b. A signal to a file-descriptor monitored by the QEMU main loop, while the vCPU thread is still blocked on KVM_RUN. This mechanism is used to implement virtual interrupts and is referred to as the `irqfd/ioeventfd` mechanism.

5. QEMU VirtIO Devices

The QEMU hypervisor is responsible for providing a view of every device to the guest. QEMU comes with several virtual devices (accessible via `$QEMU --device help`) already implemented using specific QEMU APIs and semantics. *Therefore, for a new virtual device to be exposed to a QEMU guest, it is necessary to first ensure that the QEMU hypervisor itself is aware of this device.*
QEMU devices come in two parts:

a. **a backend which does the actual work (blk, net, etc)**
b. **a transport which handles how that is exposed to the guest (pci, mmio, s390 channel controller, etc).**

Usage

Upon successful addition of a new device backend to QEMU (followed by a recompilation), a device can be attached to the Virtual Machine in two ways:

- At VM boot time via the `-device` option during the `$QEMU` invocation.
- Via "hot-plugging" after boot using the QEMU Monitor via the `` `device_attach <device-tag>` `` command.

The result is that the guest now has a view of the device using the `lspci` command. Subsequent accesses from the guest are driven by a frontend-driver that is bound to a particular device.

Note on the QEMU Object Model

Every QEMU device consists of a backend and a transport (bus). QEMU represents each through a hierarchy of classes and drives the recursive instantiation of this hierarchy during `qemu_init()`. The call order is `[[#.class_init() -> #.instance_init()]] -> #.realize()`

Each QEMU entity (devices and buses are entities) has a .c file that defines a `struct TypeInfo`. The `struct TypeInfo` of the current entity also contains the string name of its `#.parent`, thus defining an inheritance hierarchy spread across several source files. Methods of a parent class are inherited by every child.

Besides the inheritance hierarchy, the .c file also defines the `#.class_init()`, `#.instance_init()`, `and` `#.realize()` methods for that entity. Each of these methods has a specific purpose in the QEMU device creation procedure.

- The `class_init()` method of each `struct TypeInfo` is recursively called at the very start of $QEMU invocation, through the `type_init()` function which is really a constructor. This effectively converts an unusable QEMU entity (TypeInfo struct) to a OOP-like Class definition.
- The instance_init() method is called when device creation is requested via device_add or via command-line argument.

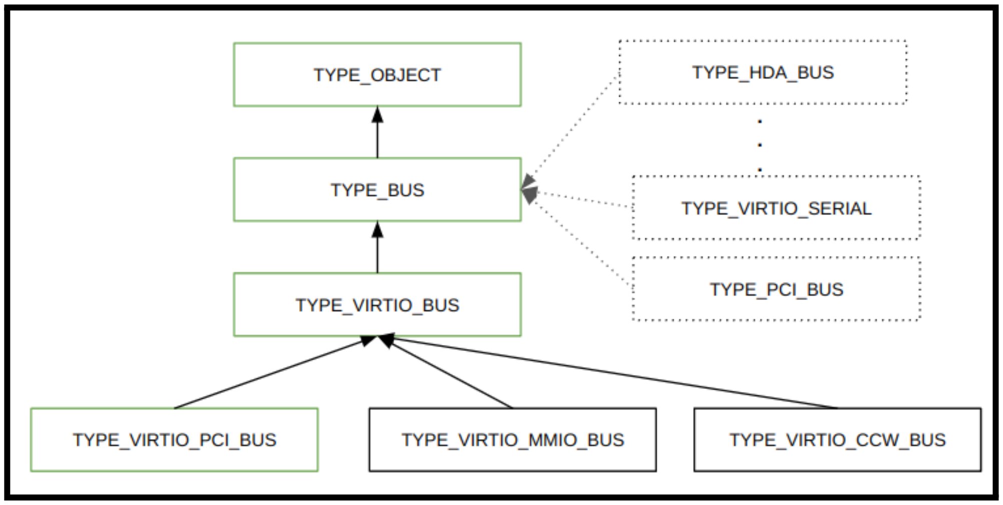


Fig 11. **QEMU's `TypeInfo` hierarchy for Buses**. The Bus hierarchy for VirtIO devices over PCI is highlighted. Unless your device uses the virtio-mmio bus (only available on the `microvm` machine), it is necessary to be attached to the VIRTIO_PCI_BUS.

