QEMU Documentation

Release 5.1.50

The QEMU Project Developers

Sep 07, 2020

Contents:

CHAPTER 1

QEMU System Emulation User's Guide

This manual is the overall guide for users using QEMU for full system emulation (as opposed to user-mode emulation). This includes working with hypervisors such as KVM, Xen, Hax or Hypervisor.Framework.

Contents:

1.1 Quick Start

Download and uncompress a PC hard disk image with Linux installed (e.g. `linux.img`) and type:

```
qemu-system-x86_64 linux.img
```

Linux should boot and give you a prompt.

1.2 Invocation

```
qemu-system-x86_64 [options] [disk_image]
```

disk_image is a raw hard disk image for IDE hard disk 0. Some targets do not need a disk image.

1.2.1 Standard options

-h Display help and exit

-version Display version information and exit

-machine [type=]name[,prop=value[,...]] Select the emulated machine by name. Use `-machine help` to list available machines.

For architectures which aim to support live migration compatibility across releases, each release will introduce a new versioned machine type. For example, the 2.8.0 release introduced machine types "pc-i440fx-2.8" and "pc-q35-2.8" for the x86_64/i686 architectures.

To allow live migration of guests from QEMU version 2.8.0, to QEMU version 2.9.0, the 2.9.0 version must support the "pc-i440fx-2.8" and "pc-q35-2.8" machines too. To allow users live migrating VMs to skip multiple intermediate releases when upgrading, new releases of QEMU will support machine types from many previous versions.

Supported machine properties are:

`accel=accels1[:accels2[:...]]` This is used to enable an accelerator. Depending on the target architecture, kvm, xen, hax, hvf, whpx or tcg can be available. By default, tcg is used. If there is more than one accelerator specified, the next one is used if the previous one fails to initialize.

`vmport=on|off|auto` Enables emulation of VMWare IO port, for vmmouse etc. auto says to select the value based on accel. For accel=xen the default is off otherwise the default is on.

`dump-guest-core=on|off` Include guest memory in a core dump. The default is on.

`mem-merge=on|off` Enables or disables memory merge support. This feature, when supported by the host, de-duplicates identical memory pages among VMs instances (enabled by default).

`aes-key-wrap=on|off` Enables or disables AES key wrapping support on s390-ccw hosts. This feature controls whether AES wrapping keys will be created to allow execution of AES cryptographic functions. The default is on.

`dea-key-wrap=on|off` Enables or disables DEA key wrapping support on s390-ccw hosts. This feature controls whether DEA wrapping keys will be created to allow execution of DEA cryptographic functions. The default is on.

`nvdimm=on|off` Enables or disables NVDIMM support. The default is off.

`enforce-config-section=on|off` If `enforce-config-section` is set to on, force migration code to send configuration section even if the machine-type sets the `migration.send-configuration` property to off. NOTE: this parameter is deprecated. Please use `-global migration.send-configuration`=on|off instead.

`memory-encryption=` Memory encryption object to use. The default is none.

`hmat=on|off` Enables or disables ACPI Heterogeneous Memory Attribute Table (HMAT) support. The default is off.

`-cpu model` Select CPU model (`-cpu help` for list and additional feature selection)

`-accel name[,prop=value[,...]]` This is used to enable an accelerator. Depending on the target architecture, kvm, xen, hax, hvf, whpx or tcg can be available. By default, tcg is used. If there is more than one accelerator specified, the next one is used if the previous one fails to initialize.

`igd-passthru=on|off` When Xen is in use, this option controls whether Intel integrated graphics devices can be passed through to the guest (default=off)

`kernel-irqchip=on|off|split` Controls KVM in-kernel irqchip support. The default is full acceleration of the interrupt controllers. On x86, split irqchip reduces the kernel attack surface, at a performance cost for non-MSI interrupts. Disabling the in-kernel irqchip completely is not recommended except for debugging purposes.

`kvm-shadow-mem=size` Defines the size of the KVM shadow MMU.

`tb-size=n` Controls the size (in MiB) of the TCG translation block cache.

`thread=single|multi` Controls number of TCG threads. When the TCG is multi-threaded there will be one thread per vCPU therefor taking advantage of additional host cores. The default is to enable multi-threading where both the back-end and front-ends support it and no incompatible TCG features have been enabled (e.g. icount/replay).

-smp [cpus=]n[,cores=cores][,threads=threads][,dies=dies][,sockets=sockets][,maxcpus=maxcpu

Simulate an SMP system with n CPUs. On the PC target, up to 255 CPUs are supported. On Sparc32 target, Linux limits the number of usable CPUs to 4. For the PC target, the number of cores per die, the number of threads per cores, the number of dies per packages and the total number of sockets can be specified. Missing values will be computed. If any on the three values is given, the total number of CPUs n can be omitted. maxcpus specifies the maximum number of hotpluggable CPUs.

-numa node[,mem=size][,cpus=firstcpu[-lastcpu]][,nodeid=node][,initiator=initiator]

-numa node[,memdev=id][,cpus=firstcpu[-lastcpu]][,nodeid=node][,initiator=initiator]

-numa dist,src=source,dst=destination,val=distance

-numa cpu,node-id=node[,socket-id=x][,core-id=y][,thread-id=z]

-numa hmat-lb,initiator=node,target=node,hierarchy=hierarchy,data-type=tpye[,latency=lat][,

-numa hmat-cache,node-id=node,size=size,level=level[,associativity=str][,policy=str][,line=

Define a NUMA node and assign RAM and VCPUs to it. Set the NUMA distance from a source node to a destination node. Set the ACPI Heterogeneous Memory Attributes for the given nodes.

Legacy VCPU assignment uses '`cpus`' option where firstcpu and lastcpu are CPU indexes. Each '`cpus`' option represent a contiguous range of CPU indexes (or a single VCPU if lastcpu is omitted). A non-contiguous set of VCPUs can be represented by providing multiple '`cpus`' options. If '`cpus`' is omitted on all nodes, VCPUs are automatically split between them.

For example, the following option assigns VCPUs 0, 1, 2 and 5 to a NUMA node:

```
-numa node,cpus=0-2,cpus=5
```

'`cpu`' option is a new alternative to '`cpus`' option which uses '`socket-id|core-id|thread-id`' properties to assign CPU objects to a node using topology layout properties of CPU. The set of properties is machine specific, and depends on used machine type/'`smp`' options. It could be queried with '`hotpluggable-cpus`' monitor command. '`node-id`' property specifies node to which CPU object will be assigned, it's required for node to be declared with '`node`' option before it's used with '`cpu`' option.

For example:

```
-M pc \
-smp 1,sockets=2,maxcpus=2 \
-numa node,nodeid=0 -numa node,nodeid=1 \
-numa cpu,node-id=0,socket-id=0 -numa cpu,node-id=1,socket-id=1
```

Legacy '`mem`' assigns a given RAM amount to a node (not supported for 5.1 and newer machine types). '`memdev`' assigns RAM from a given memory backend device to a node. If '`mem`' and '`memdev`' are omitted in all nodes, RAM is split equally between them.

'`mem`' and '`memdev`' are mutually exclusive. Furthermore, if one node uses '`memdev`', all of them have to use it.

'`initiator`' is an additional option that points to an initiator NUMA node that has best performance (the lowest latency or largest bandwidth) to this NUMA node. Note that this option can be set only when the machine property 'hmat' is set to 'on'.

Following example creates a machine with 2 NUMA nodes, node 0 has CPU. node 1 has only memory, and its initiator is node 0. Note that because node 0 has CPU, by default the initiator of node 0 is itself and must be itself.

```
-machine hmat=on \
-m 2G,slots=2,maxmem=4G \
-object memory-backend-ram,size=1G,id=m0 \
-object memory-backend-ram,size=1G,id=m1 \
-numa node,nodeid=0,memdev=m0 \
-numa node,nodeid=1,memdev=m1,initiator=0 \
-smp 2,sockets=2,maxcpus=2  \
-numa cpu,node-id=0,socket-id=0 \
-numa cpu,node-id=0,socket-id=1
```

source and destination are NUMA node IDs. distance is the NUMA distance from source to destination. The distance from a node to itself is always 10. If any pair of nodes is given a distance, then all pairs must be given distances. Although, when distances are only given in one direction for each pair of nodes, then the distances in the opposite directions are assumed to be the same. If, however, an asymmetrical pair of distances is given for even one node pair, then all node pairs must be provided distance values for both directions, even when they are symmetrical. When a node is unreachable from another node, set the pair's distance to 255.

Note that the `-numa` option doesn't allocate any of the specified resources, it just assigns existing resources to NUMA nodes. This means that one still has to use the `-m`, `-smp` options to allocate RAM and VCPUs respectively.

Use '`hmat-lb`' to set System Locality Latency and Bandwidth Information between initiator and target NUMA nodes in ACPI Heterogeneous Attribute Memory Table (HMAT). Initiator NUMA node can create memory requests, usually it has one or more processors. Target NUMA node contains addressable memory.

In '`hmat-lb`' option, node are NUMA node IDs. hierarchy is the memory hierarchy of the target NUMA node: if hierarchy is 'memory', the structure represents the memory performance; if hierarchy is 'first-level|second-level|third-level', this structure represents aggregated performance of memory side caches for each domain. type of 'data-type' is type of data represented by this structure instance: if 'hierarchy' is 'memory', 'data-type' is 'access|read|write' latency or 'access|read|write' bandwidth of the target memory; if 'hierarchy' is 'first-level|second-level|third-level', 'data-type' is 'access|read|write' hit latency or 'access|read|write' hit bandwidth of the target memory side cache.

lat is latency value in nanoseconds. bw is bandwidth value, the possible value and units are NUM[M|G|T], mean that the bandwidth value are NUM byte per second (or MB/s, GB/s or TB/s depending on used suffix). Note that if latency or bandwidth value is 0, means the corresponding latency or bandwidth information is not provided.

In '`hmat-cache`' option, node-id is the NUMA-id of the memory belongs. size is the size of memory side cache in bytes. level is the cache level described in this structure, note that the cache level 0 should not be used with '`hmat-cache`' option. associativity is the cache associativity, the possible value is 'none/direct(direct-mapped)/complex(complex cache indexing)'. policy is the write policy. line is the cache Line size in bytes.

For example, the following options describe 2 NUMA nodes. Node 0 has 2 cpus and a ram, node 1 has only a ram. The processors in node 0 access memory in node 0 with access-latency 5 nanoseconds, access-bandwidth is 200 MB/s; The processors in NUMA node 0 access memory in NUMA node 1 with access-latency 10 nanoseconds, access-bandwidth is 100 MB/s. And for memory side cache information, NUMA node 0 and 1 both have 1 level memory cache, size is 10KB, policy is write-back, the cache Line size is 8 bytes:

```
-machine hmat=on \
-m 2G \
-object memory-backend-ram,size=1G,id=m0 \
-object memory-backend-ram,size=1G,id=m1 \
-smp 2 \
-numa node,nodeid=0,memdev=m0 \
-numa node,nodeid=1,memdev=m1,initiator=0 \
-numa cpu,node-id=0,socket-id=0 \
-numa cpu,node-id=0,socket-id=1 \
```

(continues on next page)

(continued from previous page)

```
-numa hmat-lb,initiator=0,target=0,hierarchy=memory,data-type=access-latency,
↪latency=5 \
-numa hmat-lb,initiator=0,target=0,hierarchy=memory,data-type=access-bandwidth,
↪bandwidth=200M \
-numa hmat-lb,initiator=0,target=1,hierarchy=memory,data-type=access-latency,
↪latency=10 \
-numa hmat-lb,initiator=0,target=1,hierarchy=memory,data-type=access-bandwidth,
↪bandwidth=100M \
-numa hmat-cache,node-id=0,size=10K,level=1,associativity=direct,policy=write-
↪back,line=8 \
-numa hmat-cache,node-id=1,size=10K,level=1,associativity=direct,policy=write-
↪back,line=8
```

`-add-fd fd=fd,set=set[,opaque=opaque]` Add a file descriptor to an fd set. Valid options are:

`fd=fd` This option defines the file descriptor of which a duplicate is added to fd set. The file descriptor cannot be stdin, stdout, or stderr.

`set=set` This option defines the ID of the fd set to add the file descriptor to.

`opaque=opaque` This option defines a free-form string that can be used to describe fd.

You can open an image using pre-opened file descriptors from an fd set:

```
qemu-system-x86_64  -add-fd fd=3,set=2,opaque="rdwr:/path/to/file"  -add-
↪fd fd=4,set=2,opaque="rdonly:/path/to/file"  -drive file=/dev/fdset/2,
↪index=0,media=disk
```

`-set group.id.arg=value` Set parameter arg for item id of type group

`-global driver.prop=value`

`-global driver=driver,property=property,value=value` Set default value of driver's property prop to value, e.g.:

```
qemu_system-x86_64 -global ide-hd.physical_block_size=4096 disk-image.img
```

In particular, you can use this to set driver properties for devices which are created automatically by the machine model. To create a device which is not created automatically and set properties on it, use `-device`.

-global driver.prop=value is shorthand for -global driver=driver,property=prop,value=value. The longhand syntax works even when driver contains a dot.

`-boot [order=drives][,once=drives][,menu=on|off][,splash=sp_name][,splash-time=sp_time][,re`
Specify boot order drives as a string of drive letters. Valid drive letters depend on the target architecture. The x86 PC uses: a, b (floppy 1 and 2), c (first hard disk), d (first CD-ROM), n-p (Etherboot from network adapter 1-4), hard disk boot is the default. To apply a particular boot order only on the first startup, specify it via `once`. Note that the `order` or `once` parameter should not be used together with the `bootindex` property of devices, since the firmware implementations normally do not support both at the same time.

Interactive boot menus/prompts can be enabled via `menu=on` as far as firmware/BIOS supports them. The default is non-interactive boot.

A splash picture could be passed to bios, enabling user to show it as logo, when option splash=sp_name is given and menu=on, If firmware/BIOS supports them. Currently Seabios for X86 system support it. limitation: The splash file could be a jpeg file or a BMP file in 24 BPP format(true color). The resolution should be supported by the SVGA mode, so the recommended is 320x240, 640x480, 800x640.

A timeout could be passed to bios, guest will pause for rb_timeout ms when boot failed, then reboot. If rb_timeout is '-1', guest will not reboot, qemu passes '-1' to bios by default. Currently Seabios for X86 system support it.

Do strict boot via `strict=on` as far as firmware/BIOS supports it. This only effects when boot priority is changed by bootindex options. The default is non-strict boot.

```
# try to boot from network first, then from hard disk
qemu_system-x86_64 -boot order=nc
# boot from CD-ROM first, switch back to default order after reboot
qemu_system-x86_64 -boot once=d
# boot with a splash picture for 5 seconds.
qemu_system-x86_64 -boot menu=on,splash=/root/boot.bmp,splash-time=5000
```

Note: The legacy format '-boot drives' is still supported but its use is discouraged as it may be removed from future versions.

`-m [size=]megs[,slots=n,maxmem=size]` Sets guest startup RAM size to megs megabytes. Default is 128 MiB. Optionally, a suffix of "M" or "G" can be used to signify a value in megabytes or gigabytes respectively. Optional pair slots, maxmem could be used to set amount of hotpluggable memory slots and maximum amount of memory. Note that maxmem must be aligned to the page size.

For example, the following command-line sets the guest startup RAM size to 1GB, creates 3 slots to hotplug additional memory and sets the maximum memory the guest can reach to 4GB:

```
qemu-system-x86_64 -m 1G,slots=3,maxmem=4G
```

If slots and maxmem are not specified, memory hotplug won't be enabled and the guest startup RAM will never increase.

`-mem-path path` Allocate guest RAM from a temporarily created file in path.

`-mem-prealloc` Preallocate memory when using -mem-path.

`-k language` Use keyboard layout language (for example `fr` for French). This option is only needed where it is not easy to get raw PC keycodes (e.g. on Macs, with some X11 servers or with a VNC or curses display). You don't normally need to use it on PC/Linux or PC/Windows hosts.

The available layouts are:

```
ar  de-ch  es  fo     fr-ca  hu  ja  mk     no  pt-br  sv
da  en-gb  et  fr     fr-ch  is  lt  nl     pl  ru     th
de  en-us  fi  fr-be  hr     it  lv  nl-be  pt  sl     tr
```

The default is `en-us`.

`-audio-help` Will show the -audiodev equivalent of the currently specified (deprecated) environment variables.

`-audiodev [driver=]driver,id=id[,prop[=value][,...]]` Adds a new audio backend driver identified by id. There are global and driver specific properties. Some values can be set differently for input and output, they're marked with `in|out.`. You can set the input's property with `in.prop` and the output's property with `out.prop`. For example:

```
-audiodev alsa,id=example,in.frequency=44110,out.frequency=8000
-audiodev alsa,id=example,out.channels=1 # leaves in.channels unspecified
```

NOTE: parameter validation is known to be incomplete, in many cases specifying an invalid option causes QEMU to print an error message and continue emulation without sound.

Valid global options are:

`id=identifier` Identifies the audio backend.

`timer-period=period` Sets the timer period used by the audio subsystem in microseconds. Default is 10000 (10 ms).

in|out.mixing-engine=on|off Use QEMU's mixing engine to mix all streams inside QEMU and convert audio formats when not supported by the backend. When off, fixed-settings must be off too. Note that disabling this option means that the selected backend must support multiple streams and the audio formats used by the virtual cards, otherwise you'll get no sound. It's not recommended to disable this option unless you want to use 5.1 or 7.1 audio, as mixing engine only supports mono and stereo audio. Default is on.

in|out.fixed-settings=on|off Use fixed settings for host audio. When off, it will change based on how the guest opens the sound card. In this case you must not specify frequency, channels or format. Default is on.

in|out.frequency=frequency Specify the frequency to use when using fixed-settings. Default is 44100Hz.

in|out.channels=channels Specify the number of channels to use when using fixed-settings. Default is 2 (stereo).

in|out.format=format Specify the sample format to use when using fixed-settings. Valid values are: `s8`, `s16`, `s32`, `u8`, `u16`, `u32`, `f32`. Default is `s16`.

in|out.voices=voices Specify the number of voices to use. Default is 1.

in|out.buffer-length=usecs Sets the size of the buffer in microseconds.

-audiodev none,id=id[,prop[=value][,...]] Creates a dummy backend that discards all outputs. This backend has no backend specific properties.

-audiodev alsa,id=id[,prop[=value][,...]] Creates backend using the ALSA. This backend is only available on Linux.

ALSA specific options are:

in|out.dev=device Specify the ALSA device to use for input and/or output. Default is `default`.

in|out.period-length=usecs Sets the period length in microseconds.

in|out.try-poll=on|off Attempt to use poll mode with the device. Default is on.

threshold=threshold Threshold (in microseconds) when playback starts. Default is 0.

-audiodev coreaudio,id=id[,prop[=value][,...]] Creates a backend using Apple's Core Audio. This backend is only available on Mac OS and only supports playback.

Core Audio specific options are:

in|out.buffer-count=count Sets the count of the buffers.

-audiodev dsound,id=id[,prop[=value][,...]] Creates a backend using Microsoft's DirectSound. This backend is only available on Windows and only supports playback.

DirectSound specific options are:

latency=usecs Add extra usecs microseconds latency to playback. Default is 10000 (10 ms).

-audiodev oss,id=id[,prop[=value][,...]] Creates a backend using OSS. This backend is available on most Unix-like systems.

OSS specific options are:

in|out.dev=device Specify the file name of the OSS device to use. Default is `/dev/dsp`.

in|out.buffer-count=count Sets the count of the buffers.

in|out.try-poll=on|of Attempt to use poll mode with the device. Default is on.

try-mmap=on|off Try using memory mapped device access. Default is off.

exclusive=on|off Open the device in exclusive mode (vmix won't work in this case). Default is off.

dsp-policy=policy Sets the timing policy (between 0 and 10, where smaller number means smaller latency but higher CPU usage). Use -1 to use buffer sizes specified by `buffer` and `buffer-count`. This option is ignored if you do not have OSS 4. Default is 5.

-audiodev pa,id=id[,prop[=value][,...]] Creates a backend using PulseAudio. This backend is available on most systems.

PulseAudio specific options are:

server=server Sets the PulseAudio server to connect to.

in|out.name=sink Use the specified source/sink for recording/playback.

in|out.latency=usecs Desired latency in microseconds. The PulseAudio server will try to honor this value but actual latencies may be lower or higher.

-audiodev sdl,id=id[,prop[=value][,...]] Creates a backend using SDL. This backend is available on most systems, but you should use your platform's native backend if possible. This backend has no backend specific properties.

-audiodev spice,id=id[,prop[=value][,...]] Creates a backend that sends audio through SPICE. This backend requires `-spice` and automatically selected in that case, so usually you can ignore this option. This backend has no backend specific properties.

-audiodev wav,id=id[,prop[=value][,...]] Creates a backend that writes audio to a WAV file.

Backend specific options are:

path=path Write recorded audio into the specified file. Default is `qemu.wav`.

-soundhw card1[,card2,...] or -soundhw all Enable audio and selected sound hardware. Use 'help' to print all available sound hardware. For example:

```
qemu_system-x86_64 -soundhw sb16,adlib disk.img
qemu_system-x86_64 -soundhw es1370 disk.img
qemu_system-x86_64 -soundhw ac97 disk.img
qemu_system-x86_64 -soundhw hda disk.img
qemu_system-x86_64 -soundhw all disk.img
qemu_system-x86_64 -soundhw help
```

Note that Linux's i810_audio OSS kernel (for AC97) module might require manually specifying clocking.

```
modprobe i810_audio clocking=48000
```

-device driver[,prop[=value][,...]] Add device driver. prop=value sets driver properties. Valid properties depend on the driver. To get help on possible drivers and properties, use `-device help` and `-device driver,help`.