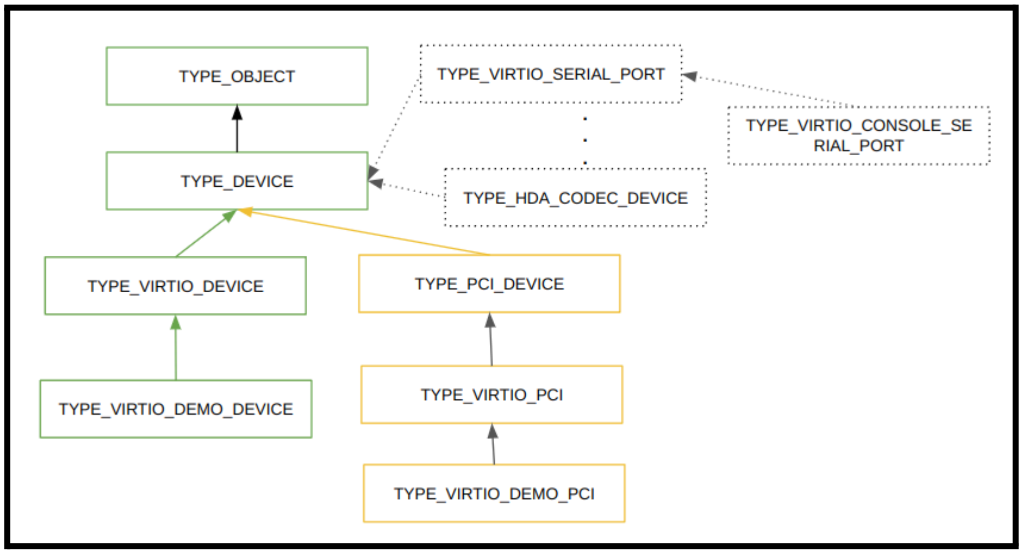


Fig 12. **QEMU's `TypeInfo` hierarchy of Device backends and PCI bindings**. The PCI bindings .c file is used to define a "has-a" relationship between the TYPE_VIRTIO_DEMO_PCI and the TYPE_VIRTIO_DEMO_DEVICE classes.

```
// qemu/hw/virtio/virtio-demo-pci.c
// template for any virtio-pci binding.
// Virtio PCI devices must extend the VirtIOPCIProxy
// base class.
// https://www.qemu.org/docs/master/devel/virtio-backends.html
struct VirtIODemoPCI {
    VirtIOPCIProxy parent_obj;

    // The .vdev is an object that represents a device backend
    // for this device. Every virtio device backend is registered on
    // the "virtio-bus", which itself is part of the pci bus. (see info qtree)
    // - I suspect the virtio-bus is mmio-based, which would make sense
    // considering virtqueue notifs are handled by the device backend
    // and not the pci-bindings file.
    // -----------------------------------------------------
    // Without pci bindings file, your device would be forced to use
    // the virtio-mmio bus, which is only implemented in the
    //  `microvm`machine.
    VirtIODemo vdev;
};
```

Fig 13. **Linkage between the PCI bindings and the virtio device.** VirtIODemo is an instance of the TYPE_VIRTIO_DEMO_DEVICE class. VirtIODemoPCI is an instance of the TYPE_VIRTIO_DEMO_PCI class. Every VirtIO PCI device must define this struct in its PCI-bindings file.

6. Writing a VirtIO Backend (After applying the helper patches)

(a) Build System Changes

`qemu/hw/virtio/Kconfig`: VirtIO KConfig
`qemu/hw/virtio/meson.build`: VirtIO subsystem build rules
In general, `Kconfig` files are used to assign boolean values (true/false) to Makefile variables by defining several dependencies. In QEMU, the meson build system uses the truth of these CONFIG_* variables to conditionally generate Make targets.

- `select` sets a variable to True.
- `depends on` defines a dependency - the default value is overridden by the truth value of the dependency.

```
// qemu/hw/virtio/Kconfig
config VIRTIO_DEMO
    bool
    default y
    depends on VIRTIO
```

Fig 9. VIRTIO_DEMO device depends on the target built by meson when CONFIG_VIRTIO is set.

```
// qemu/hw/virtio/Kconfig
config VIRTIO_PCI
    bool
    default y if PCI_DEVICES
    depends on PCI
    select VIRTIO
    select VIRTIO_MD_SUPPORTED
```

Fig 10. CONFIG_VIRTIO is always set.

Once it is established that CONFIG_VIRTIO_DEMO will evaluate to true, we add meson build rules to conditionally compile device-specific targets.

```
// qemu/hw/virtio/meson.build
[...]
specific_virtio_ss.add(when: 'CONFIG_VIRTIO_DEMO', if_true: files('virtio-demo.c'))
[...]
virtio_pci_ss.add(when: 'CONFIG_VIRTIO_DEMO', if_true: files('virtio-demo-pci.c'))
[..]
```

Fig 11. Adding two new targets to the qemu build.

(b) Defining core device functionality

`qemu/hw/virtio/virtio-demo-pci.c`: PCI bindings file
`hw/virtio/virtio-demo.c`: Device file
`include/hw/virtio/virtio-demo.h`, Device file header
`include/standard-headers/linux/virtio_demo.h` Device features header

Step 1: Get QEMU to recognize the name of your PCI device.

```
// in qemu/virtio/virtio-xblk-pci.c
#define TYPE_VIRTIO_XBLK_PCI "virtio-xblk-pci-base"
static const VirtioPCIDeviceTypeInfo virtio_demo_pci_info = {
    .parent = TYPE_VIRTIO_PCI, // see virtio-pci.h. Allows reg on the pci bus.
    .base_name = TYPE_VIRTIO_XBLK_PCI,
    .generic_name = "virtio-xblk-pci",
    .transitional_name = "virtio-xblk-pci-transitional",
    .non_transitional_name = "virtio-xblk-pci-non-transitional",
};

// Boilerplate to register a qemu type
static void virtio_demo_pci_register(void) {
    virtio_pci_types_register(&virtio_demo_pci_info);
}

type_init(virtio_demo_pci_register);
```

Fig 7. **Creating a TypeInfo struct for your new virtio device**. Added to a type-table. `$QEMU -device help` queries this table.

Step 2: Define a struct to represent your device backend

Changes to: `virtio-<device_name>.h`

```
// qemu/include/hw/virtio/virtio-xblk.h
#define TYPE_VIRTIO_XBLK "virtio-xblk-device" // name of backend

// create the VIRTIO_XBLK() macro that can obtain reference to a
// struct VirtIOXblk from an `Object`.
OBJECT_DECLARE_SIMPLE_TYPE(VirtIOXBlk, VIRTIO_XBLK)

// Define your device-backend struct
struct VirtIOXBlk {
    // Check the hierarchy in the doc,
    // each custom virtio device must inherit from the
    // TYPE_VIRTIO_DEVICE i.e. VirtIODevice class.
    VirtIODevice parent_obj;
    VirtQueue *vq;           // declare a single virtqueue
    uint64_t host_features;  // necessary by virtio spec
};
```

Step 3: Setup a PCI-bindings struct to allow your device to attach to the Virtio-PCI bus

```
// qemu/hw/virtio/virtio-xblk-pci
typedef struct VirtIOXBlkPCI VirtIOXBlkPCI;

struct VirtIOXBlkPCI {
    VirtIOPCIProxy parent_obj; // required acc. to qemu docs
    VirtIOXBlk vdev;           // link to an instance of your backend device
};


#define TYPE_VIRTIO_XBLK_PCI "virtio-xblk-pci-base"
```

Step 4. Define a set of properties for your device

```
// qemu/hw/virtio/virtio-blk-pci.c
static Property virtio_xblk_pci_properties[] = {
    DEFINE_PROP_BIT("ioeventfd", VirtIOPCIProxy, flags,
                    VIRTIO_PCI_FLAG_USE_IOEVENTFD_BIT, true), // enable the ioventfd mechanism
    DEFINE_PROP_UINT32("vectors", VirtIOPCIProxy, nvectors,
                       DEV_NVECTORS_UNSPECIFIED),
    DEFINE_PROP_END_OF_LIST(),
};
```

Fig 17. Defining a set of properties for the virtio bindings class. Every VirtIOPCIProxy object will have these properties

Step 5: Define #.class_init() method to create a useable OOP class from TypeInfo struct