Some drivers are:

-device ipmi-bmc-sim,id=id[,prop[=value][,...]] Add an IPMI BMC. This is a simulation of a hardware management interface processor that normally sits on a system. It provides a watchdog and the ability to reset and power control the system. You need to connect this to an IPMI interface to make it useful

The IPMI slave address to use for the BMC. The default is 0x20. This address is the BMC's address on the I2C network of management controllers. If you don't know what this means, it is safe to ignore it.

id=id The BMC id for interfaces to use this device.

slave_addr=val Define slave address to use for the BMC. The default is 0x20.

sdrfile=file file containing raw Sensor Data Records (SDR) data. The default is none.

fruareasize=val size of a Field Replaceable Unit (FRU) area. The default is 1024.

frudatafile=file file containing raw Field Replaceable Unit (FRU) inventory data. The default is none.

guid=uuid value for the GUID for the BMC, in standard UUID format. If this is set, get "Get GUID" command to the BMC will return it. Otherwise "Get GUID" will return an error.

-device ipmi-bmc-extern,id=id,chardev=id[,slave_addr=val] Add a connection to an external IPMI BMC simulator. Instead of locally emulating the BMC like the above item, instead connect to an external entity that provides the IPMI services.

A connection is made to an external BMC simulator. If you do this, it is strongly recommended that you use the "reconnect=" chardev option to reconnect to the simulator if the connection is lost. Note that if this is not used carefully, it can be a security issue, as the interface has the ability to send resets, NMIs, and power off the VM. It's best if QEMU makes a connection to an external simulator running on a secure port on localhost, so neither the simulator nor QEMU is exposed to any outside network.

See the "lanserv/README.vm" file in the OpenIPMI library for more details on the external interface.

-device isa-ipmi-kcs,bmc=id[,ioport=val][,irq=val] Add a KCS IPMI interafce on the ISA bus. This also adds a corresponding ACPI and SMBIOS entries, if appropriate.

bmc=id The BMC to connect to, one of ipmi-bmc-sim or ipmi-bmc-extern above.

ioport=val Define the I/O address of the interface. The default is 0xca0 for KCS.

irq=val Define the interrupt to use. The default is 5. To disable interrupts, set this to 0.

-device isa-ipmi-bt,bmc=id[,ioport=val][,irq=val] Like the KCS interface, but defines a BT interface. The default port is 0xe4 and the default interrupt is 5.

-device pci-ipmi-kcs,bmc=id Add a KCS IPMI interafce on the PCI bus.

bmc=id The BMC to connect to, one of ipmi-bmc-sim or ipmi-bmc-extern above.

-device pci-ipmi-bt,bmc=id Like the KCS interface, but defines a BT interface on the PCI bus.

-name name Sets the name of the guest. This name will be displayed in the SDL window caption. The name will also be used for the VNC server. Also optionally set the top visible process name in Linux. Naming of individual threads can also be enabled on Linux to aid debugging.

-uuid uuid Set system UUID.

1.2.2 Block device options

-fda file

-fdb file Use file as floppy disk 0/1 image (see *Disk Images*).

-hda file

-hdb file

-hdc file

-hdd file Use file as hard disk 0, 1, 2 or 3 image (see *Disk Images*).

-cdrom file Use file as CD-ROM image (you cannot use `-hdc` and `-cdrom` at the same time). You can use the host CD-ROM by using `/dev/cdrom` as filename.

-blockdev option[,option[,option[,...]]] Define a new block driver node. Some of the options apply to all block drivers, other options are only accepted for a specific block driver. See below for a list of generic options and options for the most common block drivers.

Options that expect a reference to another node (e.g. `file`) can be given in two ways. Either you specify the node name of an already existing node (file=node-name), or you define a new node inline, adding options for the referenced node after a dot (file.filename=path,file.aio=native).

A block driver node created with `-blockdev` can be used for a guest device by specifying its node name for the `drive` property in a `-device` argument that defines a block device.

`Valid options for any block driver node:`

`driver` Specifies the block driver to use for the given node.

`node-name` This defines the name of the block driver node by which it will be referenced later. The name must be unique, i.e. it must not match the name of a different block driver node, or (if you use `-drive` as well) the ID of a drive.

If no node name is specified, it is automatically generated. The generated node name is not intended to be predictable and changes between QEMU invocations. For the top level, an explicit node name must be specified.

`read-only` Open the node read-only. Guest write attempts will fail.

Note that some block drivers support only read-only access, either generally or in certain configurations. In this case, the default value `read-only=off` does not work and the option must be specified explicitly.

`auto-read-only` If `auto-read-only=on` is set, QEMU may fall back to read-only usage even when `read-only=off` is requested, or even switch between modes as needed, e.g. depending on whether the image file is writable or whether a writing user is attached to the node.

`force-share` Override the image locking system of QEMU by forcing the node to utilize weaker shared access for permissions where it would normally request exclusive access. When there is the potential for multiple instances to have the same file open (whether this invocation of QEMU is the first or the second instance), both instances must permit shared access for the second instance to succeed at opening the file.

Enabling `force-share=on` requires `read-only=on`.

`cache.direct` The host page cache can be avoided with `cache.direct=on`. This will attempt to do disk IO directly to the guest's memory. QEMU may still perform an internal copy of the data.

`cache.no-flush` In case you don't care about data integrity over host failures, you can use `cache.no-flush=on`. This option tells QEMU that it never needs to write any data to the disk but can instead keep things in cache. If anything goes wrong, like your host losing power, the disk storage getting disconnected accidentally, etc. your image will most probably be rendered unusable.

`discard=discard` discard is one of "ignore" (or "off") or "unmap" (or "on") and controls whether `discard` (also known as `trim` or `unmap`) requests are ignored or passed to the filesystem. Some machine types may not support discard requests.

`detect-zeroes=detect-zeroes` detect-zeroes is "off", "on" or "unmap" and enables the automatic conversion of plain zero writes by the OS to driver specific optimized zero write commands. You may even choose "unmap" if discard is set to "unmap" to allow a zero write to be converted to an `unmap` operation.

`Driver-specific options for file` This is the protocol-level block driver for accessing regular files.

`filename` The path to the image file in the local filesystem

`aio` Specifies the AIO backend (threads/native, default: threads)

locking Specifies whether the image file is protected with Linux OFD / POSIX locks. The default is to use the Linux Open File Descriptor API if available, otherwise no lock is applied. (auto/on/off, default: auto)

Example:

```
-blockdev driver=file,node-name=disk,filename=disk.img
```

Driver-specific options for raw This is the image format block driver for raw images. It is usually stacked on top of a protocol level block driver such as `file`.

file Reference to or definition of the data source block driver node (e.g. a `file` driver node)

Example 1:

```
-blockdev driver=file,node-name=disk_file,filename=disk.img
-blockdev driver=raw,node-name=disk,file=disk_file
```

Example 2:

```
-blockdev driver=raw,node-name=disk,file.driver=file,file.filename=disk.img
```

Driver-specific options for qcow2 This is the image format block driver for qcow2 images. It is usually stacked on top of a protocol level block driver such as `file`.

file Reference to or definition of the data source block driver node (e.g. a `file` driver node)

backing Reference to or definition of the backing file block device (default is taken from the image file). It is allowed to pass `null` here in order to disable the default backing file.

lazy-refcounts Whether to enable the lazy refcounts feature (on/off; default is taken from the image file)

cache-size The maximum total size of the L2 table and refcount block caches in bytes (default: the sum of l2-cache-size and refcount-cache-size)

l2-cache-size The maximum size of the L2 table cache in bytes (default: if cache-size is not specified - 32M on Linux platforms, and 8M on non-Linux platforms; otherwise, as large as possible within the cache-size, while permitting the requested or the minimal refcount cache size)

refcount-cache-size The maximum size of the refcount block cache in bytes (default: 4 times the cluster size; or if cache-size is specified, the part of it which is not used for the L2 cache)

cache-clean-interval Clean unused entries in the L2 and refcount caches. The interval is in seconds. The default value is 600 on supporting platforms, and 0 on other platforms. Setting it to 0 disables this feature.

pass-discard-request Whether discard requests to the qcow2 device should be forwarded to the data source (on/off; default: on if discard=unmap is specified, off otherwise)

pass-discard-snapshot Whether discard requests for the data source should be issued when a snapshot operation (e.g. deleting a snapshot) frees clusters in the qcow2 file (on/off; default: on)

pass-discard-other Whether discard requests for the data source should be issued on other occasions where a cluster gets freed (on/off; default: off)

overlap-check Which overlap checks to perform for writes to the image (none/constant/cached/all; default: cached). For details or finer granularity control refer to the QAPI documentation of `blockdev-add`.

Example 1:

```
-blockdev driver=file,node-name=my_file,filename=/tmp/disk.qcow2
-blockdev driver=qcow2,node-name=hda,file=my_file,overlap-check=none,cache-
↪size=16777216
```

Example 2:

```
-blockdev driver=qcow2,node-name=disk,file.driver=http,file.filename=http://
↪example.com/image.qcow2
```

`Driver-specific options for other drivers` Please refer to the QAPI documentation of the `blockdev-add` QMP command.

`-drive option[,option[,option[,...]]]` Define a new drive. This includes creating a block driver node (the backend) as well as a guest device, and is mostly a shortcut for defining the corresponding `-blockdev` and `-device` options.

`-drive` accepts all options that are accepted by `-blockdev`. In addition, it knows the following options:

`file=file` This option defines which disk image (see *Disk Images*) to use with this drive. If the filename contains comma, you must double it (for instance, “file=my,,file” to use file “my,file”).

Special files such as iSCSI devices can be specified using protocol specific URLs. See the section for “Device URL Syntax” for more information.

`if=interface` This option defines on which type on interface the drive is connected. Available types are: ide, scsi, sd, mtd, floppy, pflash, virtio, none.

`bus=bus,unit=unit` These options define where is connected the drive by defining the bus number and the unit id.

`index=index` This option defines where is connected the drive by using an index in the list of available connectors of a given interface type.

`media=media` This option defines the type of the media: disk or cdrom.

`snapshot=snapshot` snapshot is “on” or “off” and controls snapshot mode for the given drive (see `-snapshot`).

`cache=cache` cache is “none”, “writeback”, “unsafe”, “directsync” or “writethrough” and controls how the host cache is used to access block data. This is a shortcut that sets the `cache.direct` and `cache.no-flush` options (as in `-blockdev`), and additionally `cache.writeback`, which provides a default for the `write-cache` option of block guest devices (as in `-device`). The modes correspond to the following settings:

	cache.writeback	cache.direct	cache.no-flush
writeback	on	off	off
none	on	on	off
writethrough	off	off	off
directsync	off	on	off
unsafe	on	off	on

The default mode is `cache=writeback`.

`aio=aio` aio is “threads”, or “native” and selects between pthread based disk I/O and native Linux AIO.

`format=format` Specify which disk format will be used rather than detecting the format. Can be used to specify format=raw to avoid interpreting an untrusted format header.

`werror=action,rerror=action` Specify which action to take on write and read errors. Valid actions are: “ignore” (ignore the error and try to continue), “stop” (pause QEMU), “report” (report the error to the guest), "enospc" (pause QEMU only if the host disk is full; report the error to the guest otherwise). The default setting is `werror=enospc` and `rerror=report`.

`copy-on-read=copy-on-read` copy-on-read is "on" or "off" and enables whether to copy read backing file sectors into the image file.

`bps=b,bps_rd=r,bps_wr=w` Specify bandwidth throttling limits in bytes per second, either for all request types or for reads or writes only. Small values can lead to timeouts or hangs inside the guest. A safe minimum for disks is 2 MB/s.

`bps_max=bm,bps_rd_max=rm,bps_wr_max=wm` Specify bursts in bytes per second, either for all request types or for reads or writes only. Bursts allow the guest I/O to spike above the limit temporarily.

`iops=i,iops_rd=r,iops_wr=w` Specify request rate limits in requests per second, either for all request types or for reads or writes only.

`iops_max=bm,iops_rd_max=rm,iops_wr_max=wm` Specify bursts in requests per second, either for all request types or for reads or writes only. Bursts allow the guest I/O to spike above the limit temporarily.

`iops_size=is` Let every is bytes of a request count as a new request for iops throttling purposes. Use this option to prevent guests from circumventing iops limits by sending fewer but larger requests.

`group=g` Join a throttling quota group with given name g. All drives that are members of the same group are accounted for together. Use this option to prevent guests from circumventing throttling limits by using many small disks instead of a single larger disk.

By default, the `cache.writeback=on` mode is used. It will report data writes as completed as soon as the data is present in the host page cache. This is safe as long as your guest OS makes sure to correctly flush disk caches where needed. If your guest OS does not handle volatile disk write caches correctly and your host crashes or loses power, then the guest may experience data corruption.

For such guests, you should consider using `cache.writeback=off`. This means that the host page cache will be used to read and write data, but write notification will be sent to the guest only after QEMU has made sure to flush each write to the disk. Be aware that this has a major impact on performance.

When using the `-snapshot` option, unsafe caching is always used.

Copy-on-read avoids accessing the same backing file sectors repeatedly and is useful when the backing file is over a slow network. By default copy-on-read is off.

Instead of `-cdrom` you can use:

```
qemu-system-x86_64 -drive file=file,index=2,media=cdrom
```

Instead of `-hda`, `-hdb`, `-hdc`, `-hdd`, you can use:

```
qemu-system-x86_64 -drive file=file,index=0,media=disk
qemu-system-x86_64 -drive file=file,index=1,media=disk
qemu-system-x86_64 -drive file=file,index=2,media=disk
qemu-system-x86_64 -drive file=file,index=3,media=disk
```

You can open an image using pre-opened file descriptors from an fd set:

```
qemu-system-x86_64  -add-fd fd=3,set=2,opaque="rdwr:/path/to/file"  -add-
↪fd fd=4,set=2,opaque="rdonly:/path/to/file"  -drive file=/dev/fdset/2,
↪index=0,media=disk
```

You can connect a CDROM to the slave of ide0:

```
qemu_system-x86_64 -drive file=file,if=ide,index=1,media=cdrom
```

If you don't specify the "file=" argument, you define an empty drive:

```
qemu_system-x86_64 -drive if=ide,index=1,media=cdrom
```

Instead of `-fda`, `-fdb`, you can use:

```
qemu_system-x86_64 -drive file=file,index=0,if=floppy
qemu_system-x86_64 -drive file=file,index=1,if=floppy
```

By default, interface is "ide" and index is automatically incremented:

```
qemu_system-x86_64 -drive file=a -drive file=b"
```

is interpreted like:

```
qemu_system-x86_64 -hda a -hdb b
```

`-mtdblock file` Use file as on-board Flash memory image.

`-sd file` Use file as SecureDigital card image.

`-pflash file` Use file as a parallel flash image.

`-snapshot` Write to temporary files instead of disk image files. In this case, the raw disk image you use is not written back. You can however force the write back by pressing C-a s (see *Disk Images*).

`-fsdev local,id=id,path=path,security_model=security_model [,writeout=writeout][,readonly]`

`-fsdev proxy,id=id,socket=socket[,writeout=writeout][,readonly]`

`-fsdev proxy,id=id,sock_fd=sock_fd[,writeout=writeout][,readonly]`

`-fsdev synth,id=id[,readonly]` Define a new file system device. Valid options are:

`local` Accesses to the filesystem are done by QEMU.

`proxy` Accesses to the filesystem are done by virtfs-proxy-helper(1).

`synth` Synthetic filesystem, only used by QTests.

`id=id` Specifies identifier for this device.

`path=path` Specifies the export path for the file system device. Files under this path will be available to the 9p client on the guest.

`security_model=security_model` Specifies the security model to be used for this export path. Supported security models are "passthrough", "mapped-xattr", "mapped-file" and "none". In "passthrough" security model, files are stored using the same credentials as they are created on the guest. This requires QEMU to run as root. In "mapped-xattr" security model, some of the file attributes like uid, gid, mode bits and link target are stored as file attributes. For "mapped-file" these attributes are stored in the hidden .virtfs_metadata directory. Directories exported by this security model cannot interact with other unix tools. "none" security model is same as passthrough except the sever won't report failures if it fails to set file attributes like ownership. Security model is mandatory only for local fsdriver. Other fsdrivers (like proxy) don't take security model as a parameter.

`writeout=writeout` This is an optional argument. The only supported value is "immediate". This means that host page cache will be used to read and write data but write notification will be sent to the guest only when the data has been reported as written by the storage subsystem.

`readonly` Enables exporting 9p share as a readonly mount for guests. By default read-write access is given.

`socket=socket` Enables proxy filesystem driver to use passed socket file for communicating with virtfs-proxy-helper(1).

`sock_fd=sock_fd` Enables proxy filesystem driver to use passed socket descriptor for communicating with virtfs-proxy-helper(1). Usually a helper like libvirt will create socketpair and pass one of the fds as sock_fd.

fmode=fmode Specifies the default mode for newly created files on the host. Works only with security models "mapped-xattr" and "mapped-file".

dmode=dmode Specifies the default mode for newly created directories on the host. Works only with security models "mapped-xattr" and "mapped-file".

throttling.bps-total=b,throttling.bps-read=r,throttling.bps-write=w Specify bandwidth throttling limits in bytes per second, either for all request types or for reads or writes only.

throttling.bps-total-max=bm,bps-read-max=rm,bps-write-max=wm Specify bursts in bytes per second, either for all request types or for reads or writes only. Bursts allow the guest I/O to spike above the limit temporarily.

throttling.iops-total=i,throttling.iops-read=r, throttling.iops-write=w Specify request rate limits in requests per second, either for all request types or for reads or writes only.

throttling.iops-total-max=im,throttling.iops-read-max=irm, throttling.iops-write-max=i Specify bursts in requests per second, either for all request types or for reads or writes only. Bursts allow the guest I/O to spike above the limit temporarily.

throttling.iops-size=is Let every is bytes of a request count as a new request for iops throttling purposes.

-fsdev option is used along with -device driver "virtio-9p-. . . ".

-device virtio-9p-type,fsdev=id,mount_tag=mount_tag Options for virtio-9p-. . . driver are:

type Specifies the variant to be used. Supported values are "pci", "ccw" or "device", depending on the machine type.

fsdev=id Specifies the id value specified along with -fsdev option.

mount_tag=mount_tag Specifies the tag name to be used by the guest to mount this export point.

-virtfs local,path=path,mount_tag=mount_tag ,security_model=security_model[,writeout=write

-virtfs proxy,socket=socket,mount_tag=mount_tag [,writeout=writeout][,readonly]

-virtfs proxy,sock_fd=sock_fd,mount_tag=mount_tag [,writeout=writeout][,readonly]

-virtfs synth,mount_tag=mount_tag Define a new virtual filesystem device and expose it to the guest using a virtio-9p-device (a.k.a. 9pfs), which essentially means that a certain directory on host is made directly accessible by guest as a pass-through file system by using the 9P network protocol for communication between host and guests, if desired even accessible, shared by several guests simultaniously.

Note that `-virtfs` is actually just a convenience shortcut for its generalized form `-fsdev -device virtio-9p-pci`.

The general form of pass-through file system options are:

local Accesses to the filesystem are done by QEMU.

proxy Accesses to the filesystem are done by virtfs-proxy-helper(1).

synth Synthetic filesystem, only used by QTests.

id=id Specifies identifier for the filesystem device

path=path Specifies the export path for the file system device. Files under this path will be available to the 9p client on the guest.

security_model=security_model Specifies the security model to be used for this export path. Supported security models are "passthrough", "mapped-xattr", "mapped-file" and "none". In "passthrough" security model, files are stored using the same credentials as they are created on the guest. This requires QEMU to run as root. In "mapped-xattr" security model, some of the file attributes like uid, gid, mode bits and link target are stored as file attributes. For "mapped-file" these attributes are stored in the hidden .virtfs_metadata directory. Directories exported by this security model cannot interact with other unix tools. "none" security model is same as passthrough except the sever won't report failures if it fails to set file attributes like ownership. Security model is mandatory only for local fsdriver. Other fsdrivers (like proxy) don't take security model as a parameter.

writeout=writeout This is an optional argument. The only supported value is "immediate". This means that host page cache will be used to read and write data but write notification will be sent to the guest only when the data has been reported as written by the storage subsystem.

readonly Enables exporting 9p share as a readonly mount for guests. By default read-write access is given.

socket=socket Enables proxy filesystem driver to use passed socket file for communicating with virtfs-proxy-helper(1). Usually a helper like libvirt will create socketpair and pass one of the fds as sock_fd.

sock_fd Enables proxy filesystem driver to use passed 'sock_fd' as the socket descriptor for interfacing with virtfs-proxy-helper(1).

fmode=fmode Specifies the default mode for newly created files on the host. Works only with security models "mapped-xattr" and "mapped-file".

dmode=dmode Specifies the default mode for newly created directories on the host. Works only with security models "mapped-xattr" and "mapped-file".

mount_tag=mount_tag Specifies the tag name to be used by the guest to mount this export point.

multidevs=multidevs Specifies how to deal with multiple devices being shared with a 9p export. Supported behaviours are either "remap", "forbid" or "warn". The latter is the default behaviour on which virtfs 9p expects only one device to be shared with the same export, and if more than one device is shared and accessed via the same 9p export then only a warning message is logged (once) by qemu on host side. In order to avoid file ID collisions on guest you should either create a separate virtfs export for each device to be shared with guests (recommended way) or you might use "remap" instead which allows you to share multiple devices with only one export instead, which is achieved by remapping the original inode numbers from host to guest in a way that would prevent such collisions. Remapping inodes in such use cases is required because the original device IDs from host are never passed and exposed on guest. Instead all files of an export shared with virtfs always share the same device id on guest. So two files with identical inode numbers but from actually different devices on host would otherwise cause a file ID collision and hence potential misbehaviours on guest. "forbid" on the other hand assumes like "warn" that only one device is shared by the same export, however it will not only log a warning message but also deny access to additional devices on guest. Note though that "forbid" does currently not block all possible file access operations (e.g. readdir() would still return entries from other devices).

-iscsi Configure iSCSI session parameters.

1.2.3 USB options

-usb Enable USB emulation on machine types with an on-board USB host controller (if not enabled by default). Note that on-board USB host controllers may not support USB 3.0. In this case `-device qemu-xhci` can be used instead on machines with PCI.