The class_init() defines the attributes and methods that this upcoming OOP class has. In effect, it is a C++ class definition.

```
// qemu/hw/virtio/virtio-xblk-pci.c
static void virtio_xblk_pci_class_init(ObjectClass *klass, void *data) {
    // create direct references to all the
    // heirarchy classes to avoid an
    // obj->parent->parent chain.
    // when assigning properties
    DeviceClass *dc = DEVICE_CLASS(klass);
    PCIDeviceClass *pcidev_k = PCI_DEVICE_CLASS(klass);
    VirtioPCIClass *k = VIRTIO_PCI_CLASS(klass);

    // Register the properties
    device_class_set_props(dc, virtio_xblk_pci_properties);

    k->realize = virtio_demo_pci_realize;
    [...]
```

Fig 15. The class_init() of any custom virtio device will always have direct access to the class definitions of parent classes. This allows the class_init() of the pci-bindings file to assign values to attributes of parent classes, and also add new properties to classes.

Step 6: Enable successful device “attach” to the Virtio-PCI bus

Successful device attach requires two key steps: Creating an instance of the device backend, and qdev_realize() of the device backend. Since our entry point to device creation is the -pci binding, we must ensure that the #.instance_init() of the pci binding triggers the creation of an instance of the associated virtio backend.

```
// qemu/hw/virtio/virtio-xblk-pci.c

// QEMU cli_create_devices() triggers a instance creation
// for the device type passed. Here also, the idea is to create a
// object of .vdev type linked to the pci bindings object.
static void virtio_xblk_pci_instance_init(Object *obj) {
    VirtIOXBlkPCI *dev = VIRTIO_XBLK_PCI(obj);

    // creates the virtio backend for the virtio-bus
    virtio_instance_init_common(obj, &dev->vdev, sizeof(dev->vdev),
                                TYPE_VIRTIO_XBLK);

}
```

Fig 18. Step 1 of device attach - #.instance_init(). Any additional properties that were assigned during class_init will be a part of the dev->vdev instance.

```
// This function is called by QEMU main code, starting from qdev_device_add()
// and triggers the realization of the device backend,
// - Realization could involve several virtio-specific actions such
// as initializing virtqueues an their notification handlers.
static void virtio_xblk_pci_realize(VirtIOPCIProxy *vpci_dev, Error **errp) {
    VirtIOXBlkPCI *dev = VIRTIO_XBLK_PCI(vpci_dev);

    // not sure why cast to devicestate is valid
    DeviceState *vdev = DEVICE(&dev->vdev);

    vpci_dev->class_code = PCI_CLASS_OTHERS;

    // nvectors sets the number of MSI-X vectors supported
    // by this device. Subsequent virtqeueue <-> vector
    // mappings are done through a driver device handshake
    if (vpci_dev->nvectors == DEV_NVECTORS_UNSPECIFIED) {
        vpci_dev->nvectors = 2;
    }

    // @imp: eventually calls the realize of the backend device
    qdev_realize(vdev, BUS(&vpci_dev->bus), errp);
}
```

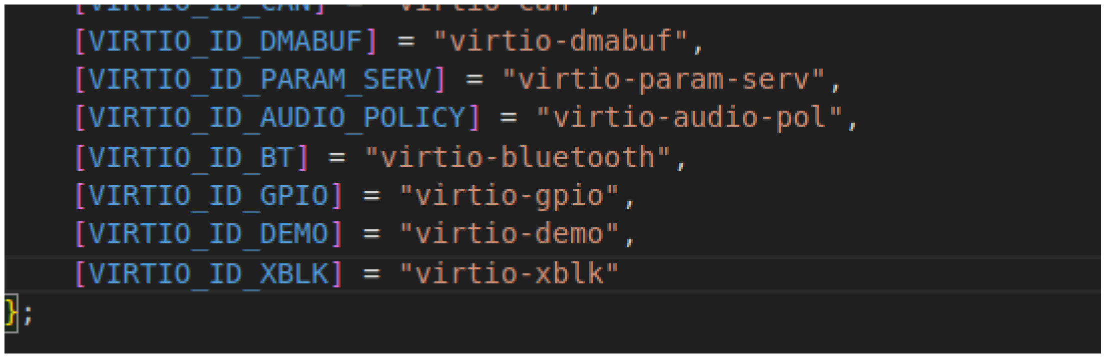
Fig 19. Step 2 of device attach - #.realize() -> qdev_realize(). The bus type was linked to the VirtIOPCIProxy device after #.instance_init() but before #.realize(). The #.realize() was called by qdev_realize() in main QEMU code.

While the above sequence sets up the pci-bindings framework correctly, it is still incomplete. The reason is simple - we have not defined a class to represent our backend device yet. Therefore, the device_add procedure will fail when it tries to init an instance of device backend. To fix, we must create a secondary source file, virtio-xblk.c

(c) Implement control and data-planes

(d) Miscellaneous Actions

Add device name to the virtio names list



```
    [VIRTIO_ID_DMABUF] = "virtio-dmabuf",
    [VIRTIO_ID_PARAM_SERV] = "virtio-param-serv",
    [VIRTIO_ID_AUDIO_POLICY] = "virtio-audio-pol",
    [VIRTIO_ID_BT] = "virtio-bluetooth",
    [VIRTIO_ID_GPIO] = "virtio-gpio",
    [VIRTIO_ID_DEMO] = "virtio-demo",
    [VIRTIO_ID_XBLK] = "virtio-xblk"
};
```

Add features to qmp-monitor to be able to query virtio-device status

virtio_queue_notify_vq (
Virtio_pci_notify (called on virtio_notify in guest)

Each entity in the QEMU hypervisor - cpu, devices, buses is implemented as a hierarchy of related/dependent classes (structs), each with `class_init(), instance_init() and realize()` methods. Every entity in QEMU is derived from the TYPE_OBJECT class - and the classes representing a virtual device entity are no different.

The pattern is simple: Each source defines a QEMU class via a `TypeInfo` struct. The developer specifies inheritance via the `.parent` attribute. The base Object class itself is defined as a TypeInfo struct.

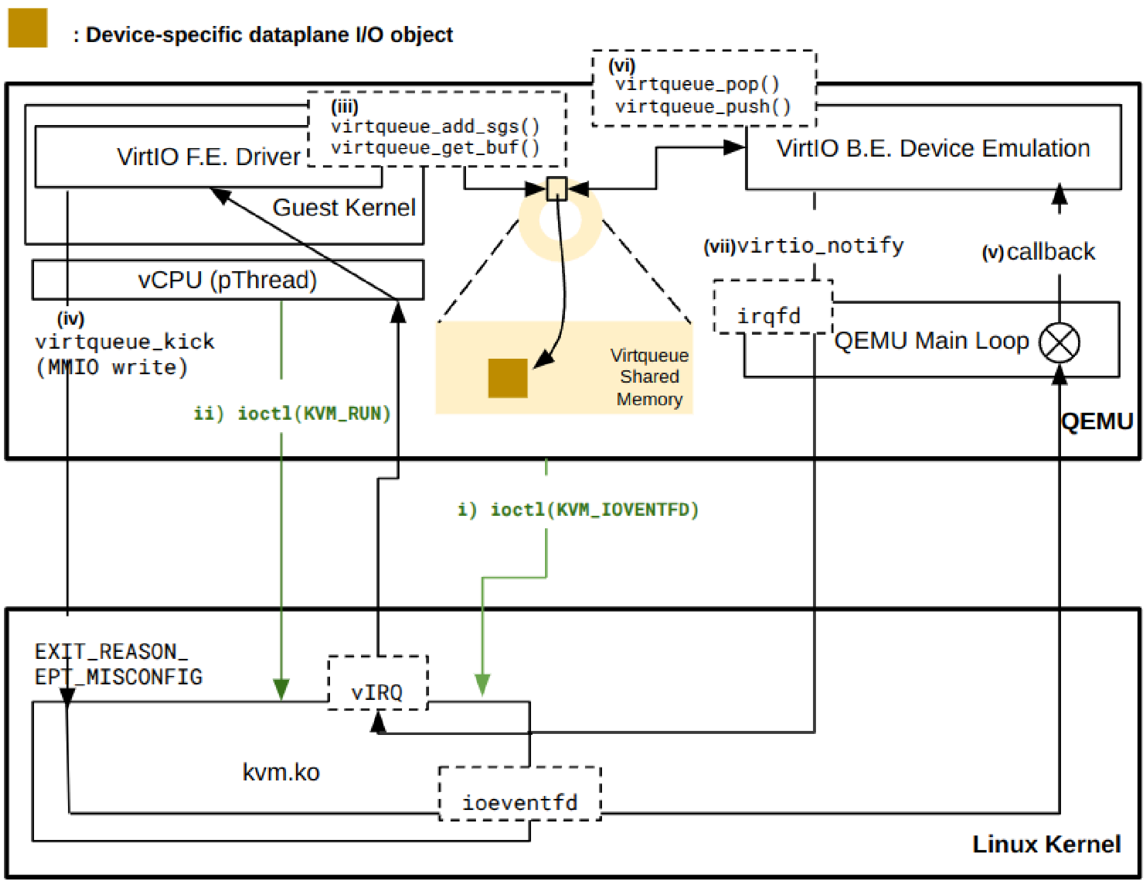
All modern devices should be represented as a derived QOM class of TYPE_DEVICE. The QEMU device API introduces the additional methods of @realize and @unrealize to represent additional stages in a device object's life cycle.