-usbdevice devname Add the USB device devname. Note that this option is deprecated, please use `-device usb-...` instead. See *Connecting USB devices*.

mouse Virtual Mouse. This will override the PS/2 mouse emulation when activated.

tablet Pointer device that uses absolute coordinates (like a touchscreen). This means QEMU is able to report the mouse position without having to grab the mouse. Also overrides the PS/2 mouse emulation when activated.

braille Braille device. This will use BrlAPI to display the braille output on a real or fake device.

1.2.4 Display options

-display type Select type of display to use. This option is a replacement for the old style -sdl/-curses/... options. Use `-display help` to list the available display types. Valid values for type are

sdl Display video output via SDL (usually in a separate graphics window; see the SDL documentation for other possibilities).

curses Display video output via curses. For graphics device models which support a text mode, QEMU can display this output using a curses/ncurses interface. Nothing is displayed when the graphics device is in graphical mode or if the graphics device does not support a text mode. Generally only the VGA device models support text mode. The font charset used by the guest can be specified with the `charset` option, for example `charset=CP850` for IBM CP850 encoding. The default is `CP437`.

none Do not display video output. The guest will still see an emulated graphics card, but its output will not be displayed to the QEMU user. This option differs from the -nographic option in that it only affects what is done with video output; -nographic also changes the destination of the serial and parallel port data.

gtk Display video output in a GTK window. This interface provides drop-down menus and other UI elements to configure and control the VM during runtime.

vnc Start a VNC server on display <arg>

egl-headless Offload all OpenGL operations to a local DRI device. For any graphical display, this display needs to be paired with either VNC or SPICE displays.

spice-app Start QEMU as a Spice server and launch the default Spice client application. The Spice server will redirect the serial consoles and QEMU monitors. (Since 4.0)

-nographic Normally, if QEMU is compiled with graphical window support, it displays output such as guest graphics, guest console, and the QEMU monitor in a window. With this option, you can totally disable graphical output so that QEMU is a simple command line application. The emulated serial port is redirected on the console and muxed with the monitor (unless redirected elsewhere explicitly). Therefore, you can still use QEMU to debug a Linux kernel with a serial console. Use C-a h for help on switching between the console and monitor.

-curses Normally, if QEMU is compiled with graphical window support, it displays output such as guest graphics, guest console, and the QEMU monitor in a window. With this option, QEMU can display the VGA output when in text mode using a curses/ncurses interface. Nothing is displayed in graphical mode.

-alt-grab Use Ctrl-Alt-Shift to grab mouse (instead of Ctrl-Alt). Note that this also affects the special keys (for fullscreen, monitor-mode switching, etc).

-ctrl-grab Use Right-Ctrl to grab mouse (instead of Ctrl-Alt). Note that this also affects the special keys (for fullscreen, monitor-mode switching, etc).

-no-quit Disable SDL window close capability.

-sdl Enable SDL.

-spice option[,option[,...]] Enable the spice remote desktop protocol. Valid options are

port=<nr> Set the TCP port spice is listening on for plaintext channels.

addr=<addr> Set the IP address spice is listening on. Default is any address.

ipv4; ipv6; unix Force using the specified IP version.

password=<secret> Set the password you need to authenticate.

sasl Require that the client use SASL to authenticate with the spice. The exact choice of authentication method used is controlled from the system / user's SASL configuration file for the 'qemu' service. This is typically found in /etc/sasl2/qemu.conf. If running QEMU as an unprivileged user, an environment variable SASL_CONF_PATH can be used to make it search alternate locations for the service config. While some SASL auth methods can also provide data encryption (eg GSSAPI), it is recommended that SASL always be combined with the 'tls' and 'x509' settings to enable use of SSL and server certificates. This ensures a data encryption preventing compromise of authentication credentials.

disable-ticketing Allow client connects without authentication.

disable-copy-paste Disable copy paste between the client and the guest.

disable-agent-file-xfer Disable spice-vdagent based file-xfer between the client and the guest.

tls-port=<nr> Set the TCP port spice is listening on for encrypted channels.

x509-dir=<dir> Set the x509 file directory. Expects same filenames as -vnc $display,x509=$dir

x509-key-file=<file>; x509-key-password=<file>; x509-cert-file=<file>; x509-cacert-file=<
The x509 file names can also be configured individually.

tls-ciphers=<list> Specify which ciphers to use.

tls-channel=[main|display|cursor|inputs|record|playback]; plaintext-channel=[main|displa
Force specific channel to be used with or without TLS encryption. The options can be specified multiple times to configure multiple channels. The special name "default" can be used to set the default mode. For channels which are not explicitly forced into one mode the spice client is allowed to pick tls/plaintext as he pleases.

image-compression=[auto_glz|auto_lz|quic|glz|lz|off] Configure image compression (lossless). Default is auto_glz.

jpeg-wan-compression=[auto|never|always]; zlib-glz-wan-compression=[auto|never|always]
Configure wan image compression (lossy for slow links). Default is auto.

streaming-video=[off|all|filter] Configure video stream detection. Default is off.

agent-mouse=[on|off] Enable/disable passing mouse events via vdagent. Default is on.

playback-compression=[on|off] Enable/disable audio stream compression (using celt 0.5.1). Default is on.

seamless-migration=[on|off] Enable/disable spice seamless migration. Default is off.

gl=[on|off] Enable/disable OpenGL context. Default is off.

rendernode=<file> DRM render node for OpenGL rendering. If not specified, it will pick the first available. (Since 2.9)

-portrait Rotate graphical output 90 deg left (only PXA LCD).

-rotate deg Rotate graphical output some deg left (only PXA LCD).

-vga type Select type of VGA card to emulate. Valid values for type are

cirrus Cirrus Logic GD5446 Video card. All Windows versions starting from Windows 95 should recognize and use this graphic card. For optimal performances, use 16 bit color depth in the guest and the host OS. (This card was the default before QEMU 2.2)

std Standard VGA card with Bochs VBE extensions. If your guest OS supports the VESA 2.0 VBE extensions (e.g. Windows XP) and if you want to use high resolution modes (>= 1280x1024x16) then you should use this option. (This card is the default since QEMU 2.2)

vmware VMWare SVGA-II compatible adapter. Use it if you have sufficiently recent XFree86/XOrg server or Windows guest with a driver for this card.

qxl QXL paravirtual graphic card. It is VGA compatible (including VESA 2.0 VBE support). Works best with qxl guest drivers installed though. Recommended choice when using the spice protocol.

tcx (sun4m only) Sun TCX framebuffer. This is the default framebuffer for sun4m machines and offers both 8-bit and 24-bit colour depths at a fixed resolution of 1024x768.

cg3 (sun4m only) Sun cgthree framebuffer. This is a simple 8-bit framebuffer for sun4m machines available in both 1024x768 (OpenBIOS) and 1152x900 (OBP) resolutions aimed at people wishing to run older Solaris versions.

virtio Virtio VGA card.

none Disable VGA card.

-full-screen Start in full screen.

-g ***widthxheight*** **[*xdepth*]** Set the initial graphical resolution and depth (PPC, SPARC only).

For PPC the default is 800x600x32.

For SPARC with the TCX graphics device, the default is 1024x768x8 with the option of 1024x768x24. For cgthree, the default is 1024x768x8 with the option of 1152x900x8 for people who wish to use OBP.

-vnc display[,option[,option[,...]]] Normally, if QEMU is compiled with graphical window support, it displays output such as guest graphics, guest console, and the QEMU monitor in a window. With this option, you can have QEMU listen on VNC display display and redirect the VGA display over the VNC session. It is very useful to enable the usb tablet device when using this option (option `-device usb-tablet`). When using the VNC display, you must use the `-k` parameter to set the keyboard layout if you are not using en-us. Valid syntax for the display is

to=L With this option, QEMU will try next available VNC displays, until the number L, if the origianlly defined “-vnc display” is not available, e.g. port 5900+display is already used by another application. By default, to=0.

host:d TCP connections will only be allowed from host on display d. By convention the TCP port is 5900+d. Optionally, host can be omitted in which case the server will accept connections from any host.

unix:path Connections will be allowed over UNIX domain sockets where path is the location of a unix socket to listen for connections on.

none VNC is initialized but not started. The monitor `change` command can be used to later start the VNC server.

Following the display value there may be one or more option flags separated by commas. Valid options are

reverse Connect to a listening VNC client via a “reverse” connection. The client is specified by the display. For reverse network connections (host:d,‘‘reverse‘‘), the d argument is a TCP port number, not a display number.

websocket Opens an additional TCP listening port dedicated to VNC Websocket connections. If a bare websocket option is given, the Websocket port is 5700+display. An alternative port can be specified with the syntax `websocket`=port.

If host is specified connections will only be allowed from this host. It is possible to control the websocket listen address independently, using the syntax `websocket`=host:port.

If no TLS credentials are provided, the websocket connection runs in unencrypted mode. If TLS credentials are provided, the websocket connection requires encrypted client connections.

password Require that password based authentication is used for client connections.

The password must be set separately using the `set_password` command in the *QEMU Monitor*. The syntax to change your password is: `set_password <protocol> <password>` where <protocol> could be either "vnc" or "spice".

If you would like to change <protocol> password expiration, you should use `expire_password <protocol> <expiration-time>` where expiration time could be one of the following options: now, never, +seconds or UNIX time of expiration, e.g. +60 to make password expire in 60 seconds, or 1335196800 to make password expire on "Mon Apr 23 12:00:00 EDT 2012" (UNIX time for this date and time).

You can also use keywords "now" or "never" for the expiration time to allow <protocol> password to expire immediately or never expire.

`tls-creds=ID` Provides the ID of a set of TLS credentials to use to secure the VNC server. They will apply to both the normal VNC server socket and the websocket socket (if enabled). Setting TLS credentials will cause the VNC server socket to enable the VeNCrypt auth mechanism. The credentials should have been previously created using the `-object tls-creds` argument.

`tls-authz=ID` Provides the ID of the QAuthZ authorization object against which the client's x509 distinguished name will validated. This object is only resolved at time of use, so can be deleted and recreated on the fly while the VNC server is active. If missing, it will default to denying access.

`sasl` Require that the client use SASL to authenticate with the VNC server. The exact choice of authentication method used is controlled from the system / user's SASL configuration file for the 'qemu' service. This is typically found in /etc/sasl2/qemu.conf. If running QEMU as an unprivileged user, an environment variable SASL_CONF_PATH can be used to make it search alternate locations for the service config. While some SASL auth methods can also provide data encryption (eg GSSAPI), it is recommended that SASL always be combined with the 'tls' and 'x509' settings to enable use of SSL and server certificates. This ensures a data encryption preventing compromise of authentication credentials. See the *VNC security* section for details on using SASL authentication.

`sasl-authz=ID` Provides the ID of the QAuthZ authorization object against which the client's SASL username will validated. This object is only resolved at time of use, so can be deleted and recreated on the fly while the VNC server is active. If missing, it will default to denying access.

`acl` Legacy method for enabling authorization of clients against the x509 distinguished name and SASL username. It results in the creation of two `authz-list` objects with IDs of `vnc.username` and `vnc.x509dname`. The rules for these objects must be configured with the HMP ACL commands.

This option is deprecated and should no longer be used. The new `sasl-authz` and `tls-authz` options are a replacement.

`lossy` Enable lossy compression methods (gradient, JPEG, . . .). If this option is set, VNC client may receive lossy framebuffer updates depending on its encoding settings. Enabling this option can save a lot of bandwidth at the expense of quality.

`non-adaptive` Disable adaptive encodings. Adaptive encodings are enabled by default. An adaptive encoding will try to detect frequently updated screen regions, and send updates in these regions using a lossy encoding (like JPEG). This can be really helpful to save bandwidth when playing videos. Disabling adaptive encodings restores the original static behavior of encodings like Tight.

`share=[allow-exclusive|force-shared|ignore]` Set display sharing policy. 'allow-exclusive' allows clients to ask for exclusive access. As suggested by the rfb spec this is implemented by dropping other connections. Connecting multiple clients in parallel requires all clients asking for a shared session (vncviewer: -shared switch). This is the default. 'force-shared' disables exclusive client access. Useful for shared desktop sessions, where you don't want someone forgetting specify -shared disconnect everybody else. 'ignore' completely ignores the shared flag and allows everybody connect unconditionally. Doesn't conform to the rfb spec but is traditional QEMU behavior.

key-delay-ms Set keyboard delay, for key down and key up events, in milliseconds. Default is 10. Keyboards are low-bandwidth devices, so this slowdown can help the device and guest to keep up and not lose events in case events are arriving in bulk. Possible causes for the latter are flaky network connections, or scripts for automated testing.

audiodev=audiodev Use the specified audiodev when the VNC client requests audio transmission. When not using an -audiodev argument, this option must be omitted, otherwise is must be present and specify a valid audiodev.

1.2.5 i386 target only

-win2k-hack Use it when installing Windows 2000 to avoid a disk full bug. After Windows 2000 is installed, you no longer need this option (this option slows down the IDE transfers).

-no-fd-bootchk Disable boot signature checking for floppy disks in BIOS. May be needed to boot from old floppy disks.

-no-acpi Disable ACPI (Advanced Configuration and Power Interface) support. Use it if your guest OS complains about ACPI problems (PC target machine only).

-no-hpet Disable HPET support.

-acpitable [sig=str][,rev=n][,oem_id=str][,oem_table_id=str][,oem_rev=n] [,asl_compiler_id=
Add ACPI table with specified header fields and context from specified files. For file=, take whole ACPI table from the specified files, including all ACPI headers (possible overridden by other options). For data=, only data portion of the table is used, all header information is specified in the command line. If a SLIC table is supplied to QEMU, then the SLIC's oem_id and oem_table_id fields will override the same in the RSDT and the FADT (a.k.a. FACP), in order to ensure the field matches required by the Microsoft SLIC spec and the ACPI spec.

-smbios file=binary Load SMBIOS entry from binary file.

-smbios type=0[,vendor=str][,version=str][,date=str][,release=%d.%d][,uefi=on|off]
Specify SMBIOS type 0 fields

-smbios type=1[,manufacturer=str][,product=str][,version=str][,serial=str][,uuid=uuid][,sku
Specify SMBIOS type 1 fields

-smbios type=2[,manufacturer=str][,product=str][,version=str][,serial=str][,asset=str][,loc
Specify SMBIOS type 2 fields

-smbios type=3[,manufacturer=str][,version=str][,serial=str][,asset=str][,sku=str]
Specify SMBIOS type 3 fields

-smbios type=4[,sock_pfx=str][,manufacturer=str][,version=str][,serial=str][,asset=str][,pa
Specify SMBIOS type 4 fields

-smbios type=17[,loc_pfx=str][,bank=str][,manufacturer=str][,serial=str][,asset=str][,part=
Specify SMBIOS type 17 fields

1.2.6 Network options

-nic [tap|bridge|user|l2tpv3|vde|netmap|vhost-user|socket][,...][,mac=macaddr][,model=mn]
This option is a shortcut for configuring both the on-board (default) guest NIC hardware and the host network backend in one go. The host backend options are the same as with the corresponding `-netdev` options below. The guest NIC model can be set with `model=modelname`. Use `model=help` to list the available device types. The hardware MAC address can be set with `mac=macaddr`.

The following two example do exactly the same, to show how `-nic` can be used to shorten the command line length:

```
qemu-system-x86_64 -netdev user,id=n1,ipv6=off -device e1000,netdev=n1,
↪mac=52:54:98:76:54:32
qemu-system-x86_64 -nic user,ipv6=off,model=e1000,mac=52:54:98:76:54:32
```

-nic none Indicate that no network devices should be configured. It is used to override the default configuration (default NIC with "user" host network backend) which is activated if no other networking options are provided.

-netdev user,id=id[,option][,option][,...] Configure user mode host network backend which requires no administrator privilege to run. Valid options are:

id=id Assign symbolic name for use in monitor commands.

ipv4=on|off and ipv6=on|off Specify that either IPv4 or IPv6 must be enabled. If neither is specified both protocols are enabled.

net=addr[/mask] Set IP network address the guest will see. Optionally specify the netmask, either in the form a.b.c.d or as number of valid top-most bits. Default is 10.0.2.0/24.

host=addr Specify the guest-visible address of the host. Default is the 2nd IP in the guest network, i.e. x.x.x.2.

ipv6-net=addr[/int] Set IPv6 network address the guest will see (default is fec0::/64). The network prefix is given in the usual hexadecimal IPv6 address notation. The prefix size is optional, and is given as the number of valid top-most bits (default is 64).

ipv6-host=addr Specify the guest-visible IPv6 address of the host. Default is the 2nd IPv6 in the guest network, i.e. xxxx::2.

restrict=on|off If this option is enabled, the guest will be isolated, i.e. it will not be able to contact the host and no guest IP packets will be routed over the host to the outside. This option does not affect any explicitly set forwarding rules.

hostname=name Specifies the client hostname reported by the built-in DHCP server.

dhcpstart=addr Specify the first of the 16 IPs the built-in DHCP server can assign. Default is the 15th to 31st IP in the guest network, i.e. x.x.x.15 to x.x.x.31.

dns=addr Specify the guest-visible address of the virtual nameserver. The address must be different from the host address. Default is the 3rd IP in the guest network, i.e. x.x.x.3.

ipv6-dns=addr Specify the guest-visible address of the IPv6 virtual nameserver. The address must be different from the host address. Default is the 3rd IP in the guest network, i.e. xxxx::3.

dnssearch=domain Provides an entry for the domain-search list sent by the built-in DHCP server. More than one domain suffix can be transmitted by specifying this option multiple times. If supported, this will cause the guest to automatically try to append the given domain suffix(es) in case a domain name can not be resolved.

Example:

```
qemu-system-x86_64 -nic user,dnssearch=mgmt.example.org,
↪dnssearch=example.org
```

domainname=domain Specifies the client domain name reported by the built-in DHCP server.

tftp=dir When using the user mode network stack, activate a built-in TFTP server. The files in dir will be exposed as the root of a TFTP server. The TFTP client on the guest must be configured in binary mode (use the command `bin` of the Unix TFTP client).

tftp-server-name=name In BOOTP reply, broadcast name as the "TFTP server name" (RFC2132 option 66). This can be used to advise the guest to load boot files or configurations from a different server than the host address.

bootfile=file When using the user mode network stack, broadcast file as the BOOTP filename. In conjunction with `tftp`, this can be used to network boot a guest from a local directory.

Example (using pxelinux):

```
qemu-system-x86_64 -hda linux.img -boot n -device e1000,netdev=n1 ␣
↪-netdev user,id=n1,tftp=/path/to/tftp/files,bootfile=/pxelinux.0
```

smb=dir[,smbserver=addr] When using the user mode network stack, activate a built-in SMB server so that Windows OSes can access to the host files in `dir` transparently. The IP address of the SMB server can be set to addr. By default the 4th IP in the guest network is used, i.e. x.x.x.4.

In the guest Windows OS, the line:

```
10.0.2.4 smbserver
```

must be added in the file `C:\WINDOWS\LMHOSTS` (for windows 9x/Me) or `C:\WINNT\SYSTEM32\DRIVERS\ETC\LMHOSTS` (Windows NT/2000).

Then `dir` can be accessed in `\\smbserver\qemu`.

Note that a SAMBA server must be installed on the host OS.

hostfwd=[tcp|udp]:[hostaddr]:hostport-[guestaddr]:guestport Redirect incoming TCP or UDP connections to the host port hostport to the guest IP address guestaddr on guest port guestport. If guestaddr is not specified, its value is x.x.x.15 (default first address given by the built-in DHCP server). By specifying hostaddr, the rule can be bound to a specific host interface. If no connection type is set, TCP is used. This option can be given multiple times.

For example, to redirect host X11 connection from screen 1 to guest screen 0, use the following:

```
# on the host
qemu-system-x86_64 -nic user,hostfwd=tcp:127.0.0.1:6001-:6000
# this host xterm should open in the guest X11 server
xterm -display :1
```

To redirect telnet connections from host port 5555 to telnet port on the guest, use the following:

```
# on the host
qemu-system-x86_64 -nic user,hostfwd=tcp::5555-:23
telnet localhost 5555
```

Then when you use on the host `telnet localhost 5555`, you connect to the guest telnet server.

guestfwd=[tcp]:server:port-dev; guestfwd=[tcp]:server:port-cmd:command Forward guest TCP connections to the IP address server on port port to the character device dev or to a program executed by cmd:command which gets spawned for each connection. This option can be given multiple times.

You can either use a chardev directly and have that one used throughout QEMU's lifetime, like in the following example:

```
# open 10.10.1.1:4321 on bootup, connect 10.0.2.100:1234 to it␣
↪whenever
# the guest accesses it
qemu-system-x86_64 -nic user,guestfwd=tcp:10.0.2.100:1234-tcp:10.10.1.
1:4321
```

Or you can execute a command on every TCP connection established by the guest, so that QEMU behaves similar to an inetd process for that virtual server:

```
# call "netcat 10.10.1.1 4321" on every TCP connection to 10.0.2.
↪100:1234
```

```
    # and connect the TCP stream to its stdin/stdout
    qemu-system-x86_64 -nic  'user,id=n1,guestfwd=tcp:10.0.2.100:1234-
    ↪cmd:netcat 10.10.1.1 4321'
```

-netdev tap,id=id[,fd=h][,ifname=name][,script=file][,downscript=dfile][,br=bridge][,helpe
Configure a host TAP network backend with ID id.

Use the network script file to configure it and the network script dfile to deconfigure it. If name is not provided, the OS automatically provides one. The default network configure script is `/etc/qemu-ifup` and the default network deconfigure script is `/etc/qemu-ifdown`. Use `script=no` or `downscript=no` to disable script execution.

If running QEMU as an unprivileged user, use the network helper to configure the TAP interface and attach it to the bridge. The default network helper executable is `/path/to/qemu-bridge-helper` and the default bridge device is `br0`.

`fd`=h can be used to specify the handle of an already opened host TAP interface.