To implement a custom device backend for QEMU, one must override the default `TYPE_DEVICE` init methods. Our goal is to implement a VirtIO device over the PCIe bus. QEMU provides two helper classes for this purpose - the virtio-device and pci-device classes.

1. Register the `TypeInfo` with `TypeImpl`.
2. Instantiate the `ObjectClass`. (class_init)

3. Instantiate the `Object`.: Using `object_new(typename)` a new `Object` derivative will be instantiated from the type. (intance_init)
4. Add properties.

Mechanics of VirtIO Device Emulation



: Device-specific dataplane I/O object
(vi) virtqueue_pop() virtqueue_push()
(iii) virtqueue_add_sgs() virtqueue_get_buf()
VirtIO F.E. Driver
VirtIO B.E. Device Emulation
Guest Kernel
(vii)virtio_notify
(v)callback
vCPU (pThread)
irqfd
QEMU Main Loop
(iv) virtqueue_kick (MMIO write)
Virtqueue Shared Memory
ii) ioctl(KVM_RUN)
QEMU
i) ioctl(KVM_IOEVENTFD)
EXIT_REASON_ EPT_MISCONFIG
vIRQ
kvm.ko
ioeventfd
Linux Kernel

Fig 5. Mechanics of a QEMU VirtIO device emulation.
(i) The KVM_IOEVENTFD ioctl is called during QEMU initialization to associate guest physical address ranges (MMIO addresses) with KVM eventfd_ctx in the kernel as part of the ioeventfd setup.
(ii) QEMU spawns pthreads that are blocked in the KVM_RUN ioctl until a VM-Exit occurs due to a non-ioeventfd event.
(iii) (at this stage, the backend has been created, device has been attached, driver probe has completed) To send a message over the virtqueue, the FE driver uses virtqueue_add_sgs() specifying in_bufs and out_bufs as scatter-gather lists in guest kernel space.
(iv) The virtual machine requests I/O through the VirtIO Device Driver. The VirtIO Device Driver triggers exit to KVM through: `virtqueue_kick() -> virtqueue_notify() -> iowrite16(vq->index, (void __iomem *)vq->priv)`

(iii) KVM wakes up QEMU's Main Loop again using the `ioeventfd` mechanism.
(iv, v) The woken up QEMU enters device emulation and writes I/O data in the VirtQueue to the actual I/O Device and writes the response back to Virtqueue.
(vi, vii, viii) After that, QEMU tells the KVM Module to inject a Virtual IRQ to the Guest (at some vCPU thread) through `irqfd`, thereby completing the virtual machine's I/O processing. The irqfd virtual interrupt will be visible through `cat /proc/interrupts` in the guest.

a. QEMU declares a memory region(but does not allocate ram or commit it to kvm)
b. Guest's first access the MMIO address, cause an exit to KVM with EXIT_REASON_EPT_VIOLATION
c. KVM constructs the EPT page table and marks the page table entry with a special flag (110b). (see `kvm_mmu_set_ept_masks()`)
d. Later the guest access these MMIO, will also cause an exit to KVM, but with EXIT_REASON_EPT_MISCONFIG
    i. The handler for EXIT_REASON_EPT_MISCONFIG at the guestPA will result in an `eventfd_signal()` on a fd that was previously registered by QEMU.

6. VirtIO API Descriptions

Talk about in_sg, out_sg description

DRIVER_SIDE

`sg_init_one():`

- Given a kernel-allocated physically contiguous region (storing custom struct, array of ints etc.), create a scatter-gather list containing 1 scatter-gather (sg) entry. Each sg entry is now linked to a fixed mem region in kernel space.