Examples:

```
#launch a QEMU instance with the default network script
qemu-system-x86_64 linux.img -nic tap
```

```
#launch a QEMU instance with two NICs, each one connected
#to a TAP device
qemu-system-x86_64 linux.img         -netdev tap,id=nd0,ifname=tap0 -
↪device e1000,netdev=nd0         -netdev tap,id=nd1,ifname=tap1 -device␣
↪rtl8139,netdev=nd1
```

```
#launch a QEMU instance with the default network helper to
#connect a TAP device to bridge br0
qemu-system-x86_64 linux.img -device virtio-net-pci,netdev=n1         -
↪netdev tap,id=n1,"helper=/path/to/qemu-bridge-helper"
```

-netdev bridge,id=id[,br=bridge][,helper=helper] Connect a host TAP network interface to a host bridge device.

Use the network helper helper to configure the TAP interface and attach it to the bridge. The default network helper executable is `/path/to/qemu-bridge-helper` and the default bridge device is `br0`.

Examples:

```
#launch a QEMU instance with the default network helper to
#connect a TAP device to bridge br0
qemu-system-x86_64 linux.img -netdev bridge,id=n1 -device virtio-net,
↪netdev=n1
```

```
#launch a QEMU instance with the default network helper to
#connect a TAP device to bridge qemubr0
qemu-system-x86_64 linux.img -netdev bridge,br=qemubr0,id=n1 -device␣
↪virtio-net,netdev=n1
```

-netdev socket,id=id[,fd=h][,listen=[host]:port][,connect=host:port] This host network backend can be used to connect the guest's network to another QEMU virtual machine using a TCP socket connection. If `listen` is specified, QEMU waits for incoming connections on port (host is optional). `connect` is used to connect to another QEMU instance using the `listen` option. `fd`=h specifies an already opened TCP socket.

Example:

```
# launch a first QEMU instance
```

```
qemu-system-x86_64 linux.img                    -device e1000,netdev=n1,
↪mac=52:54:00:12:34:56                    -netdev socket,id=n1,listen=:1234
# connect the network of this instance to the network of the first␣
↪instance
qemu-system-x86_64 linux.img                    -device e1000,netdev=n2,
↪mac=52:54:00:12:34:57                    -netdev socket,id=n2,connect=127.
↪0.0.1:1234
```

-netdev socket,id=id[,fd=h][,mcast=maddr:port[,localaddr=addr]] Configure a socket host network backend to share the guest's network traffic with another QEMU virtual machines using a UDP multicast socket, effectively making a bus for every QEMU with same multicast address maddr and port. NOTES:

1. Several QEMU can be running on different hosts and share same bus (assuming correct multicast setup for these hosts).
2. mcast support is compatible with User Mode Linux (argument `ethN=mcast`), see http://user-mode-linux.sf.net.
3. Use `fd=h` to specify an already opened UDP multicast socket.

Example:

```
# launch one QEMU instance
qemu-system-x86_64 linux.img                    -device e1000,netdev=n1,
↪mac=52:54:00:12:34:56                    -netdev socket,id=n1,mcast=230.0.
↪0.1:1234
# launch another QEMU instance on same "bus"
qemu-system-x86_64 linux.img                    -device e1000,netdev=n2,
↪mac=52:54:00:12:34:57                    -netdev socket,id=n2,mcast=230.0.
↪0.1:1234
# launch yet another QEMU instance on same "bus"
qemu-system-x86_64 linux.img                    -device e1000,netdev=n3,
↪mac=52:54:00:12:34:58                    -netdev socket,id=n3,mcast=230.0.
↪0.1:1234
```

Example (User Mode Linux compat.):

```
# launch QEMU instance (note mcast address selected is UML's default)
qemu-system-x86_64 linux.img                    -device e1000,netdev=n1,
↪mac=52:54:00:12:34:56                    -netdev socket,id=n1,mcast=239.
↪192.168.1:1102
# launch UML
/path/to/linux ubd0=/path/to/root_fs eth0=mcast
```

Example (send packets from host's 1.2.3.4):

```
qemu-system-x86_64 linux.img                    -device e1000,netdev=n1,
↪mac=52:54:00:12:34:56                    -netdev socket,id=n1,mcast=239.
↪192.168.1:1102,localaddr=1.2.3.4
```

-netdev l2tpv3,id=id,src=srcaddr,dst=dstaddr[,srcport=srcport][,dstport=dstport],txsession=
Configure a L2TPv3 pseudowire host network backend. L2TPv3 (RFC3931) is a popular protocol to transport Ethernet (and other Layer 2) data frames between two systems. It is present in routers, firewalls and the Linux kernel (from version 3.3 onwards).

This transport allows a VM to communicate to another VM, router or firewall directly.

src=srcaddr source address (mandatory)

dst=dstaddr destination address (mandatory)

udp select udp encapsulation (default is ip).

srcport=srcport source udp port.

dstport=dstport destination udp port.

ipv6 force v6, otherwise defaults to v4.

rxcookie=rxcookie; txcookie=txcookie Cookies are a weak form of security in the l2tpv3 specification. Their function is mostly to prevent misconfiguration. By default they are 32 bit.

cookie64 Set cookie size to 64 bit instead of the default 32

counter=off Force a 'cut-down' L2TPv3 with no counter as in draft-mkonstan-l2tpext-keyed-ipv6-tunnel-00

pincounter=on Work around broken counter handling in peer. This may also help on networks which have packet reorder.

offset=offset Add an extra offset between header and data

For example, to attach a VM running on host 4.3.2.1 via L2TPv3 to the bridge br-lan on the remote Linux host 1.2.3.4:

```
# Setup tunnel on linux host using raw ip as encapsulation
# on 1.2.3.4
ip l2tp add tunnel remote 4.3.2.1 local 1.2.3.4 tunnel_id 1 peer_tunnel_
↪id 1     encap udp udp_sport 16384 udp_dport 16384
ip l2tp add session tunnel_id 1 name vmtunnel0 session_id     0xFFFFFFFF␣
↪peer_session_id 0xFFFFFFFF
ifconfig vmtunnel0 mtu 1500
ifconfig vmtunnel0 up
brctl addif br-lan vmtunnel0


# on 4.3.2.1
# launch QEMU instance - if your network has reorder or is very lossy add␣
↪,pincounter

qemu-system-x86_64 linux.img -device e1000,netdev=n1     -netdev␣
↪l2tpv3,id=n1,src=4.2.3.1,dst=1.2.3.4,udp,srcport=16384,dstport=16384,
↪rxsession=0xffffffff,txsession=0xffffffff,counter
```

-netdev vde,id=id[,sock=socketpath][,port=n][,group=groupname][,mode=octalmode]
Configure VDE backend to connect to PORT n of a vde switch running on host and listening for incoming connections on socketpath. Use GROUP groupname and MODE octalmode to change default ownership and permissions for communication port. This option is only available if QEMU has been compiled with vde support enabled.

Example:

```
# launch vde switch
vde_switch -F -sock /tmp/myswitch
# launch QEMU instance
qemu-system-x86_64 linux.img -nic vde,sock=/tmp/myswitch
```

-netdev vhost-user,chardev=id[,vhostforce=on|off][,queues=n] Establish a vhost-user netdev, backed by a chardev id. The chardev should be a unix domain socket backed one. The vhost-user uses a specifically defined protocol to pass vhost ioctl replacement messages to an application on the other end of the socket. On non-MSIX guests, the feature can be forced with vhostforce. Use 'queues=n' to specify the number of queues to be created for multiqueue vhost-user.

Example:

```
qemu -m 512 -object memory-backend-file,id=mem,size=512M,mem-path=/hugetlbfs,
↪share=on \
     -numa node,memdev=mem \
     -chardev socket,id=chr0,path=/path/to/socket \
     -netdev type=vhost-user,id=net0,chardev=chr0 \
     -device virtio-net-pci,netdev=net0
```

`-netdev vhost-vdpa,vhostdev=/path/to/dev` Establish a vhost-vdpa netdev.

vDPA device is a device that uses a datapath which complies with the virtio specifications with a vendor specific control path. vDPA devices can be both physically located on the hardware or emulated by software.

`-netdev hubport,id=id,hubid=hubid[,netdev=nd]` Create a hub port on the emulated hub with ID hubid.

The hubport netdev lets you connect a NIC to a QEMU emulated hub instead of a single netdev. Alternatively, you can also connect the hubport to another netdev with ID nd by using the `netdev=nd` option.

`-net nic[,netdev=nd][,macaddr=mac][,model=type] [,name=name][,addr=addr][,vectors=v]` Legacy option to configure or create an on-board (or machine default) Network Interface Card(NIC) and connect it either to the emulated hub with ID 0 (i.e. the default hub), or to the netdev nd. If model is omitted, then the default NIC model associated with the machine type is used. Note that the default NIC model may change in future QEMU releases, so it is highly recommended to always specify a model. Optionally, the MAC address can be changed to mac, the device address set to addr (PCI cards only), and a name can be assigned for use in monitor commands. Optionally, for PCI cards, you can specify the number v of MSI-X vectors that the card should have; this option currently only affects virtio cards; set v = 0 to disable MSI-X. If no `-net` option is specified, a single NIC is created. QEMU can emulate several different models of network card. Use `-net nic,model=help` for a list of available devices for your target.

`-net user|tap|bridge|socket|l2tpv3|vde[,...][,name=name]` Configure a host network backend (with the options corresponding to the same `-netdev` option) and connect it to the emulated hub 0 (the default hub). Use name to specify the name of the hub port.

1.2.7 Character device options

The general form of a character device option is:

`-chardev backend,id=id[,mux=on|off][,options]` Backend is one of: `null`, `socket`, `udp`, `msmouse`, `vc`, `ringbuf`, `file`, `pipe`, `console`, `serial`, `pty`, `stdio`, `braille`, `tty`, `parallel`, `parport`, `spicevmc`, `spiceport`. The specific backend will determine the applicable options.

Use `-chardev help` to print all available chardev backend types.

All devices must have an id, which can be any string up to 127 characters long. It is used to uniquely identify this device in other command line directives.

A character device may be used in multiplexing mode by multiple front-ends. Specify `mux=on` to enable this mode. A multiplexer is a “1:N” device, and here the “1” end is your specified chardev backend, and the “N” end is the various parts of QEMU that can talk to a chardev. If you create a chardev with `id=myid` and `mux=on`, QEMU will create a multiplexer with your specified ID, and you can then configure multiple front ends to use that chardev ID for their input/output. Up to four different front ends can be connected to a single multiplexed chardev. (Without multiplexing enabled, a chardev can only be used by a single front end.) For instance you could use this to allow a single stdio chardev to be used by two serial ports and the QEMU monitor:

```
-chardev stdio,mux=on,id=char0 \
-mon chardev=char0,mode=readline \
```

(continues on next page)

(continued from previous page)

```
-serial chardev:char0 \
-serial chardev:char0
```

You can have more than one multiplexer in a system configuration; for instance you could have a TCP port multiplexed between UART 0 and UART 1, and stdio multiplexed between the QEMU monitor and a parallel port:

```
-chardev stdio,mux=on,id=char0 \
-mon chardev=char0,mode=readline \
-parallel chardev:char0 \
-chardev tcp,...,mux=on,id=char1 \
-serial chardev:char1 \
-serial chardev:char1
```

When you're using a multiplexed character device, some escape sequences are interpreted in the input. See *Keys in the character backend multiplexer*.

Note that some other command line options may implicitly create multiplexed character backends; for instance `-serial mon:stdio` creates a multiplexed stdio backend connected to the serial port and the QEMU monitor, and `-nographic` also multiplexes the console and the monitor to stdio.

There is currently no support for multiplexing in the other direction (where a single QEMU front end takes input and output from multiple chardevs).

Every backend supports the `logfile` option, which supplies the path to a file to record all data transmitted via the backend. The `logappend` option controls whether the log file will be truncated or appended to when opened.

The available backends are:

`-chardev null,id=id` A void device. This device will not emit any data, and will drop any data it receives. The null backend does not take any options.

`-chardev socket,id=id[,TCP options or unix options][,server][,nowait][,telnet][,websocket]`
Create a two-way stream socket, which can be either a TCP or a unix socket. A unix socket will be created if `path` is specified. Behaviour is undefined if TCP options are specified for a unix socket.

`server` specifies that the socket shall be a listening socket.

`nowait` specifies that QEMU should not block waiting for a client to connect to a listening socket.

`telnet` specifies that traffic on the socket should interpret telnet escape sequences.

`websocket` specifies that the socket uses WebSocket protocol for communication.

`reconnect` sets the timeout for reconnecting on non-server sockets when the remote end goes away. qemu will delay this many seconds and then attempt to reconnect. Zero disables reconnecting, and is the default.

`tls-creds` requests enablement of the TLS protocol for encryption, and specifies the id of the TLS credentials to use for the handshake. The credentials must be previously created with the `-object tls-creds` argument.

`tls-auth` provides the ID of the QAuthZ authorization object against which the client's x509 distinguished name will be validated. This object is only resolved at time of use, so can be deleted and recreated on the fly while the chardev server is active. If missing, it will default to denying access.

TCP and unix socket options are given below:

`TCP options: port=port[,host=host][,to=to][,ipv4][,ipv6][,nodelay]` `host` for a listening socket specifies the local address to be bound. For a connecting socket species the remote host to connect to. `host` is optional for listening sockets. If not specified it defaults to `0.0.0.0`.

`port` for a listening socket specifies the local port to be bound. For a connecting socket specifies the port on the remote host to connect to. `port` can be given as either a port number or a service name. `port` is required.

`to` is only relevant to listening sockets. If it is specified, and `port` cannot be bound, QEMU will attempt to bind to subsequent ports up to and including `to` until it succeeds. `to` must be specified as a port number.

`ipv4` and `ipv6` specify that either IPv4 or IPv6 must be used. If neither is specified the socket may use either protocol.

`nodelay` disables the Nagle algorithm.

`unix options: path=path[,abstract=on|off][,tight=on|off]` `path` specifies the local path of the unix socket. `path` is required. `abstract` specifies the use of the abstract socket namespace, rather than the filesystem. Optional, defaults to false. `tight` sets the socket length of abstract sockets to their minimum, rather than the full sun_path length. Optional, defaults to true.

`-chardev udp,id=id[,host=host],port=port[,localaddr=localaddr][,localport=localport][,ipv4]`
Sends all traffic from the guest to a remote host over UDP.

`host` specifies the remote host to connect to. If not specified it defaults to `localhost`.

`port` specifies the port on the remote host to connect to. `port` is required.

`localaddr` specifies the local address to bind to. If not specified it defaults to `0.0.0.0`.

`localport` specifies the local port to bind to. If not specified any available local port will be used.

`ipv4` and `ipv6` specify that either IPv4 or IPv6 must be used. If neither is specified the device may use either protocol.

`-chardev msmouse,id=id` Forward QEMU's emulated msmouse events to the guest. `msmouse` does not take any options.

`-chardev vc,id=id[[,width=width][,height=height]][[,cols=cols][,rows=rows]]`
Connect to a QEMU text console. `vc` may optionally be given a specific size.

`width` and `height` specify the width and height respectively of the console, in pixels.

`cols` and `rows` specify that the console be sized to fit a text console with the given dimensions.

`-chardev ringbuf,id=id[,size=size]` Create a ring buffer with fixed size `size`. size must be a power of two and defaults to `64K`.

`-chardev file,id=id,path=path` Log all traffic received from the guest to a file.

`path` specifies the path of the file to be opened. This file will be created if it does not already exist, and overwritten if it does. `path` is required.

`-chardev pipe,id=id,path=path` Create a two-way connection to the guest. The behaviour differs slightly between Windows hosts and other hosts:

On Windows, a single duplex pipe will be created at `\\.pipe\path`.

On other hosts, 2 pipes will be created called `path.in` and `path.out`. Data written to `path.in` will be received by the guest. Data written by the guest can be read from `path.out`. QEMU will not create these fifos, and requires them to be present.

`path` forms part of the pipe path as described above. `path` is required.

`-chardev console,id=id` Send traffic from the guest to QEMU's standard output. `console` does not take any options.

`console` is only available on Windows hosts.

-chardev serial,id=id,path=path Send traffic from the guest to a serial device on the host.

On Unix hosts serial will actually accept any tty device, not only serial lines.

`path` specifies the name of the serial device to open.

-chardev pty,id=id Create a new pseudo-terminal on the host and connect to it. `pty` does not take any options.

`pty` is not available on Windows hosts.

-chardev stdio,id=id[,signal=on|off] Connect to standard input and standard output of the QEMU process.

`signal` controls if signals are enabled on the terminal, that includes exiting QEMU with the key sequence Control-c. This option is enabled by default, use `signal=off` to disable it.

-chardev braille,id=id Connect to a local BrlAPI server. `braille` does not take any options.

-chardev tty,id=id,path=path `tty` is only available on Linux, Sun, FreeBSD, NetBSD, OpenBSD and DragonFlyBSD hosts. It is an alias for `serial`.

`path` specifies the path to the tty. `path` is required.

-chardev parallel,id=id,path=path

-chardev parport,id=id,path=path `parallel` is only available on Linux, FreeBSD and DragonFlyBSD hosts.

Connect to a local parallel port.

`path` specifies the path to the parallel port device. `path` is required.

-chardev spicevmc,id=id,debug=debug,name=name `spicevmc` is only available when spice support is built in.

`debug` debug level for spicevmc

`name` name of spice channel to connect to

Connect to a spice virtual machine channel, such as vdiport.

-chardev spiceport,id=id,debug=debug,name=name `spiceport` is only available when spice support is built in.

`debug` debug level for spicevmc

`name` name of spice port to connect to

Connect to a spice port, allowing a Spice client to handle the traffic identified by a name (preferably a fqdn).

1.2.8 TPM device options

The general form of a TPM device option is:

-tpmdev backend,id=id[,options] The specific backend type will determine the applicable options. The `-tpmdev` option creates the TPM backend and requires a `-device` option that specifies the TPM frontend interface model.

Use `-tpmdev help` to print all available TPM backend types.

The available backends are:

-tpmdev passthrough,id=id,path=path,cancel-path=cancel-path (Linux-host only) Enable access to the host's TPM using the passthrough driver.

`path` specifies the path to the host's TPM device, i.e., on a Linux host this would be `/dev/tpm0`. `path` is optional and by default `/dev/tpm0` is used.

`cancel-path` specifies the path to the host TPM device's sysfs entry allowing for cancellation of an ongoing TPM command. `cancel-path` is optional and by default QEMU will search for the sysfs entry to use.

Some notes about using the host's TPM with the passthrough driver:

The TPM device accessed by the passthrough driver must not be used by any other application on the host.

Since the host's firmware (BIOS/UEFI) has already initialized the TPM, the VM's firmware (BIOS/UEFI) will not be able to initialize the TPM again and may therefore not show a TPM-specific menu that would otherwise allow the user to configure the TPM, e.g., allow the user to enable/disable or activate/deactivate the TPM. Further, if TPM ownership is released from within a VM then the host's TPM will get disabled and deactivated. To enable and activate the TPM again afterwards, the host has to be rebooted and the user is required to enter the firmware's menu to enable and activate the TPM. If the TPM is left disabled and/or deactivated most TPM commands will fail.

To create a passthrough TPM use the following two options:

```
-tpmdev passthrough,id=tpm0 -device tpm-tis,tpmdev=tpm0
```

Note that the `-tpmdev` id is `tpm0` and is referenced by `tpmdev=tpm0` in the device option.

`-tpmdev emulator,id=id,chardev=dev` (Linux-host only) Enable access to a TPM emulator using Unix domain socket based chardev backend.

`chardev` specifies the unique ID of a character device backend that provides connection to the software TPM server.

To create a TPM emulator backend device with chardev socket backend:

```
-chardev socket,id=chrtpm,path=/tmp/swtpm-sock -tpmdev emulator,id=tpm0,
↪chardev=chrtpm -device tpm-tis,tpmdev=tpm0
```

1.2.9 Linux/Multiboot boot specific

When using these options, you can use a given Linux or Multiboot kernel without installing it in the disk image. It can be useful for easier testing of various kernels.

`-kernel bzImage` Use bzImage as kernel image. The kernel can be either a Linux kernel or in multiboot format.

`-append cmdline` Use cmdline as kernel command line

`-initrd file` Use file as initial ram disk.

`-initrd "file1 arg=foo,file2"` This syntax is only available with multiboot.

Use file1 and file2 as modules and pass arg=foo as parameter to the first module.

`-dtb file` Use file as a device tree binary (dtb) image and pass it to the kernel on boot.

1.2.10 Debug/Expert options

`-fw_cfg [name=]name,file=file` Add named fw_cfg entry with contents from file file.

`-fw_cfg [name=]name,string=str` Add named fw_cfg entry with contents from string str.

The terminating NUL character of the contents of str will not be included as part of the fw_cfg item data. To insert contents with embedded NUL characters, you have to use the file parameter.

The fw_cfg entries are passed by QEMU through to the guest.

Example:

```
-fw_cfg name=opt/com.mycompany/blob,file=./my_blob.bin
```

creates an fw_cfg entry named opt/com.mycompany/blob with contents from ./my_blob.bin.

-serial dev Redirect the virtual serial port to host character device dev. The default device is `vc` in graphical mode and `stdio` in non graphical mode.

This option can be used several times to simulate up to 4 serial ports.

Use `-serial none` to disable all serial ports.

Available character devices are:

vc[:WxH] Virtual console. Optionally, a width and height can be given in pixel with

```
vc:800x600
```

It is also possible to specify width or height in characters:

```
vc:80Cx24C
```

pty [Linux only] Pseudo TTY (a new PTY is automatically allocated)

none No device is allocated.

null void device

chardev:id Use a named character device defined with the `-chardev` option.

/dev/XXX [Linux only] Use host tty, e.g. `/dev/ttyS0`. The host serial port parameters are set according to the emulated ones.

/dev/parportN [Linux only, parallel port only] Use host parallel port N. Currently SPP and EPP parallel port features can be used.

file:filename Write output to filename. No character can be read.

stdio [Unix only] standard input/output

pipe:filename name pipe filename

COMn [Windows only] Use host serial port n

udp:[remote_host]:remote_port[@[src_ip]:src_port] This implements UDP Net Console. When remote_host or src_ip are not specified they default to `0.0.0.0`. When not using a specified src_port a random port is automatically chosen.

If you just want a simple readonly console you can use `netcat` or `nc`, by starting QEMU with: `-serial udp::4555` and nc as: `nc -u -l -p 4555`. Any time QEMU writes something to that port it will appear in the netconsole session.

If you plan to send characters back via netconsole or you want to stop and start QEMU a lot of times, you should have QEMU use the same source port each time by using something like `-serial udp::4555@:4556` to QEMU. Another approach is to use a patched version of netcat which can listen to a TCP port and send and receive characters via udp. If you have a patched version of netcat which activates telnet remote echo and single char transfer, then you can use the following options to set up a netcat redirector to allow telnet on port 5555 to access the QEMU port.

QEMU Options: -serial udp::4555@:4556

netcat options: -u -P 4555 -L 0.0.0.0:4556 -t -p 5555 -I -T

telnet options: localhost 5555

tcp:[host]:port[,server][,nowait][,nodelay][,reconnect=seconds] The TCP Net Console has two modes of operation. It can send the serial I/O to a location or wait for a connection from a location. By default the TCP Net Console is sent to host at the port. If you use the server option QEMU will wait for a client socket application to connect to the port before continuing, unless the `nowait` option was specified. The `nodelay` option disables the Nagle buffering algorithm. The `reconnect` option only applies if noserver is set, if the connection goes down it will attempt to reconnect at the given interval. If host is omitted, 0.0.0.0 is assumed. Only one TCP connection at a time is accepted. You can use `telnet` to connect to the corresponding character device.

Example to send tcp console to 192.168.0.2 port 4444 -serial tcp:192.168.0.2:4444

Example to listen and wait on port 4444 for connection -serial tcp::4444,server

Example to not wait and listen on ip 192.168.0.100 port 4444 -serial tcp:192.168.0.100:4444,server,nowait

telnet:host:port[,server][,nowait][,nodelay] The telnet protocol is used instead of raw tcp sockets. The options work the same as if you had specified `-serial tcp`. The difference is that the port acts like a telnet server or client using telnet option negotiation. This will also allow you to send the MAGIC_SYSRQ sequence if you use a telnet that supports sending the break sequence. Typically in unix telnet you do it with Control-] and then type "send break" followed by pressing the enter key.

websocket:host:port,server[,nowait][,nodelay] The WebSocket protocol is used instead of raw tcp socket. The port acts as a WebSocket server. Client mode is not supported.

unix:path[,server][,nowait][,reconnect=seconds] A unix domain socket is used instead of a tcp socket. The option works the same as if you had specified `-serial tcp` except the unix domain socket path is used for connections.

mon:dev_string This is a special option to allow the monitor to be multiplexed onto another serial port. The monitor is accessed with key sequence of Control-a and then pressing c. dev_string should be any one of the serial devices specified above. An example to multiplex the monitor onto a telnet server listening on port 4444 would be:

```
-serial mon:telnet::4444,server,nowait
```

When the monitor is multiplexed to stdio in this way, Ctrl+C will not terminate QEMU any more but will be passed to the guest instead.

braille Braille device. This will use BrlAPI to display the braille output on a real or fake device.

msmouse Three button serial mouse. Configure the guest to use Microsoft protocol.

-parallel dev Redirect the virtual parallel port to host device dev (same devices as the serial port). On Linux hosts, `/dev/parportN` can be used to use hardware devices connected on the corresponding host parallel port.

This option can be used several times to simulate up to 3 parallel ports.

Use `-parallel none` to disable all parallel ports.

-monitor dev Redirect the monitor to host device dev (same devices as the serial port). The default device is `vc` in graphical mode and `stdio` in non graphical mode. Use `-monitor none` to disable the default monitor.

-qmp dev Like -monitor but opens in 'control' mode.

-qmp-pretty dev Like -qmp but uses pretty JSON formatting.

-mon [chardev=]name[,mode=readline|control][,pretty[=on|off]] Setup monitor on chardev name. `pretty` turns on JSON pretty printing easing human reading and debugging.

-debugcon dev Redirect the debug console to host device dev (same devices as the serial port). The debug console is an I/O port which is typically port 0xe9; writing to that I/O port sends output to this device. The default device is `vc` in graphical mode and `stdio` in non graphical mode.

-pidfile file Store the QEMU process PID in file. It is useful if you launch QEMU from a script.

-singlestep Run the emulation in single step mode.

--preconfig Pause QEMU for interactive configuration before the machine is created, which allows querying and configuring properties that will affect machine initialization. Use QMP command 'x-exit-preconfig' to exit the preconfig state and move to the next state (i.e. run guest if -S isn't used or pause the second time if -S is used). This option is experimental.

-S Do not start CPU at startup (you must type 'c' in the monitor).

-realtime mlock=on|off Run qemu with realtime features. mlocking qemu and guest memory can be enabled via `mlock=on` (enabled by default).

-overcommit mem-lock=on|off

-overcommit cpu-pm=on|off Run qemu with hints about host resource overcommit. The default is to assume that host overcommits all resources.

Locking qemu and guest memory can be enabled via `mem-lock=on` (disabled by default). This works when host memory is not overcommitted and reduces the worst-case latency for guest. This is equivalent to `realtime`.

Guest ability to manage power state of host cpus (increasing latency for other processes on the same host cpu, but decreasing latency for guest) can be enabled via `cpu-pm=on` (disabled by default). This works best when host CPU is not overcommitted. When used, host estimates of CPU cycle and power utilization will be incorrect, not taking into account guest idle time.

-gdb dev Accept a gdb connection on device dev (see *GDB usage*). Note that this option does not pause QEMU execution – if you want QEMU to not start the guest until you connect with gdb and issue a `continue` command, you will need to also pass the `-S` option to QEMU.

The most usual configuration is to listen on a local TCP socket:

```
-gdb tcp::3117
```

but you can specify other backends; UDP, pseudo TTY, or even stdio are all reasonable use cases. For example, a stdio connection allows you to start QEMU from within gdb and establish the connection via a pipe:

```
(gdb) target remote | exec qemu-system-x86_64 -gdb stdio ...
```

-s Shorthand for -gdb tcp::1234, i.e. open a gdbserver on TCP port 1234 (see *GDB usage*).

-d item1[,...] Enable logging of specified items. Use '-d help' for a list of log items.

-D logfile Output log in logfile instead of to stderr

-dfilter range1[,...] Filter debug output to that relevant to a range of target addresses. The filter spec can be either start+size, start-size or start..end where start end and size are the addresses and sizes required. For example:

```
-dfilter 0x8000..0x8fff,0xffffffc000080000+0x200,0xffffffc000060000-0x1000
```

Will dump output for any code in the 0x1000 sized block starting at 0x8000 and the 0x200 sized block starting at 0xffffffc000080000 and another 0x1000 sized block starting at 0xffffffc00005f000.

-seed number Force the guest to use a deterministic pseudo-random number generator, seeded with number. This does not affect crypto routines within the host.

`-L path` Set the directory for the BIOS, VGA BIOS and keymaps.

To list all the data directories, use `-L help`.

`-bios file` Set the filename for the BIOS.

`-enable-kvm` Enable KVM full virtualization support. This option is only available if KVM support is enabled when compiling.

`-xen-domid id` Specify xen guest domain id (XEN only).

`-xen-attach` Attach to existing xen domain. libxl will use this when starting QEMU (XEN only). Restrict set of available xen operations to specified domain id (XEN only).

`-no-reboot` Exit instead of rebooting.

`-no-shutdown` Don't exit QEMU on guest shutdown, but instead only stop the emulation. This allows for instance switching to monitor to commit changes to the disk image.

`-loadvm file` Start right away with a saved state (`loadvm` in monitor)

`-daemonize` Daemonize the QEMU process after initialization. QEMU will not detach from standard IO until it is ready to receive connections on any of its devices. This option is a useful way for external programs to launch QEMU without having to cope with initialization race conditions.

`-option-rom file` Load the contents of file as an option ROM. This option is useful to load things like EtherBoot.

`-rtc [base=utc|localtime|datetime][,clock=host|rt|vm][,driftfix=none|slew]`
Specify `base` as `utc` or `localtime` to let the RTC start at the current UTC or local time, respectively. `localtime` is required for correct date in MS-DOS or Windows. To start at a specific point in time, provide datetime in the format `2006-06-17T16:01:21` or `2006-06-17`. The default base is UTC.

By default the RTC is driven by the host system time. This allows using of the RTC as accurate reference clock inside the guest, specifically if the host time is smoothly following an accurate external reference clock, e.g. via NTP. If you want to isolate the guest time from the host, you can set `clock` to `rt` instead, which provides a host monotonic clock if host support it. To even prevent the RTC from progressing during suspension, you can set `clock` to `vm` (virtual clock). '`clock=vm`' is recommended especially in icount mode in order to preserve determinism; however, note that in icount mode the speed of the virtual clock is variable and can in general differ from the host clock.

Enable `driftfix` (i386 targets only) if you experience time drift problems, specifically with Windows' ACPI HAL. This option will try to figure out how many timer interrupts were not processed by the Windows guest and will re-inject them.

`-icount [shift=N|auto][,rr=record|replay,rrfile=filename,rrsnapshot=snapshot]`
Enable virtual instruction counter. The virtual cpu will execute one instruction every 2^N ns of virtual time. If `auto` is specified then the virtual cpu speed will be automatically adjusted to keep virtual time within a few seconds of real time.

When the virtual cpu is sleeping, the virtual time will advance at default speed unless `sleep=on|off` is specified. With `sleep=on|off`, the virtual time will jump to the next timer deadline instantly whenever the virtual cpu goes to sleep mode and will not advance if no timer is enabled. This behavior give deterministic execution times from the guest point of view.

Note that while this option can give deterministic behavior, it does not provide cycle accurate emulation. Modern CPUs contain superscalar out of order cores with complex cache hierarchies. The number of instructions executed often has little or no correlation with actual performance.

`align=on` will activate the delay algorithm which will try to synchronise the host clock and the virtual clock. The goal is to have a guest running at the real frequency imposed by the shift option. Whenever the guest clock is behind the host clock and if `align=on` is specified then we print a message to the user to inform about the delay. Currently this option does not work when `shift` is `auto`. Note: The sync algorithm will work for those shift values for which the guest clock runs ahead of the host clock. Typically this happens when the shift value is high (how high depends on the host machine).

When `rr` option is specified deterministic record/replay is enabled. Replay log is written into filename file in record mode and read from this file in replay mode.

Option rrsnapshot is used to create new vm snapshot named snapshot at the start of execution recording. In replay mode this option is used to load the initial VM state.

`-watchdog model` Create a virtual hardware watchdog device. Once enabled (by a guest action), the watchdog must be periodically polled by an agent inside the guest or else the guest will be restarted. Choose a model for which your guest has drivers.

The model is the model of hardware watchdog to emulate. Use `-watchdog help` to list available hardware models. Only one watchdog can be enabled for a guest.

The following models may be available:

`ib700` iBASE 700 is a very simple ISA watchdog with a single timer.

`i6300esb` Intel 6300ESB I/O controller hub is a much more featureful PCI-based dual-timer watchdog.

`diag288` A virtual watchdog for s390x backed by the diagnose 288 hypercall (currently KVM only).

`-watchdog-action action` The action controls what QEMU will do when the watchdog timer expires. The default is `reset` (forcefully reset the guest). Other possible actions are: `shutdown` (attempt to gracefully shutdown the guest), `poweroff` (forcefully poweroff the guest), `inject-nmi` (inject a NMI into the guest), `pause` (pause the guest), `debug` (print a debug message and continue), or `none` (do nothing).

Note that the `shutdown` action requires that the guest responds to ACPI signals, which it may not be able to do in the sort of situations where the watchdog would have expired, and thus `-watchdog-action shutdown` is not recommended for production use.

Examples:

`-watchdog i6300esb -watchdog-action pause`; `-watchdog ib700`

`-echr numeric_ascii_value` Change the escape character used for switching to the monitor when using monitor and serial sharing. The default is `0x01` when using the `-nographic` option. `0x01` is equal to pressing `Control-a`. You can select a different character from the ascii control keys where 1 through 26 map to Control-a through Control-z. For instance you could use the either of the following to change the escape character to Control-t.

`-echr 0x14`; `-echr 20`

`-show-cursor` Show cursor.

`-tb-size n` Set TCG translation block cache size. Deprecated, use '`-accel tcg,tb-size=n`' instead.

`-incoming tcp:[host]:port[,to=maxport][,ipv4][,ipv6]`

`-incoming rdma:host:port[,ipv4][,ipv6]` Prepare for incoming migration, listen on a given tcp port.

`-incoming unix:socketpath` Prepare for incoming migration, listen on a given unix socket.

`-incoming fd:fd` Accept incoming migration from a given filedescriptor.

`-incoming exec:cmdline` Accept incoming migration as an output from specified external command.

`-incoming defer` Wait for the URI to be specified via migrate_incoming. The monitor can be used to change settings (such as migration parameters) prior to issuing the migrate_incoming to allow the migration to begin.

`-only-migratable` Only allow migratable devices. Devices will not be allowed to enter an unmigratable state.

-nodefaults Don't create default devices. Normally, QEMU sets the default devices like serial port, parallel port, virtual console, monitor device, VGA adapter, floppy and CD-ROM drive and others. The `-nodefaults` option will disable all those default devices.

-chroot dir Immediately before starting guest execution, chroot to the specified directory. Especially useful in combination with -runas.

-runas user Immediately before starting guest execution, drop root privileges, switching to the specified user.

-prom-env variable=value Set OpenBIOS nvram variable to given value (PPC, SPARC only).

```
qemu-system-sparc -prom-env 'auto-boot?=false' \
 -prom-env 'boot-device=sd(0,2,0):d' -prom-env 'boot-args=linux single'
```

```
qemu-system-ppc -prom-env 'auto-boot?=false' \
 -prom-env 'boot-device=hd:2,\yaboot' \
 -prom-env 'boot-args=conf=hd:2,\yaboot.conf'
```

-semihosting Enable semihosting mode (ARM, M68K, Xtensa, MIPS, Nios II only).

Note that this allows guest direct access to the host filesystem, so should only be used with a trusted guest OS.

See the -semihosting-config option documentation for further information about the facilities this enables.

-semihosting-config [enable=on|off][,target=native|gdb|auto][,chardev=id][,arg=str[,...]] Enable and configure semihosting (ARM, M68K, Xtensa, MIPS, Nios II only).

Note that this allows guest direct access to the host filesystem, so should only be used with a trusted guest OS.

On Arm this implements the standard semihosting API, version 2.0.

On M68K this implements the "ColdFire GDB" interface used by libgloss.

Xtensa semihosting provides basic file IO calls, such as open/read/write/seek/select. Tensilica baremetal libc for ISS and linux platform "sim" use this interface.

target=native|gdb|auto Defines where the semihosting calls will be addressed, to QEMU (`native`) or to GDB (`gdb`). The default is `auto`, which means `gdb` during debug sessions and `native` otherwise.

chardev=str1 Send the output to a chardev backend output for native or auto output when not in gdb

arg=str1,arg=str2,... Allows the user to pass input arguments, and can be used multiple times to build up a list. The old-style `-kernel`/`-append` method of passing a command line is still supported for backward compatibility. If both the `--semihosting-config arg` and the `-kernel`/`-append` are specified, the former is passed to semihosting as it always takes precedence.

-old-param Old param mode (ARM only).

-sandbox arg[,obsolete=string][,elevateprivileges=string][,spawn=string][,resourcecontrol=s Enable Seccomp mode 2 system call filter. 'on' will enable syscall filtering and 'off' will disable it. The default is 'off'.

obsolete=string Enable Obsolete system calls

elevateprivileges=string Disable set*uid|gid system calls

spawn=string Disable *fork and execve

resourcecontrol=string Disable process affinity and schedular priority

-readconfig file Read device configuration from file. This approach is useful when you want to spawn QEMU process with many command line options but you don't want to exceed the command line character limit.

-writeconfig file Write device configuration to file. The file can be either filename to save command line and device configuration into file or dash -) character to print the output to stdout. This can be later used as input file for `-readconfig` option.

-no-user-config The `-no-user-config` option makes QEMU not load any of the user-provided config files on sysconfdir.

-trace [[enable=]pattern][,events=file][,file=file] Specify tracing options.

[enable=]PATTERN
Immediately enable events matching *PATTERN* (either event name or a globbing pattern). This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend. To specify multiple events or patterns, specify the `-trace` option multiple times.

Use `-trace help` to print a list of names of trace points.

events=FILE
Immediately enable events listed in *FILE*. The file must contain one event name (as listed in the `trace-events-all` file) per line; globbing patterns are accepted too. This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend.

file=FILE
Log output traces to *FILE*. This option is only available if QEMU has been compiled with the `simple` tracing backend.

-plugin file=file[,arg=string] Load a plugin.

file=file Load the given plugin from a shared library file.

arg=string Argument string passed to the plugin. (Can be given multiple times.)

-enable-fips Enable FIPS 140-2 compliance mode.

-msg [timestamp[=on|off]][,guest-name[=on|off]] Control error message format.

timestamp=on|off Prefix messages with a timestamp. Default is off.

guest-name=on|off Prefix messages with guest name but only if -name guest option is set otherwise the option is ignored. Default is off.

-dump-vmstate file Dump json-encoded vmstate information for current machine type to file in file

-enable-sync-profile Enable synchronization profiling.

1.2.11 Generic object creation

-object typename[,prop1=value1,...] Create a new object of type typename setting properties in the order they are specified. Note that the 'id' property must be set. These objects are placed in the '/objects' path.

-object memory-backend-file,id=id,size=size,mem-path=dir,share=on|off,discard-data=on|
Creates a memory file backend object, which can be used to back the guest RAM with huge pages.

The `id` parameter is a unique ID that will be used to reference this memory region when configuring the `-numa` argument.

The `size` option provides the size of the memory region, and accepts common suffixes, eg `500M`.

The `mem-path` provides the path to either a shared memory or huge page filesystem mount.

The `share` boolean option determines whether the memory region is marked as private to QEMU, or shared. The latter allows a co-operating external process to access the QEMU memory region.

The `share` is also required for pvrdma devices due to limitations in the RDMA API provided by Linux.

Setting share=on might affect the ability to configure NUMA bindings for the memory backend under some circumstances, see Documentation/vm/numa_memory_policy.txt on the Linux kernel source tree for additional details.

Setting the `discard-data` boolean option to on indicates that file contents can be destroyed when QEMU exits, to avoid unnecessarily flushing data to the backing file. Note that `discard-data` is only an optimization, and QEMU might not discard file contents if it aborts unexpectedly or is terminated using SIGKILL.

The `merge` boolean option enables memory merge, also known as MADV_MERGEABLE, so that Kernel Samepage Merging will consider the pages for memory deduplication.

Setting the `dump` boolean option to off excludes the memory from core dumps. This feature is also known as MADV_DONTDUMP.

The `prealloc` boolean option enables memory preallocation.

The `host-nodes` option binds the memory range to a list of NUMA host nodes.

The `policy` option sets the NUMA policy to one of the following values:

`default` default host policy

`preferred` prefer the given host node list for allocation

`bind` restrict memory allocation to the given host node list

`interleave` interleave memory allocations across the given host node list

The `align` option specifies the base address alignment when QEMU mmap(2) `mem-path`, and accepts common suffixes, eg `2M`. Some backend store specified by `mem-path` requires an alignment different than the default one used by QEMU, eg the device DAX /dev/dax0.0 requires 2M alignment rather than 4K. In such cases, users can specify the required alignment via this option.

The `pmem` option specifies whether the backing file specified by `mem-path` is in host persistent memory that can be accessed using the SNIA NVM programming model (e.g. Intel NVDIMM). If `pmem` is set to 'on', QEMU will take necessary operations to guarantee the persistence of its own writes to `mem-path` (e.g. in vNVDIMM label emulation and live migration). Also, we will map the backend-file with MAP_SYNC flag, which ensures the file metadata is in sync for `mem-path` in case of host crash or a power failure. MAP_SYNC requires support from both the host kernel (since Linux kernel 4.15) and the filesystem of `mem-path` mounted with DAX option.

`-object memory-backend-ram,id=id,merge=on|off,dump=on|off,share=on|off,prealloc=on|off`

Creates a memory backend object, which can be used to back the guest RAM. Memory backend objects offer more control than the `-m` option that is traditionally used to define guest RAM. Please refer to `memory-backend-file` for a description of the options.

`-object memory-backend-memfd,id=id,merge=on|off,dump=on|off,share=on|off,prealloc=on|o`

Creates an anonymous memory file backend object, which allows QEMU to share the memory with an external process (e.g. when using vhost-user). The memory is allocated with memfd and optional sealing. (Linux only)

The `seal` option creates a sealed-file, that will block further resizing the memory ('on' by default).

The `hugetlb` option specify the file to be created resides in the hugetlbfs filesystem (since Linux 4.14). Used in conjunction with the `hugetlb` option, the `hugetlbsize` option specify the hugetlb page size on systems that support multiple hugetlb page sizes (it must be a power of 2 value supported by the system).

In some versions of Linux, the `hugetlb` option is incompatible with the `seal` option (requires at least Linux 4.16).

Please refer to `memory-backend-file` for a description of the other options.

The `share` boolean option is on by default with memfd.

`-object rng-builtin,id=id` Creates a random number generator backend which obtains entropy from QEMU builtin functions. The `id` parameter is a unique ID that will be used to reference this entropy backend from the `virtio-rng` device. By default, the `virtio-rng` device uses this RNG backend.

`-object rng-random,id=id,filename=/dev/random` Creates a random number generator backend which obtains entropy from a device on the host. The `id` parameter is a unique ID that will be used to reference this entropy backend from the `virtio-rng` device. The `filename` parameter specifies which file to obtain entropy from and if omitted defaults to `/dev/urandom`.

`-object rng-egd,id=id,chardev=chardevid` Creates a random number generator backend which obtains entropy from an external daemon running on the host. The `id` parameter is a unique ID that will be used to reference this entropy backend from the `virtio-rng` device. The `chardev` parameter is the unique ID of a character device backend that provides the connection to the RNG daemon.