`virtqueue_add_sgs():`

- Given a list of sg entries for outgoing data (outbufs) and empty buffers to store any device-written data (inbufs), and an associated `token`, this function creates descriptor chains, stored in the descriptor ring of the specified virtqueue and updates the available ring according to virtio spec.
- A single sg entry would create a single descriptor.

`virtqueue_get_buf():`

- Looks up the used ring, updates metadata. Finally, returns a pointer to the updated driver `token` that was registered via _add_sgs().

Virtio spec asserts that a descriptor entry must have a GPA to some buffer. This "buffer" is really just the struct that was previously contained in the sg entry.

DEVICE-SIDE

The `VirtqueueElement` is the representation of a message in the virtqueue. It contains references to the actual sg_in and sg_out, i.e. driver-initialized sg buffers meant to be written to and read from resp. by the device.

`iov_from_buf():`

- adds a device buffer to the specified sg-entry. The device backend will have written directly to the memory location of guest response inbuf through the DMA mapping created by the virtio subsystem.

`virtqueue_push()`:

- Writes descriptor metadata to the used ring. A virtqueue_get_buf() in driver will consume this.

Appendix

Downloading And Building QEMU

We choose the latest QEMU 9.2.1 (Feb 2025)

QEMU can be built with a vast set of optional features that can be configured at build time by the developer. These features require additional `sudo apt install-able` libraries that are subsequently linked with the built QEMU executable.

- Example: `sudo apt-get install libnfs-dev libiscsi-dev` for newer versions of debian/ubuntu.

Besides these optional features, QEMU also relies on several other open-source projects that are referenced via git submodules.

- Some modules `(e.g. ui/keycodemapdb)` are compulsory for all builds, while other modules `(e.g. dtc)` are optional depending on build config & host system. When *./configure* runs it will determine which submodules are to be used and print a list of them in the summary output.

Legacy v/s Modern Devices

QEMU devices are identified by a <PCI_VENDOR_ID, DEVICE_ID> pair. `pci-ids.rst` provides a list of reserved device-ids. The device-ids are the mechanism by which the guest drivers are chosen for a particular device.

a. For example, the 1000 -> 10ff device ID range is used for virtio-pci devices. The guest kernel is aware of this range and during boot, the driver-devices are bound to the `virtio-pci` kernel module. In the demo kernel image, `virtio-pci` is compiled into the kernel itself.
b. For an experimental device, the 1af4:10f0 to 1a4f:10ff range is recommended. These device ids are not bound to any driver by default. Therefore it is necessary to implement a new front-end driver as well.

In practice, ids for modern devices are computed and assigned by QEMU.

`virtio_legacy_allowed()`

- set device to legacy, then specify the device-id using a #define.
- exposes both PIO and MMIO region to guest.

If the `disable-legacy=on` option is passed to qemu monitor, the device-id is calc by qemu itself.

- Modern devices only expose MMIO regions to guest

Useful Monitor Commands

From QEMU monitor
`` `info virtio` ``, followed by
`` `info virtio-status <device_machine_str>` ``
`` `info virtio-queue-status <device_machine_str> <queue_index>` ``
`` `info mtree -f` ``: To tell if an MemoryRegion is registered in KVM (see kvm_int.h)

QEMU Object Model

Each entity in the QEMU hypervisor - cpu, devices, buses is implemented as a hierarchy of related/dependent classes (structs), each with `class_init(), instance_init() and realize()` methods. Every entity in qemu is derived from the Object class - and the classes representing a virtual device entity are no different.

```
// qemu/qom/qom.c
static const TypeInfo object_info = {
     .name = TYPE_OBJECT,
     .instance_size = sizeof(Object),
     .class_init = object_class_init,
     .abstract = true,
 };
```

Fig 3. Definition of the TYPE_OBJECT base class. The pattern is simple: Each file defines a class via a `TypeInfo` struct. The developer specifies inheritance via the .parent attribute. The Object class itself is defined as a TypeInfo struct.

```
// qemu/hw/core/qdev.c
static const TypeInfo device_type_info = {
    .name = TYPE_DEVICE,
    .parent = TYPE_OBJECT,
    .instance_size = sizeof(DeviceState),
    .instance_init = device_initfn,
    .instance_post_init = device_post_init,
    .instance_finalize = device_finalize,
    .class_base_init = device_class_base_init,
    .class_init = device_class_init,
    .abstract = true,
    .class_size = sizeof(DeviceClass),
    .interfaces = (InterfaceInfo[]) {
        { TYPE_VMSTATE_IF },
        { TYPE_RESETTABLE_INTERFACE },
        { }
    }
};
```