`-object tls-creds-anon,id=id,endpoint=endpoint,dir=/path/to/cred/dir,verify-peer=on|of`
Creates a TLS anonymous credentials object, which can be used to provide TLS support on network backends. The `id` parameter is a unique ID which network backends will use to access the credentials. The `endpoint` is either `server` or `client` depending on whether the QEMU network backend that uses the credentials will be acting as a client or as a server. If `verify-peer` is enabled (the default) then once the handshake is completed, the peer credentials will be verified, though this is a no-op for anonymous credentials.

The dir parameter tells QEMU where to find the credential files. For server endpoints, this directory may contain a file dh-params.pem providing diffie-hellman parameters to use for the TLS server. If the file is missing, QEMU will generate a set of DH parameters at startup. This is a computationally expensive operation that consumes random pool entropy, so it is recommended that a persistent set of parameters be generated upfront and saved.

`-object tls-creds-psk,id=id,endpoint=endpoint,dir=/path/to/keys/dir[,username=username`
Creates a TLS Pre-Shared Keys (PSK) credentials object, which can be used to provide TLS support on network backends. The `id` parameter is a unique ID which network backends will use to access the credentials. The `endpoint` is either `server` or `client` depending on whether the QEMU network backend that uses the credentials will be acting as a client or as a server. For clients only, `username` is the username which will be sent to the server. If omitted it defaults to “qemu”.

The dir parameter tells QEMU where to find the keys file. It is called “dir/keys.psk” and contains “username:key” pairs. This file can most easily be created using the GnuTLS `psktool` program.

For server endpoints, dir may also contain a file dh-params.pem providing diffie-hellman parameters to use for the TLS server. If the file is missing, QEMU will generate a set of DH parameters at startup. This is a computationally expensive operation that consumes random pool entropy, so it is recommended that a persistent set of parameters be generated up front and saved.

`-object tls-creds-x509,id=id,endpoint=endpoint,dir=/path/to/cred/dir,priority=priority`
Creates a TLS anonymous credentials object, which can be used to provide TLS support on network backends. The `id` parameter is a unique ID which network backends will use to access the credentials. The `endpoint` is either `server` or `client` depending on whether the QEMU network backend that uses the credentials will be acting as a client or as a server. If `verify-peer` is enabled (the default) then once the handshake is completed, the peer credentials will be verified. With x509 certificates, this implies that the clients must be provided with valid client certificates too.

The dir parameter tells QEMU where to find the credential files. For server endpoints, this directory may contain a file dh-params.pem providing diffie-hellman parameters to use for the TLS server. If the file is missing, QEMU will generate a set of DH parameters at startup. This is a computationally expensive operation that consumes random pool entropy, so it is recommended that a persistent set of parameters be generated upfront and saved.

For x509 certificate credentials the directory will contain further files providing the x509 certificates. The certificates must be stored in PEM format, in filenames ca-cert.pem, ca-crl.pem (optional), server-cert.pem (only servers), server-key.pem (only servers), client-cert.pem (only clients), and client-key.pem (only clients).

For the server-key.pem and client-key.pem files which contain sensitive private keys, it is possible to use an encrypted version by providing the passwordid parameter. This provides the ID of a previously created `secret` object containing the password for decryption.

The priority parameter allows to override the global default priority used by gnutls. This can be useful if the system administrator needs to use a weaker set of crypto priorities for QEMU without potentially forcing the weakness onto all applications. Or conversely if one wants wants a stronger default for QEMU than for all other applications, they can do this through this parameter. Its format is a gnutls priority string as described at https://gnutls.org/manual/html_node/Priority-Strings.html.

`-object tls-cipher-suites,id=id,priority=priority` Creates a TLS cipher suites object, which can be used to control the TLS cipher/protocol algorithms that applications are permitted to use.

The `id` parameter is a unique ID which frontends will use to access the ordered list of permitted TLS cipher suites from the host.

The `priority` parameter allows to override the global default priority used by gnutls. This can be useful if the system administrator needs to use a weaker set of crypto priorities for QEMU without potentially forcing the weakness onto all applications. Or conversely if one wants wants a stronger default for QEMU than for all other applications, they can do this through this parameter. Its format is a gnutls priority string as described at https://gnutls.org/manual/html_node/Priority-Strings.html.

An example of use of this object is to control UEFI HTTPS Boot. The tls-cipher-suites object exposes the ordered list of permitted TLS cipher suites from the host side to the guest firmware, via fw_cfg. The list is represented as an array of IANA_TLS_CIPHER objects. The firmware uses the IANA_TLS_CIPHER array for configuring guest-side TLS.

In the following example, the priority at which the host-side policy is retrieved is given by the `priority` property. Given that QEMU uses GNUTLS, `priority=@SYSTEM` may be used to refer to /etc/crypto-policies/back-ends/gnutls.config.

```
# qemu-system-x86_64     -object tls-cipher-suites,id=mysuite0,
↪priority=@SYSTEM     -fw_cfg name=etc/edk2/https/ciphers,gen_
↪id=mysuite0
```

`-object filter-buffer,id=id,netdev=netdevid,interval=t[,queue=all|rx|tx][,status=on|of`
Interval t can't be 0, this filter batches the packet delivery: all packets arriving in a given interval on netdev netdevid are delayed until the end of the interval. Interval is in microseconds. `status` is optional that indicate whether the netfilter is on (enabled) or off (disabled), the default status for netfilter will be 'on'.

queue all|rx|tx is an option that can be applied to any netfilter.

`all`: the filter is attached both to the receive and the transmit queue of the netdev (default).

`rx`: the filter is attached to the receive queue of the netdev, where it will receive packets sent to the netdev.

`tx`: the filter is attached to the transmit queue of the netdev, where it will receive packets sent by the netdev.

position head|tail|id=<id> is an option to specify where the filter should be inserted in the filter list. It can be applied to any netfilter.

`head`: the filter is inserted at the head of the filter list, before any existing filters.

`tail`: the filter is inserted at the tail of the filter list, behind any existing filters (default).

`id=<id>`: the filter is inserted before or behind the filter specified by <id>, see the insert option below.

insert behind|before is an option to specify where to insert the new filter relative to the one specified with position=id=<id>. It can be applied to any netfilter.

`before`: insert before the specified filter.

`behind`: insert behind the specified filter (default).

`-object filter-mirror,id=id,netdev=netdevid,outdev=chardevid,queue=all|rx|tx[,vnet_hdr`
filter-mirror on netdev netdevid,mirror net packet to chardevchardevid, if it has the vnet_hdr_support flag, filter-mirror will mirror packet with vnet_hdr_len.

`-object filter-redirector,id=id,netdev=netdevid,indev=chardevid,outdev=chardevid,queue`
filter-redirector on netdev netdevid,redirect filter's net packet to chardev chardevid,and redirect indev's packet to filter.if it has the vnet_hdr_support flag, filter-redirector will redirect packet with vnet_hdr_len. Create a filter-redirector we need to differ outdev id from indev id, id can not be the same. we can just use indev or outdev, but at least one of indev or outdev need to be specified.

`-object filter-rewriter,id=id,netdev=netdevid,queue=all|rx|tx,[vnet_hdr_support][,posi`
Filter-rewriter is a part of COLO project.It will rewrite tcp packet to secondary from primary to keep secondary tcp connection,and rewrite tcp packet to primary from secondary make tcp packet can be handled by client.if it has the vnet_hdr_support flag, we can parse packet with vnet header.

usage: colo secondary: -object filter-redirector,id=f1,netdev=hn0,queue=tx,indev=red0 -object filter-redirector,id=f2,netdev=hn0,queue=rx,outdev=red1 -object filter-rewriter,id=rew0,netdev=hn0,queue=all

`-object filter-dump,id=id,netdev=dev[,file=filename][,maxlen=len][,position=head|tail|`
Dump the network traffic on netdev dev to the file specified by filename. At most len bytes (64k by default) per packet are stored. The file format is libpcap, so it can be analyzed with tools such as tcpdump or Wireshark.

`-object colo-compare,id=id,primary_in=chardevid,secondary_in=chardevid,outdev=chardevi`
Colo-compare gets packet from primary_in chardevid and secondary_in, then compare whether the payload of primary packet and secondary packet are the same. If same, it will output primary packet to out_dev, else it will notify COLO-framework to do checkpoint and send primary packet to out_dev. In order to improve efficiency, we need to put the task of comparison in another iothread. If it has the vnet_hdr_support flag, colo compare will send/recv packet with vnet_hdr_len. The compare_timeout=@var{ms} determines the maximum time of the colo-compare hold the packet. The expired_scan_cycle=@var{ms} is to set the period of scanning expired primary node network packets. The max_queue_size=@var{size} is to set the max compare queue size depend on user environment. If user want to use Xen COLO, need to add the notify_dev to notify Xen colo-frame to do checkpoint.

COLO-compare must be used with the help of filter-mirror, filter-redirector and filter-rewriter.

```
KVM COLO

primary:
-netdev tap,id=hn0,vhost=off,script=/etc/qemu-ifup,downscript=/etc/qemu-ifdown
-device e1000,id=e0,netdev=hn0,mac=52:a4:00:12:78:66
-chardev socket,id=mirror0,host=3.3.3.3,port=9003,server,nowait
-chardev socket,id=compare1,host=3.3.3.3,port=9004,server,nowait
-chardev socket,id=compare0,host=3.3.3.3,port=9001,server,nowait
-chardev socket,id=compare0-0,host=3.3.3.3,port=9001
-chardev socket,id=compare_out,host=3.3.3.3,port=9005,server,nowait
-chardev socket,id=compare_out0,host=3.3.3.3,port=9005
-object iothread,id=iothread1
-object filter-mirror,id=m0,netdev=hn0,queue=tx,outdev=mirror0
-object filter-redirector,netdev=hn0,id=redire0,queue=rx,indev=compare_out
-object filter-redirector,netdev=hn0,id=redire1,queue=rx,outdev=compare0
-object colo-compare,id=comp0,primary_in=compare0-0,secondary_in=compare1,
↪outdev=compare_out0,iothread=iothread1
```

(continues on next page)

(continued from previous page)

```
secondary:
-netdev tap,id=hn0,vhost=off,script=/etc/qemu-ifup,down script=/etc/qemu-
↪ifdown
-device e1000,netdev=hn0,mac=52:a4:00:12:78:66
-chardev socket,id=red0,host=3.3.3.3,port=9003
-chardev socket,id=red1,host=3.3.3.3,port=9004
-object filter-redirector,id=f1,netdev=hn0,queue=tx,indev=red0
-object filter-redirector,id=f2,netdev=hn0,queue=rx,outdev=red1


Xen COLO

primary:
-netdev tap,id=hn0,vhost=off,script=/etc/qemu-ifup,downscript=/etc/qemu-ifdown
-device e1000,id=e0,netdev=hn0,mac=52:a4:00:12:78:66
-chardev socket,id=mirror0,host=3.3.3.3,port=9003,server,nowait
-chardev socket,id=compare1,host=3.3.3.3,port=9004,server,nowait
-chardev socket,id=compare0,host=3.3.3.3,port=9001,server,nowait
-chardev socket,id=compare0-0,host=3.3.3.3,port=9001
-chardev socket,id=compare_out,host=3.3.3.3,port=9005,server,nowait
-chardev socket,id=compare_out0,host=3.3.3.3,port=9005
-chardev socket,id=notify_way,host=3.3.3.3,port=9009,server,nowait
-object filter-mirror,id=m0,netdev=hn0,queue=tx,outdev=mirror0
-object filter-redirector,netdev=hn0,id=redire0,queue=rx,indev=compare_out
-object filter-redirector,netdev=hn0,id=redire1,queue=rx,outdev=compare0
-object iothread,id=iothread1
-object colo-compare,id=comp0,primary_in=compare0-0,secondary_in=compare1,
↪outdev=compare_out0,notify_dev=nofity_way,iothread=iothread1

secondary:
-netdev tap,id=hn0,vhost=off,script=/etc/qemu-ifup,down script=/etc/qemu-
↪ifdown
-device e1000,netdev=hn0,mac=52:a4:00:12:78:66
-chardev socket,id=red0,host=3.3.3.3,port=9003
-chardev socket,id=red1,host=3.3.3.3,port=9004
-object filter-redirector,id=f1,netdev=hn0,queue=tx,indev=red0
-object filter-redirector,id=f2,netdev=hn0,queue=rx,outdev=red1
```

If you want to know the detail of above command line, you can read the colo-compare git log.

-object cryptodev-backend-builtin,id=id[,queues=queues] Creates a cryptodev backend which executes crypto opreation from the QEMU cipher APIS. The id parameter is a unique ID that will be used to reference this cryptodev backend from the `virtio-crypto` device. The queues parameter is optional, which specify the queue number of cryptodev backend, the default of queues is 1.

```
# qemu-system-x86_64   [...]       -object cryptodev-backend-
↪builtin,id=cryptodev0       -device virtio-crypto-pci,id=crypto0,
↪cryptodev=cryptodev0   [...]
```

-object cryptodev-vhost-user,id=id,chardev=chardevid[,queues=queues] Creates a vhost-user cryptodev backend, backed by a chardev chardevid. The id parameter is a unique ID that will be used to reference this cryptodev backend from the `virtio-crypto` device. The chardev should be a unix domain socket backed one. The vhost-user uses a specifically defined protocol to pass vhost ioctl replacement messages to an application on the other end of the socket. The queues parameter is optional, which specify the queue number of cryptodev backend for multiqueue vhost-user, the default of queues is 1.

```
# qemu-system-x86_64   [...]       -chardev socket,id=chardev0,path=/
↪path/to/socket       -object cryptodev-vhost-user,id=cryptodev0,
↪chardev=chardev0       -device virtio-crypto-pci,id=crypto0,
↪cryptodev=cryptodev0   [...]
```

-object secret,id=id,data=string,format=raw|base64[,keyid=secretid,iv=string]

-object secret,id=id,file=filename,format=raw|base64[,keyid=secretid,iv=string]

Defines a secret to store a password, encryption key, or some other sensitive data. The sensitive data can either be passed directly via the data parameter, or indirectly via the file parameter. Using the data parameter is insecure unless the sensitive data is encrypted.

The sensitive data can be provided in raw format (the default), or base64. When encoded as JSON, the raw format only supports valid UTF-8 characters, so base64 is recommended for sending binary data. QEMU will convert from which ever format is provided to the format it needs internally. eg, an RBD password can be provided in raw format, even though it will be base64 encoded when passed onto the RBD sever.

For added protection, it is possible to encrypt the data associated with a secret using the AES-256-CBC cipher. Use of encryption is indicated by providing the keyid and iv parameters. The keyid parameter provides the ID of a previously defined secret that contains the AES-256 decryption key. This key should be 32-bytes long and be base64 encoded. The iv parameter provides the random initialization vector used for encryption of this particular secret and should be a base64 encrypted string of the 16-byte IV.

The simplest (insecure) usage is to provide the secret inline

```
# qemu-system-x86_64 -object secret,id=sec0,data=letmein,format=raw
```

The simplest secure usage is to provide the secret via a file

# printf "letmein" > mypasswd.txt # QEMU_SYSTEM_MACRO -object secret,id=sec0,file=mypasswd.txt,format=raw

For greater security, AES-256-CBC should be used. To illustrate usage, consider the openssl command line tool which can encrypt the data. Note that when encrypting, the plaintext must be padded to the cipher block size (32 bytes) using the standard PKCS#5/6 compatible padding algorithm.

First a master key needs to be created in base64 encoding:

```
# openssl rand -base64 32 > key.b64
# KEY=$(base64 -d key.b64 | hexdump  -v -e '/1 "%02X"')
```

Each secret to be encrypted needs to have a random initialization vector generated. These do not need to be kept secret

```
# openssl rand -base64 16 > iv.b64
# IV=$(base64 -d iv.b64 | hexdump  -v -e '/1 "%02X"')
```

The secret to be defined can now be encrypted, in this case we're telling openssl to base64 encode the result, but it could be left as raw bytes if desired.

```
# SECRET=$(printf "letmein" |
           openssl enc -aes-256-cbc -a -K $KEY -iv $IV)
```

When launching QEMU, create a master secret pointing to `key.b64` and specify that to be used to decrypt the user password. Pass the contents of `iv.b64` to the second secret

```
# qemu-system-x86_64     -object secret,id=secmaster0,format=base64,
↪file=key.b64     -object secret,id=sec0,keyid=secmaster0,
↪format=base64,        data=$SECRET,iv=$(<iv.b64)
```

-object sev-guest,id=id,cbitpos=cbitpos,reduced-phys-bits=val,[sev-device=string,polic
Create a Secure Encrypted Virtualization (SEV) guest object, which can be used to provide the guest memory encryption support on AMD processors.

When memory encryption is enabled, one of the physical address bit (aka the C-bit) is utilized to mark if a memory page is protected. The `cbitpos` is used to provide the C-bit position. The C-bit position is Host family dependent hence user must provide this value. On EPYC, the value should be 47.

When memory encryption is enabled, we loose certain bits in physical address space. The `reduced-phys-bits` is used to provide the number of bits we loose in physical address space. Similar to C-bit, the value is Host family dependent. On EPYC, the value should be 5.

The `sev-device` provides the device file to use for communicating with the SEV firmware running inside AMD Secure Processor. The default device is '/dev/sev'. If hardware supports memory encryption then /dev/sev devices are created by CCP driver.

The `policy` provides the guest policy to be enforced by the SEV firmware and restrict what configuration and operational commands can be performed on this guest by the hypervisor. The policy should be provided by the guest owner and is bound to the guest and cannot be changed throughout the lifetime of the guest. The default is 0.

If guest `policy` allows sharing the key with another SEV guest then `handle` can be use to provide handle of the guest from which to share the key.

The `dh-cert-file` and `session-file` provides the guest owner's Public Diffie-Hillman key defined in SEV spec. The PDH and session parameters are used for establishing a cryptographic session with the guest owner to negotiate keys used for attestation. The file must be encoded in base64.

e.g to launch a SEV guest

```
# qemu_system-x86_64     ......
    -object sev-guest,id=sev0,cbitpos=47,reduced-phys-bits=5     -
↪machine ...,memory-encryption=sev0
    .....
```

-object authz-simple,id=id,identity=string Create an authorization object that will control access to network services.

The `identity` parameter is identifies the user and its format depends on the network service that authorization object is associated with. For authorizing based on TLS x509 certificates, the identity must be the x509 distinguished name. Note that care must be taken to escape any commas in the distinguished name.

An example authorization object to validate a x509 distinguished name would look like:

```
# qemu-system-x86_64     ...
    -object 'authz-simple,id=auth0,identity=CN=laptop.example.com,,
↪O=Example Org,,L=London,,ST=London,,C=GB'     ...
```

Note the use of quotes due to the x509 distinguished name containing whitespace, and escaping of ','.

-object authz-listfile,id=id,filename=path,refresh=yes|no Create an authorization object that will control access to network services.

The `filename` parameter is the fully qualified path to a file containing the access control list rules in JSON format.

An example set of rules that match against SASL usernames might look like:

```
{
  "rules": [
     { "match": "fred", "policy": "allow", "format": "exact" },
     { "match": "bob", "policy": "allow", "format": "exact" },
```

(continues on next page)

(continued from previous page)

```
    { "match": "danb", "policy": "deny", "format": "glob" },
    { "match": "dan*", "policy": "allow", "format": "exact" },
  ],
  "policy": "deny"
}
```

When checking access the object will iterate over all the rules and the first rule to match will have its `policy` value returned as the result. If no rules match, then the default `policy` value is returned.

The rules can either be an exact string match, or they can use the simple UNIX glob pattern matching to allow wildcards to be used.

If `refresh` is set to true the file will be monitored and automatically reloaded whenever its content changes.

As with the `authz-simple` object, the format of the identity strings being matched depends on the network service, but is usually a TLS x509 distinguished name, or a SASL username.

An example authorization object to validate a SASL username would look like:

```
# qemu-system-x86_64     ...
    -object authz-simple,id=auth0,filename=/etc/qemu/vnc-sasl.acl,
↪refresh=yes
    ...
```

`-object authz-pam,id=id,service=string` Create an authorization object that will control access to network services.

The `service` parameter provides the name of a PAM service to use for authorization. It requires that a file `/etc/pam.d/service` exist to provide the configuration for the `account` subsystem.

An example authorization object to validate a TLS x509 distinguished name would look like:

```
# qemu-system-x86_64     ...
    -object authz-pam,id=auth0,service=qemu-vnc
    ...
```

There would then be a corresponding config file for PAM at `/etc/pam.d/qemu-vnc` that contains:

```
account requisite  pam_listfile.so item=user sense=allow \
           file=/etc/qemu/vnc.allow
```

Finally the `/etc/qemu/vnc.allow` file would contain the list of x509 distingished names that are permitted access

```
CN=laptop.example.com,O=Example Home,L=London,ST=London,C=GB
```

`-object iothread,id=id,poll-max-ns=poll-max-ns,poll-grow=poll-grow,poll-shrink=poll-sh`
Creates a dedicated event loop thread that devices can be assigned to. This is known as an IOThread. By default device emulation happens in vCPU threads or the main event loop thread. This can become a scalability bottleneck. IOThreads allow device emulation and I/O to run on other host CPUs.

The `id` parameter is a unique ID that will be used to reference this IOThread from `-device ...,iothread=id`. Multiple devices can be assigned to an IOThread. Note that not all devices support an `iothread` parameter.

The `query-iothreads` QMP command lists IOThreads and reports their thread IDs so that the user can configure host CPU pinning/affinity.

IOThreads use an adaptive polling algorithm to reduce event loop latency. Instead of entering a blocking system call to monitor file descriptors and then pay the cost of being woken up when an event occurs, the polling algorithm spins waiting for events for a short time. The algorithm's default parameters are suitable for many cases but can be adjusted based on knowledge of the workload and/or host device latency.

The `poll-max-ns` parameter is the maximum number of nanoseconds to busy wait for events. Polling can be disabled by setting this value to 0.

The `poll-grow` parameter is the multiplier used to increase the polling time when the algorithm detects it is missing events due to not polling long enough.

The `poll-shrink` parameter is the divisor used to decrease the polling time when the algorithm detects it is spending too long polling without encountering events.

The polling parameters can be modified at run-time using the `qom-set` command (where `iothread1` is the IOThread's `id`):

```
(qemu) qom-set /objects/iothread1 poll-max-ns 100000
```

1.2.12 Device URL Syntax

In addition to using normal file images for the emulated storage devices, QEMU can also use networked resources such as iSCSI devices. These are specified using a special URL syntax.

`iSCSI` iSCSI support allows QEMU to access iSCSI resources directly and use as images for the guest storage. Both disk and cdrom images are supported.

Syntax for specifying iSCSI LUNs is "iscsi://<target-ip>[:<port>]/<target-iqn>/<lun>"

By default qemu will use the iSCSI initiator-name 'iqn.2008-11.org.linux-kvm[:<name>]' but this can also be set from the command line or a configuration file.

Since version Qemu 2.4 it is possible to specify a iSCSI request timeout to detect stalled requests and force a reestablishment of the session. The timeout is specified in seconds. The default is 0 which means no timeout. Libiscsi 1.15.0 or greater is required for this feature.

Example (without authentication):

```
qemu-system-x86_64 -iscsi initiator-name=iqn.2001-04.com.example:my-
↪initiator                    -cdrom iscsi://192.0.2.1/iqn.2001-04.com.
↪example/2                    -drive file=iscsi://192.0.2.1/iqn.2001-04.
↪com.example/1
```

Example (CHAP username/password via URL):

```
qemu-system-x86_64 -drive file=iscsi://user%password@192.0.2.1/iqn.2001-
↪04.com.example/1
```

Example (CHAP username/password via environment variables):

```
LIBISCSI_CHAP_USERNAME="user" LIBISCSI_CHAP_PASSWORD="password" qemu-
↪system-x86_64 -drive file=iscsi://192.0.2.1/iqn.2001-04.com.example/1
```

`NBD` QEMU supports NBD (Network Block Devices) both using TCP protocol as well as Unix Domain Sockets. With TCP, the default port is 10809.

Syntax for specifying a NBD device using TCP, in preferred URI form: "nbd://<server-ip>[:<port>]/[<export>]"

Syntax for specifying a NBD device using Unix Domain Sockets; remember that '?' is a shell glob character and may need quoting: "nbd+unix:///[<export>]?socket=<domain-socket>"

Older syntax that is also recognized: "nbd:<server-ip>:<port>[:exportname=<export>]"

Syntax for specifying a NBD device using Unix Domain Sockets “nbd:unix:<domain-socket>[:exportname=<export>]”

Example for TCP

```
qemu-system-x86_64 --drive file=nbd:192.0.2.1:30000
```

Example for Unix Domain Sockets

```
qemu-system-x86_64 --drive file=nbd:unix:/tmp/nbd-socket
```

SSH QEMU supports SSH (Secure Shell) access to remote disks.

Examples:

```
qemu-system-x86_64 -drive file=ssh://user@host/path/to/disk.img
qemu-system-x86_64 -drive file.driver=ssh,file.user=user,file.host=host,
↪file.port=22,file.path=/path/to/disk.img
```

Currently authentication must be done using ssh-agent. Other authentication methods may be supported in future.

Sheepdog Sheepdog is a distributed storage system for QEMU. QEMU supports using either local sheepdog devices or remote networked devices.

Syntax for specifying a sheepdog device

```
sheepdog[+tcp|+unix]://[host:port]/vdiname[?socket=path][#snapid|#tag]
```

Example

```
qemu-system-x86_64 --drive file=sheepdog://192.0.2.1:30000/
↪MyVirtualMachine
```

See also https://sheepdog.github.io/sheepdog/.

GlusterFS GlusterFS is a user space distributed file system. QEMU supports the use of GlusterFS volumes for hosting VM disk images using TCP, Unix Domain Sockets and RDMA transport protocols.

Syntax for specifying a VM disk image on GlusterFS volume is

```
URI:
gluster[+type]://[host[:port]]/volume/path[?socket=...][,debug=N][,logfile=...]

JSON:
'json:{"driver":"qcow2","file":{"driver":"gluster","volume":"testvol","path":"a.
↪img","debug":N,"logfile":"...",
                                 "server":[{"type":"tcp","host":"...","port":"...
↪"},
                                           {"type":"unix","socket":"..."}]}}'
```

Example

```
URI:
qemu-system-x86_64 --drive file=gluster://192.0.2.1/testvol/a.img,
                               file.debug=9,file.logfile=/var/log/qemu-
↪gluster.log

JSON:
qemu-system-x86_64 'json:{"driver":"qcow2",
                          "file":{"driver":"gluster",
                                   "volume":"testvol","path":"a.img",
                                        "debug":9,"logfile":"/var/log/qemu-
↪gluster.log",
                                        "server":[{"type":"tcp","host":"1.2.3.
↪4","port":24007},
                                                  {"type":"unix","socket":"/
↪var/run/glusterd.socket"}]}}'
qemu-system-x86_64 -drive driver=qcow2,file.driver=gluster,file.
↪volume=testvol,file.path=/path/a.img,
                                           file.debug=9,file.logfile=/var/log/
↪qemu-gluster.log,
                                           file.server.0.type=tcp,file.server.
↪0.host=1.2.3.4,file.server.0.port=24007,
                                           file.server.1.type=unix,file.server.
↪1.socket=/var/run/glusterd.socket
```

See also http://www.gluster.org.

`HTTP/HTTPS/FTP/FTPS` QEMU supports read-only access to files accessed over http(s) and ftp(s).

Syntax using a single filename:

```
<protocol>://[<username>[:<password>]@]<host>/<path>
```

where:

`protocol` ‘http’, ‘https’, ‘ftp’, or ‘ftps’.

`username` Optional username for authentication to the remote server.

`password` Optional password for authentication to the remote server.

`host` Address of the remote server.

`path` Path on the remote server, including any query string.

The following options are also supported:

`url` The full URL when passing options to the driver explicitly.

`readahead` The amount of data to read ahead with each range request to the remote server. This value may optionally have the suffix ‘T’, ‘G’, ‘M’, ‘K’, ‘k’ or ‘b’. If it does not have a suffix, it will be assumed to be in bytes. The value must be a multiple of 512 bytes. It defaults to 256k.

`sslverify` Whether to verify the remote server’s certificate when connecting over SSL. It can have the value ‘on’ or ‘off’. It defaults to ‘on’.

`cookie` Send this cookie (it can also be a list of cookies separated by ‘;’) with each outgoing request. Only supported when using protocols such as HTTP which support cookies, otherwise ignored.

`timeout` Set the timeout in seconds of the CURL connection. This timeout is the time that CURL waits for a response from the remote server to get the size of the image to be downloaded. If not set, the default timeout of 5 seconds is used.

Note that when passing options to qemu explicitly, `driver` is the value of <protocol>.

Example: boot from a remote Fedora 20 live ISO image

```
qemu_system-x86_64 --drive media=cdrom,file=https://archives.
↪fedoraproject.org/pub/archive/fedora/linux/releases/20/Live/x86_64/
↪Fedora-Live-Desktop-x86_64-20-1.iso,readonly
```

```
qemu_system-x86_64 --drive media=cdrom,file.driver=http,file.url=http:/
↪/archives.fedoraproject.org/pub/fedora/linux/releases/20/Live/x86_64/
↪Fedora-Live-Desktop-x86_64-20-1.iso,readonly
```

Example: boot from a remote Fedora 20 cloud image using a local overlay for writes, copy-on-read, and a readahead of 64k

```
qemu-img create -f qcow2 -o backing_file='json:{"file.driver":"http",,␣
↪"file.url":"http://archives.fedoraproject.org/pub/archive/fedora/linux/
releases/20/Images/x86_64/Fedora-x86_64-20-20131211.1-sda.qcow2",, "file.
↪readahead":"64k"}' /tmp/Fedora-x86_64-20-20131211.1-sda.qcow2

qemu_system-x86_64 -drive file=/tmp/Fedora-x86_64-20-20131211.1-sda.qcow2,
↪copy-on-read=on
```

Example: boot from an image stored on a VMware vSphere server with a self-signed certificate using a local overlay for writes, a readahead of 64k and a timeout of 10 seconds.

```
qemu-img create -f qcow2 -o backing_file='json:{"file.driver":"https",
↪, "file.url":"https://user:password@vsphere.example.com/folder/
test/test-flat.vmdk?dcPath=Datacenter&dsName=datastore1",, "file.
↪sslverify":"off",, "file.readahead":"64k",, "file.timeout":10}' /tmp/
↪test.qcow2

qemu_system-x86_64 -drive file=/tmp/test.qcow2
```

1.3 Keys in the graphical frontends

During the graphical emulation, you can use special key combinations to change modes. The default key mappings are shown below, but if you use `-alt-grab` then the modifier is Ctrl-Alt-Shift (instead of Ctrl-Alt) and if you use `-ctrl-grab` then the modifier is the right Ctrl key (instead of Ctrl-Alt):

Ctrl-Alt-f Toggle full screen

Ctrl-Alt-+ Enlarge the screen

Ctrl-Alt– Shrink the screen

Ctrl-Alt-u Restore the screen's un-scaled dimensions

Ctrl-Alt-n Switch to virtual console 'n'. Standard console mappings are:

1 Target system display

2 Monitor

3 Serial port

Ctrl-Alt Toggle mouse and keyboard grab.

In the virtual consoles, you can use Ctrl-Up, Ctrl-Down, Ctrl-PageUp and Ctrl-PageDown to move in the back log.

1.4 Keys in the character backend multiplexer

During emulation, if you are using a character backend multiplexer (which is the default if you are using `-nographic`) then several commands are available via an escape sequence. These key sequences all start with an escape character, which is Ctrl-a by default, but can be changed with `-echr`. The list below assumes you're using the default.

Ctrl-a h Print this help

Ctrl-a x Exit emulator

Ctrl-a s Save disk data back to file (if -snapshot)

Ctrl-a t Toggle console timestamps

Ctrl-a b Send break (magic sysrq in Linux)

Ctrl-a c Rotate between the frontends connected to the multiplexer (usually this switches between the monitor and the console)

Ctrl-a Ctrl-a Send the escape character to the frontend

1.5 QEMU Monitor

The QEMU monitor is used to give complex commands to the QEMU emulator. You can use it to:

- Remove or insert removable media images (such as CD-ROM or floppies).
- Freeze/unfreeze the Virtual Machine (VM) and save or restore its state from a disk file.
- Inspect the VM state without an external debugger.

1.5.1 Commands

The following commands are available:

`help` or `?` [*cmd*] Show the help for all commands or just for command *cmd*.

`commit` Commit changes to the disk images (if -snapshot is used) or backing files. If the backing file is smaller than the snapshot, then the backing file will be resized to be the same size as the snapshot. If the snapshot is smaller than the backing file, the backing file will not be truncated. If you want the backing file to match the size of the smaller snapshot, you can safely truncate it yourself once the commit operation successfully completes.

`q` or `quit` Quit the emulator.

`exit_preconfig` This command makes QEMU exit the preconfig state and proceed with VM initialization using configuration data provided on the command line and via the QMP monitor during the preconfig state. The command is only available during the preconfig state (i.e. when the –preconfig command line option was in use).

`block_resize` Resize a block image while a guest is running. Usually requires guest action to see the updated size. Resize to a lower size is supported, but should be used with extreme caution. Note that this command only resizes image files, it can not resize block devices like LVM volumes.

`block_stream` Copy data from a backing file into a block device.

`block_job_set_speed` Set maximum speed for a background block operation.

`block_job_cancel` Stop an active background block operation (streaming, mirroring).

`block_job_complete` Manually trigger completion of an active background block operation. For mirroring, this will switch the device to the destination path.

`block_job_pause` Pause an active block streaming operation.

`block_job_resume` Resume a paused block streaming operation.

`eject [-f]` *device* Eject a removable medium (use -f to force it).

`drive_del` ***device*** Remove host block device. The result is that guest generated IO is no longer submitted against the host device underlying the disk. Once a drive has been deleted, the QEMU Block layer returns -EIO which results in IO errors in the guest for applications that are reading/writing to the device. These errors are always reported to the guest, regardless of the drive's error actions (drive options rerror, werror).

`change` ***device setting*** Change the configuration of a device.

`change` ***diskdevice filename*** **[*format* [*read-only-mode*]]** Change the medium for a removable disk device to point to *filename*. eg:

```
(qemu) change ide1-cd0 /path/to/some.iso
```

format is optional.

read-only-mode may be used to change the read-only status of the device. It accepts the following values:

retain Retains the current status; this is the default.

read-only Makes the device read-only.

read-write Makes the device writable.

`change vnc` ***display,options*** Change the configuration of the VNC server. The valid syntax for *display* and *options* are described at *Invocation*. eg:

```
(qemu) change vnc localhost:1
```

`change vnc password` [*password*]

Change the password associated with the VNC server. If the new password is not supplied, the monitor will prompt for it to be entered. VNC passwords are only significant up to 8 letters. eg:

```
(qemu) change vnc password
Password: ********
```

`screendump` ***filename*** Save screen into PPM image *filename*.

`logfile` ***filename*** Output logs to *filename*.

`trace-event` changes status of a trace event

`trace-file on|off|flush` Open, close, or flush the trace file. If no argument is given, the status of the trace file is displayed.

`log` ***item1*****[,...]** Activate logging of the specified items.

`savevm` ***tag*** Create a snapshot of the whole virtual machine. If *tag* is provided, it is used as human readable identifier. If there is already a snapshot with the same tag, it is replaced. More info at *VM snapshots*.

Since 4.0, savevm stopped allowing the snapshot id to be set, accepting only *tag* as parameter.

`loadvm` ***tag*** Set the whole virtual machine to the snapshot identified by the tag *tag*.

Since 4.0, loadvm stopped accepting snapshot id as parameter.

`delvm` ***tag*** Delete the snapshot identified by *tag*.

Since 4.0, delvm stopped deleting snapshots by snapshot id, accepting only *tag* as parameter.

`singlestep [off]` Run the emulation in single step mode. If called with option off, the emulation returns to normal mode.

`stop` Stop emulation.

`c` **or** **`cont`** Resume emulation.

`system_wakeup` Wakeup guest from suspend.

`gdbserver` **[*port*]** Start gdbserver session (default *port*=1234)

`x/`***fmt addr*** Virtual memory dump starting at *addr*.

`xp` **`/`*****fmt addr*** Physical memory dump starting at *addr*.

fmt is a format which tells the command how to format the data. Its syntax is: `/{count}{format}{size}`

count is the number of items to be dumped.

format can be x (hex), d (signed decimal), u (unsigned decimal), o (octal), c (char) or i (asm instruction).

size can be b (8 bits), h (16 bits), w (32 bits) or g (64 bits). On x86, `h` or `w` can be specified with the `i` format to respectively select 16 or 32 bit code instruction size.

Examples:

Dump 10 instructions at the current instruction pointer:

```
(qemu) x/10i $eip
0x90107063:  ret
0x90107064:  sti
0x90107065:  lea    0x0(%esi,1),%esi
0x90107069:  lea    0x0(%edi,1),%edi
0x90107070:  ret
0x90107071:  jmp    0x90107080
0x90107073:  nop
0x90107074:  nop
0x90107075:  nop
0x90107076:  nop
```

Dump 80 16 bit values at the start of the video memory:

```
(qemu) xp/80hx 0xb8000
0x000b8000: 0x0b50 0x0b6c 0x0b65 0x0b78 0x0b38 0x0b36 0x0b2f 0x0b42
0x000b8010: 0x0b6f 0x0b63 0x0b68 0x0b73 0x0b20 0x0b56 0x0b47 0x0b41
0x000b8020: 0x0b42 0x0b69 0x0b6f 0x0b73 0x0b20 0x0b63 0x0b75 0x0b72
0x000b8030: 0x0b72 0x0b65 0x0b6e 0x0b74 0x0b2d 0x0b63 0x0b76 0x0b73
0x000b8040: 0x0b20 0x0b30 0x0b35 0x0b20 0x0b4e 0x0b6f 0x0b76 0x0b20
0x000b8050: 0x0b32 0x0b30 0x0b30 0x0b33 0x0720 0x0720 0x0720 0x0720
0x000b8060: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
0x000b8070: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
0x000b8080: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
0x000b8090: 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720 0x0720
```

`gpa2hva` ***addr*** Print the host virtual address at which the guest's physical address *addr* is mapped.

`gpa2hpa` ***addr*** Print the host physical address at which the guest's physical address *addr* is mapped.

`gva2gpa` ***addr*** Print the guest physical address at which the guest's virtual address *addr* is mapped based on the mapping for the current CPU.

`p` **or** **`print/`*****fmt expr*** Print expression value. Only the *format* part of *fmt* is used.

`i/`***fmt addr*** **[.*index*]** Read I/O port.

`o/`***fmt addr val*** Write to I/O port.

`sendkey` ***keys*** Send *keys* to the guest. *keys* could be the name of the key or the raw value in hexadecimal format. Use − to press several keys simultaneously. Example:

```
sendkey ctrl-alt-f1
```

This command is useful to send keys that your graphical user interface intercepts at low level, such as `ctrl-alt-f1` in X Window.

`sync-profile [on|off|reset]` Enable, disable or reset synchronization profiling. With no arguments, prints whether profiling is on or off.

`system_reset` Reset the system.

`system_powerdown` Power down the system (if supported).

`sum` ***addr size*** Compute the checksum of a memory region.

`device_add` ***config*** Add device.

`device_del` ***id*** Remove device *id*. *id* may be a short ID or a QOM object path.

`cpu` ***index*** Set the default CPU.

`mouse_move` ***dx dy*** **[*dz*]** Move the active mouse to the specified coordinates *dx dy* with optional scroll axis *dz*.

`mouse_button` ***val*** Change the active mouse button state *val* (1=L, 2=M, 4=R).

`mouse_set` ***index*** Set which mouse device receives events at given *index*, index can be obtained with:

```
info mice
```

`wavcapture` ***filename audiodev*** **[*frequency* [*bits* [*channels*]]]** Capture audio into *filename* from *audiodev*, using sample rate *frequency* bits per sample *bits* and number of channels *channels*.

Defaults:

- Sample rate = 44100 Hz - CD quality
- Bits = 16
- Number of channels = 2 - Stereo

`stopcapture` ***index*** Stop capture with a given *index*, index can be obtained with:

```
info capture
```

`memsave` ***addr size file*** save to disk virtual memory dump starting at *addr* of size *size*.

`pmemsave` ***addr size file*** save to disk physical memory dump starting at *addr* of size *size*.

`boot_set` ***bootdevicelist*** Define new values for the boot device list. Those values will override the values specified on the command line through the `-boot` option.

The values that can be specified here depend on the machine type, but are the same that can be specified in the `-boot` command line option.

`nmi` ***cpu*** Inject an NMI on the default CPU (x86/s390) or all CPUs (ppc64).

`ringbuf_write` ***device data*** Write *data* to ring buffer character device *device*. *data* must be a UTF-8 string.

`ringbuf_read` ***device*** Read and print up to *size* bytes from ring buffer character device *device*. Certain non-printable characters are printed `\uXXXX`, where `XXXX` is the character code in hexadecimal. Character `\` is printed `\\`. Bug: can screw up when the buffer contains invalid UTF-8 sequences, NUL characters, after the ring buffer lost data, and when reading stops because the size limit is reached.

`announce_self` Trigger a round of GARP/RARP broadcasts; this is useful for explicitly updating the network infrastructure after a reconfiguration or some forms of migration. The timings of the round are set by the migration announce parameters. An optional comma separated *interfaces* list restricts the announce to the named set of interfaces. An optional *id* can be used to start a separate announce timer and to change the parameters of it later.

`migrate [-d] [-b] [-i]` ***uri*** Migrate to *uri* (using -d to not wait for completion).

`-b` for migration with full copy of disk

`-i` for migration with incremental copy of disk (base image is shared)

`migrate_cancel` Cancel the current VM migration.

`migrate_continue` ***state*** Continue migration from the paused state *state*

`migrate_incoming` ***uri*** Continue an incoming migration using the *uri* (that has the same syntax as the `-incoming` option).

`migrate_recover` ***uri*** Continue a paused incoming postcopy migration using the *uri*.

`migrate_pause` Pause an ongoing migration. Currently it only supports postcopy.

`migrate_set_cache_size` ***value*** Set cache size to *value* (in bytes) for xbzrle migrations.

`migrate_set_speed` ***value*** Set maximum speed to *value* (in bytes) for migrations.

`migrate_set_downtime` ***second*** Set maximum tolerated downtime (in seconds) for migration.

`migrate_set_capability` ***capability state*** Enable/Disable the usage of a capability *capability* for migration.

`migrate_set_parameter` ***parameter value*** Set the parameter *parameter* for migration.

`migrate_start_postcopy` Switch in-progress migration to postcopy mode. Ignored after the end of migration (or once already in postcopy).

`x_colo_lost_heartbeat` Tell COLO that heartbeat is lost, a failover or takeover is needed.

`client_migrate_info` ***protocol hostname port tls-port cert-subject*** Set migration information for remote display. This makes the server ask the client to automatically reconnect using the new parameters once migration finished successfully. Only implemented for SPICE.

`dump-guest-memory [-p]` ***filename begin length***

`dump-guest-memory [-z|-l|-s|-w]` ***filename*** Dump guest memory to *protocol*. The file can be processed with crash or gdb. Without `-z|-l|-s|-w`, the dump format is ELF.

`-p` do paging to get guest's memory mapping.

`-z` dump in kdump-compressed format, with zlib compression.

`-l` dump in kdump-compressed format, with lzo compression.

`-s` dump in kdump-compressed format, with snappy compression.

`-w` dump in Windows crashdump format (can be used instead of ELF-dump converting), for Windows x64 guests with vmcoreinfo driver only

filename dump file name.

begin the starting physical address. It's optional, and should be specified together with *length*.

length the memory size, in bytes. It's optional, and should be specified together with *begin*.