Fig 4. Definition of the QEMU TYPE_DEVICE class. Every qemu virtual device is represented as a hierarchy of classes that are derived from the Device class. Similar to the Object class, the Device class has its own definition as well.

QEMU emulates a variety of devices - each class derives from the `TYPE_DEVICE` class.

Usually, devices that conform to a standard (PCIe, SCSI, etc) also have their own buses (which themselves are represented as a hierarchy of subclasses of `TYPE_BUS`)

Therefore, a QEMU virtual device is defined by its `<device_info, bus_info>` pair. Note that depending on the device, the `device_info` and `bus_info` class hierarchy may be defined across several source files.

```
// pci.c
static const TypeInfo pci_device_type_info = {
    .name = TYPE_PCI_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(PCIDevice),
    .abstract = true,
    .class_size = sizeof(PCIDeviceClass),
    .class_init = pci_device_class_init,
    .class_base_init = pci_device_class_base_in
};

static const TypeInfo pci_bus_info = {
    .name = TYPE_PCI_BUS,
    .parent = TYPE_BUS,
    .instance_size = sizeof(PCIBus),
    .class_size = sizeof(PCIBusClass),
    .class_init = pci_bus_class_init,
};

static void pci_register_types(void)
{
    [...]
    type_register_static(&pcie_bus_info);
    [...]
    type_register_static(&pci_device_type_info)
}

type_init(pci_register_types)
```

Fig 5. QEMU code snippet showing the bus-device TypeInfo

heirarchy for all QEMU PCI devices.

All modern devices should be represented as a derived QOM class of TYPE_DEVICE. The device API introduces the additional methods of @realize and @unrealize to represent additional stages in a device object's life cycle.

To implement a custom device backend for QEMU, one must override the default `TYPE_DEVICE` init methods. Our goal is to implement a VirtIO device over the PCIe bus. QEMU provides two helper classes for this purpose - the virtio-device and pci-device classes.

5. Register the `TypeInfo` with `TypeImpl`.
6. Instantiate the `ObjectClass`. (class_init)
7. Instantiate the `Object`.: Using `object_new(typename)` a new `Object` derivative will be instantiated from the type. (intance_init)
8. Add properties.

At QEMU startup, all TypeInfo structs are registered as TypeInfo structs in a map via the `type_register()` constructor. Therefore the class_init(), instance_init() and _realize() methods are associated with each class, indexed by the class name. **Every TypeInfo must have a class_init() method defined.**

Each class has an associated hash table of properties (attributes) with getters and setters, specified in the class_init() function. These can be set via helper macros and `device_class_set_props()`. Most classes also have a _realize() method. At QEMU initialization time, the class_init() method of every Type is called. (starting with object_class_init(), device_class_init(), etc.

References

https://web.archive.org/web/20210921171542/http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html
https://web.archive.org/web/20210923215006/https://ssup2.github.io/theory_analysis/IO_Virtualization_Software/
https://insujang.github.io/2021-03-10/virtio-and-vhost-architecture-part-1/
https://www.redhat.com/en/blog/virtqueues-and-virtio-ring-how-data-travels
https://stackoverflow.com/questions/65194712/in-which-conditions-the-ioctl-kvm-run-returns
https://docs.kernel.org/virt/kvm/api.html
https://fosdem.org/2025/events/attachments/fosdem-2025-5100-can-qemu-and-vhost-user-devices-be-used-on-macos-and-bsd-/slides/238480/FOSDEM_20_O6dJsKR.pdf
https://lists.endsoftwarepatents.org/archive/html/qemu-discuss/2023-07/msg00019.html
https://blog.vmsplice.net/2024/01/qemu-aiocontext-removal-and-how-it-was.html

Diagram:
https://docs.google.com/presentation/d/1XVj3-xTipVv0eLgL2QUTKFFFvLZDilmTyUxpT2rFZpg/edit?usp=sharing