`dump-skeys` ***filename*** Save guest storage keys to a file.

`migration_mode` ***mode*** Enables or disables migration mode.

`snapshot_blkdev` Snapshot device, using snapshot file as target if provided

`snapshot_blkdev_internal` Take an internal snapshot on device if it support

`snapshot_delete_blkdev_internal` Delete an internal snapshot on device if it support

`drive_mirror` Start mirroring a block device's writes to a new destination, using the specified target.

`drive_backup` Start a point-in-time copy of a block device to a specificed target.

`drive_add` Add drive to PCI storage controller.

`pcie_aer_inject_error` Inject PCIe AER error

`netdev_add` Add host network device.

`netdev_del` Remove host network device.

`object_add` Create QOM object.

`object_del` Destroy QOM object.

`hostfwd_add` Redirect TCP or UDP connections from host to guest (requires -net user).

`hostfwd_remove` Remove host-to-guest TCP or UDP redirection.

`balloon` ***value*** Request VM to change its memory allocation to *value* (in MB).

`set_link` ***name*** **`[on|off]`** Switch link *name* on (i.e. up) or off (i.e. down).

`watchdog_action` Change watchdog action.

`acl_show` ***aclname*** List all the matching rules in the access control list, and the default policy. There are currently two named access control lists, *vnc.x509dname* and *vnc.username* matching on the x509 client certificate distinguished name, and SASL username respectively.

`acl_policy` ***aclname*** **`allow|deny`** Set the default access control list policy, used in the event that none of the explicit rules match. The default policy at startup is always `deny`.

`acl_add` ***aclname match*** **`allow|deny`** **[*index*]** Add a match rule to the access control list, allowing or denying access. The match will normally be an exact username or x509 distinguished name, but can optionally include wildcard globs. eg `*@EXAMPLE.COM` to allow all users in the `EXAMPLE.COM` kerberos realm. The match will normally be appended to the end of the ACL, but can be inserted earlier in the list if the optional *index* parameter is supplied.

`acl_remove` ***aclname match*** Remove the specified match rule from the access control list.

`acl_reset` ***aclname*** Remove all matches from the access control list, and set the default policy back to `deny`.

`nbd_server_start` ***host*:*port*** Start an NBD server on the given host and/or port. If the `-a` option is included, all of the virtual machine's block devices that have an inserted media on them are automatically exported; in this case, the `-w` option makes the devices writable too.

`nbd_server_add` ***device*** **[*name*]** Export a block device through QEMU's NBD server, which must be started beforehand with `nbd_server_start`. The `-w` option makes the exported device writable too. The export name is controlled by *name*, defaulting to *device*.

`nbd_server_remove [-f]` ***name*** Stop exporting a block device through QEMU's NBD server, which was previously started with `nbd_server_add`. The `-f` option forces the server to drop the export immediately even if clients are connected; otherwise the command fails unless there are no clients.

`nbd_server_stop` Stop the QEMU embedded NBD server.

`mce` ***cpu bank status mcgstatus addr misc*** Inject an MCE on the given CPU (x86 only).

`getfd` ***fdname*** If a file descriptor is passed alongside this command using the SCM_RIGHTS mechanism on unix sockets, it is stored using the name *fdname* for later use by other monitor commands.

`closefd` ***fdname*** Close the file descriptor previously assigned to *fdname* using the `getfd` command. This is only needed if the file descriptor was never used by another monitor command.

`block_passwd` ***device password*** Set the encrypted device *device* password to *password*

This command is now obsolete and will always return an error since 2.10

`block_set_io_throttle` ***device bps bps_rd bps_wr iops iops_rd iops_wr*** Change I/O throttle limits for a block drive to *bps bps_rd bps_wr iops iops_rd iops_wr*. *device* can be a block device name, a qdev ID or a QOM path.

`set_password [ vnc | spice ] password [ action-if-connected ]` Change spice/vnc password. Use zero to make the password stay valid forever. *action-if-connected* specifies what should happen in case a connection is established: *fail* makes the password change fail. *disconnect* changes the password and disconnects the client. *keep* changes the password and keeps the connection up. *keep* is the default.

`expire_password [ vnc | spice ]` ***expire-time*** Specify when a password for spice/vnc becomes invalid. *expire-time* accepts:

`now` Invalidate password instantly.

`never` Password stays valid forever.

+*nsec* Password stays valid for *nsec* seconds starting now.

nsec Password is invalidated at the given time. *nsec* are the seconds passed since 1970, i.e. unix epoch.

`chardev-add` ***args*** chardev-add accepts the same parameters as the -chardev command line switch.

`chardev-change` ***args*** chardev-change accepts existing chardev *id* and then the same arguments as the -chardev command line switch (except for "id").

`chardev-remove` ***id*** Removes the chardev *id*.

`chardev-send-break` ***id*** Send a break on the chardev *id*.

`qemu-io` ***device command*** Executes a qemu-io command on the given block device.

`cpu-add` ***id*** Add CPU with id *id*. This command is deprecated, please +use `device_add` instead. For details, refer to 'docs/cpu-hotplug.rst'.

`qom-list` **[*path*]** Print QOM properties of object at location *path*

`qom-get` ***path property*** Print QOM property *property* of object at location *path*

`qom-set` ***path property value*** Set QOM property *property* of object at location *path* to value *value*

`info` ***subcommand*** Show various information about the system state.

`info version` Show the version of QEMU.

`info network` Show the network state.

`info chardev` Show the character devices.

`info block` Show info of one block device or all block devices.

`info blockstats` Show block device statistics.

`info block-jobs` Show progress of ongoing block device operations.

`info registers` Show the cpu registers.

`info lapic` Show local APIC state

`info ioapic` Show io APIC state

`info cpus` Show infos for each CPU.

`info history` Show the command line history.

`info irq` Show the interrupts statistics (if available).

info pic Show PIC state.

info rdma Show RDMA state.

info pci Show PCI information.

info tlb Show virtual to physical memory mappings.

info mem Show the active virtual memory mappings.

info mtree Show memory tree.

info jit Show dynamic compiler info.

info opcount Show dynamic compiler opcode counters

info sync-profile [-m|-n] [*max*] Show synchronization profiling info, up to *max* entries (default: 10), sorted by total wait time.

-m sort by mean wait time

-n do not coalesce objects with the same call site

When different objects that share the same call site are coalesced, the "Object" field shows—enclosed in brackets—the number of objects being coalesced.

info kvm Show KVM information.

info numa Show NUMA information.

info usb Show guest USB devices.

info usbhost Show host USB devices.

info profile Show profiling information.

info capture Show capture information.

info snapshots Show the currently saved VM snapshots.

info status Show the current VM status (running|paused).

info mice Show which guest mouse is receiving events.

info vnc Show the vnc server status.

info spice Show the spice server status.

info name Show the current VM name.

info uuid Show the current VM UUID.

info cpustats Show CPU statistics.

info usernet Show user network stack connection states.

info migrate Show migration status.

info migrate_capabilities Show current migration capabilities.

info migrate_parameters Show current migration parameters.

info migrate_cache_size Show current migration xbzrle cache size.

info balloon Show balloon information.

info qtree Show device tree.

info qdm Show qdev device model list.

info qom-tree Show QOM composition tree.

info roms Show roms.

info trace-events Show available trace-events & their state.

info tpm Show the TPM device.

info memdev Show memory backends

info memory-devices Show memory devices.

info iothreads Show iothread's identifiers.

info rocker *name* Show rocker switch.

info rocker-ports *name*-ports Show rocker ports.

info rocker-of-dpa-flows *name* [*tbl_id*] Show rocker OF-DPA flow tables.

info rocker-of-dpa-groups *name* [*type*] Show rocker OF-DPA groups.

info skeys *address* Display the value of a storage key (s390 only)

info cmma *address* Display the values of the CMMA storage attributes for a range of pages (s390 only)

info dump Display the latest dump status.

info ramblock Dump all the ramblocks of the system.

info hotpluggable-cpus Show information about hotpluggable CPUs

info vm-generation-id Show Virtual Machine Generation ID

info memory_size_summary Display the amount of initially allocated and present hotpluggable (if enabled) memory in bytes.

info sev Show SEV information.

1.5.2 Integer expressions

The monitor understands integers expressions for every integer argument. You can use register names to get the value of specifics CPU registers by prefixing them with *$*.

1.6 Disk Images

QEMU supports many disk image formats, including growable disk images (their size increase as non empty sectors are written), compressed and encrypted disk images.

1.6.1 Quick start for disk image creation

You can create a disk image with the command:

```
qemu-img create myimage.img mysize
```

where myimage.img is the disk image filename and mysize is its size in kilobytes. You can add an `M` suffix to give the size in megabytes and a `G` suffix for gigabytes.

See the qemu-img invocation documentation for more information.

1.6.2 Snapshot mode

If you use the option `-snapshot`, all disk images are considered as read only. When sectors in written, they are written in a temporary file created in `/tmp`. You can however force the write back to the raw disk images by using the `commit` monitor command (or C-a s in the serial console).

1.6.3 VM snapshots

VM snapshots are snapshots of the complete virtual machine including CPU state, RAM, device state and the content of all the writable disks. In order to use VM snapshots, you must have at least one non removable and writable block device using the `qcow2` disk image format. Normally this device is the first virtual hard drive.

Use the monitor command `savevm` to create a new VM snapshot or replace an existing one. A human readable name can be assigned to each snapshot in addition to its numerical ID.

Use `loadvm` to restore a VM snapshot and `delvm` to remove a VM snapshot. `info snapshots` lists the available snapshots with their associated information:

```
(qemu) info snapshots
Snapshot devices: hda
Snapshot list (from hda):
ID        TAG                 VM SIZE                DATE       VM CLOCK
1         start                   41M 2006-08-06 12:38:02   00:00:14.954
2                                 40M 2006-08-06 12:43:29   00:00:18.633
3         msys                    40M 2006-08-06 12:44:04   00:00:23.514
```

A VM snapshot is made of a VM state info (its size is shown in `info snapshots`) and a snapshot of every writable disk image. The VM state info is stored in the first `qcow2` non removable and writable block device. The disk image snapshots are stored in every disk image. The size of a snapshot in a disk image is difficult to evaluate and is not shown by `info snapshots` because the associated disk sectors are shared among all the snapshots to save disk space (otherwise each snapshot would need a full copy of all the disk images).

When using the (unrelated) `-snapshot` option (*Snapshot mode*), you can always make VM snapshots, but they are deleted as soon as you exit QEMU.

VM snapshots currently have the following known limitations:

- They cannot cope with removable devices if they are removed or inserted after a snapshot is done.
- A few device drivers still have incomplete snapshot support so their state is not saved or restored properly (in particular USB).

1.6.4 Disk image file formats

QEMU supports many image file formats that can be used with VMs as well as with any of the tools (like `qemu-img`). This includes the preferred formats raw and qcow2 as well as formats that are supported for compatibility with older QEMU versions or other hypervisors.

Depending on the image format, different options can be passed to `qemu-img create` and `qemu-img convert` using the `-o` option. This section describes each format and the options that are supported for it.

`raw`

Raw disk image format. This format has the advantage of being simple and easily exportable to all other emulators. If your file system supports *holes* (for example in ext2 or ext3 on Linux or NTFS on Windows), then only the written sectors will reserve space. Use `qemu-img info` to know the real size used by the image or `ls -ls` on Unix/Linux.

Supported options:

preallocation
Preallocation mode (allowed values: `off`, `falloc`, `full`). `falloc` mode preallocates space for image by calling `posix_fallocate()`. `full` mode preallocates space for image by writing data to underlying storage. This data may or may not be zero, depending on the storage location.

qcow2
QEMU image format, the most versatile format. Use it to have smaller images (useful if your filesystem does not supports holes, for example on Windows), zlib based compression and support of multiple VM snapshots.

Supported options:

compat
Determines the qcow2 version to use. `compat=0.10` uses the traditional image format that can be read by any QEMU since 0.10. `compat=1.1` enables image format extensions that only QEMU 1.1 and newer understand (this is the default). Amongst others, this includes zero clusters, which allow efficient copy-on-read for sparse images.

backing_file
File name of a base image (see `create` subcommand)

backing_fmt
Image format of the base image

encryption
This option is deprecated and equivalent to `encrypt.format=aes`

encrypt.format
If this is set to `luks`, it requests that the qcow2 payload (not qcow2 header) be encrypted using the LUKS format. The passphrase to use to unlock the LUKS key slot is given by the `encrypt.key-secret` parameter. LUKS encryption parameters can be tuned with the other `encrypt.*` parameters.

If this is set to `aes`, the image is encrypted with 128-bit AES-CBC. The encryption key is given by the `encrypt.key-secret` parameter. This encryption format is considered to be flawed by modern cryptography standards, suffering from a number of design problems:

- The AES-CBC cipher is used with predictable initialization vectors based on the sector number. This makes it vulnerable to chosen plaintext attacks which can reveal the existence of encrypted data.
- The user passphrase is directly used as the encryption key. A poorly chosen or short passphrase will compromise the security of the encryption.
- In the event of the passphrase being compromised there is no way to change the passphrase to protect data in any qcow images. The files must be cloned, using a different encryption passphrase in the new file. The original file must then be securely erased using a program like shred, though even this is ineffective with many modern storage technologies.

The use of this is no longer supported in system emulators. Support only remains in the command line utilities, for the purposes of data liberation and interoperability with old versions of QEMU. The `luks` format should be used instead.

encrypt.key-secret
Provides the ID of a `secret` object that contains the passphrase (`encrypt.format=luks`) or encryption key (`encrypt.format=aes`).

encrypt.cipher-alg
Name of the cipher algorithm and key length. Currently defaults to `aes-256`. Only used when `encrypt.format=luks`.

encrypt.cipher-mode
Name of the encryption mode to use. Currently defaults to `xts`. Only used when `encrypt.format=luks`.

encrypt.ivgen-alg
Name of the initialization vector generator algorithm. Currently defaults to `plain64`. Only used when `encrypt.format=luks`.

encrypt.ivgen-hash-alg
Name of the hash algorithm to use with the initialization vector generator (if required). Defaults to `sha256`. Only used when `encrypt.format=luks`.

encrypt.hash-alg
Name of the hash algorithm to use for PBKDF algorithm Defaults to `sha256`. Only used when `encrypt.format=luks`.

encrypt.iter-time
Amount of time, in milliseconds, to use for PBKDF algorithm per key slot. Defaults to `2000`. Only used when `encrypt.format=luks`.

cluster_size
Changes the qcow2 cluster size (must be between 512 and 2M). Smaller cluster sizes can improve the image file size whereas larger cluster sizes generally provide better performance.

preallocation
Preallocation mode (allowed values: `off`, `metadata`, `falloc`, `full`). An image with preallocated metadata is initially larger but can improve performance when the image needs to grow. `falloc` and `full` preallocations are like the same options of `raw` format, but sets up metadata also.

lazy_refcounts
If this option is set to `on`, reference count updates are postponed with the goal of avoiding metadata I/O and improving performance. This is particularly interesting with `cache=writethrough` which doesn't batch metadata updates. The tradeoff is that after a host crash, the reference count tables must be rebuilt, i.e. on the next open an (automatic) `qemu-img check -r all` is required, which may take some time.

This option can only be enabled if `compat=1.1` is specified.

nocow
If this option is set to `on`, it will turn off COW of the file. It's only valid on btrfs, no effect on other file systems.

Btrfs has low performance when hosting a VM image file, even more when the guest on the VM also using btrfs as file system. Turning off COW is a way to mitigate this bad performance. Generally there are two ways to turn off COW on btrfs:

- Disable it by mounting with nodatacow, then all newly created files will be NOCOW.
- For an empty file, add the NOCOW file attribute. That's what this option does.

Note: this option is only valid to new or empty files. If there is an existing file which is COW and has data blocks already, it couldn't be changed to NOCOW by setting `nocow=on`. One can issue `lsattr filename` to check if the NOCOW flag is set or not (Capital 'C' is NOCOW flag).

qed
Old QEMU image format with support for backing files and compact image files (when your filesystem or transport medium does not support holes).

When converting QED images to qcow2, you might want to consider using the `lazy_refcounts=on` option to get a more QED-like behaviour.

Supported options:

backing_file
File name of a base image (see `create` subcommand).

backing_fmt
Image file format of backing file (optional). Useful if the format cannot be autodetected because it has no header, like some vhd/vpc files.

cluster_size
Changes the cluster size (must be power-of-2 between 4K and 64K). Smaller cluster sizes can improve the image file size whereas larger cluster sizes generally provide better performance.

table_size
Changes the number of clusters per L1/L2 table (must be power-of-2 between 1 and 16). There is normally no need to change this value but this option can between used for performance benchmarking.

qcow
Old QEMU image format with support for backing files, compact image files, encryption and compression.

Supported options:

backing_file
File name of a base image (see `create` subcommand)

encryption
This option is deprecated and equivalent to `encrypt.format=aes`

encrypt.format
If this is set to `aes`, the image is encrypted with 128-bit AES-CBC. The encryption key is given by the `encrypt.key-secret` parameter. This encryption format is considered to be flawed by modern cryptography standards, suffering from a number of design problems enumerated previously against the `qcow2` image format.

The use of this is no longer supported in system emulators. Support only remains in the command line utilities, for the purposes of data liberation and interoperability with old versions of QEMU.

Users requiring native encryption should use the `qcow2` format instead with `encrypt.format=luks`.

encrypt.key-secret
Provides the ID of a `secret` object that contains the encryption key (`encrypt.format=aes`).

luks
LUKS v1 encryption format, compatible with Linux dm-crypt/cryptsetup

Supported options:

key-secret
Provides the ID of a `secret` object that contains the passphrase.

cipher-alg
Name of the cipher algorithm and key length. Currently defaults to `aes-256`.

cipher-mode
Name of the encryption mode to use. Currently defaults to `xts`.

ivgen-alg
Name of the initialization vector generator algorithm. Currently defaults to `plain64`.

ivgen-hash-alg
Name of the hash algorithm to use with the initialization vector generator (if required). Defaults to `sha256`.

hash-alg
Name of the hash algorithm to use for PBKDF algorithm Defaults to `sha256`.

iter-time
    Amount of time, in milliseconds, to use for PBKDF algorithm per key slot. Defaults to `2000`.

vdi
VirtualBox 1.1 compatible image format.

Supported options:

static
    If this option is set to `on`, the image is created with metadata preallocation.

vmdk
VMware 3 and 4 compatible image format.

Supported options:

backing_file
    File name of a base image (see `create` subcommand).

compat6
    Create a VMDK version 6 image (instead of version 4)

hwversion
    Specify vmdk virtual hardware version. Compat6 flag cannot be enabled if hwversion is specified.

subformat
    Specifies which VMDK subformat to use. Valid options are `monolithicSparse` (default), `monolithicFlat`, `twoGbMaxExtentSparse`, `twoGbMaxExtentFlat` and `streamOptimized`.

vpc
VirtualPC compatible image format (VHD).

Supported options:

subformat
    Specifies which VHD subformat to use. Valid options are `dynamic` (default) and `fixed`.

VHDX
Hyper-V compatible image format (VHDX).

Supported options:

subformat
    Specifies which VHDX subformat to use. Valid options are `dynamic` (default) and `fixed`.

block_state_zero
    Force use of payload blocks of type 'ZERO'. Can be set to `on` (default) or `off`. When set to `off`, new blocks will be created as `PAYLOAD_BLOCK_NOT_PRESENT`, which means parsers are free to return arbitrary data for those blocks. Do not set to `off` when using `qemu-img convert` with `subformat=dynamic`.

block_size
    Block size; min 1 MB, max 256 MB. 0 means auto-calculate based on image size.

log_size
    Log size; min 1 MB.

1.6.5 Read-only formats

More disk image file formats are supported in a read-only mode.

`bochs`
Bochs images of `growing` type.

`cloop`
Linux Compressed Loop image, useful only to reuse directly compressed CD-ROM images present for example in the Knoppix CD-ROMs.

`dmg`
Apple disk image.

`parallels`
Parallels disk image format.

1.6.6 Using host drives

In addition to disk image files, QEMU can directly access host devices. We describe here the usage for QEMU version >= 0.8.3.

Linux

On Linux, you can directly use the host device filename instead of a disk image filename provided you have enough privileges to access it. For example, use `/dev/cdrom` to access to the CDROM.

CD You can specify a CDROM device even if no CDROM is loaded. QEMU has specific code to detect CDROM insertion or removal. CDROM ejection by the guest OS is supported. Currently only data CDs are supported.

Floppy You can specify a floppy device even if no floppy is loaded. Floppy removal is currently not detected accurately (if you change floppy without doing floppy access while the floppy is not loaded, the guest OS will think that the same floppy is loaded). Use of the host's floppy device is deprecated, and support for it will be removed in a future release.

Hard disks Hard disks can be used. Normally you must specify the whole disk (`/dev/hdb` instead of `/dev/hdb1`) so that the guest OS can see it as a partitioned disk. WARNING: unless you know what you do, it is better to only make READ-ONLY accesses to the hard disk otherwise you may corrupt your host data (use the `-snapshot` command line option or modify the device permissions accordingly).

Windows

CD The preferred syntax is the drive letter (e.g. `d:`). The alternate syntax `\\.\d:` is supported. `/dev/cdrom` is supported as an alias to the first CDROM drive.

Currently there is no specific code to handle removable media, so it is better to use the `change` or `eject` monitor commands to change or eject media.

Hard disks Hard disks can be used with the syntax: `\\.\PhysicalDriveN` where *N* is the drive number (0 is the first hard disk).

WARNING: unless you know what you do, it is better to only make READ-ONLY accesses to the hard disk otherwise you may corrupt your host data (use the `-snapshot` command line so that the modifications are written in a temporary file).

Mac OS X

`/dev/cdrom` is an alias to the first CDROM.

Currently there is no specific code to handle removable media, so it is better to use the `change` or `eject` monitor commands to change or eject media.

1.6.7 Virtual FAT disk images

QEMU can automatically create a virtual FAT disk image from a directory tree. In order to use it, just type:

```
qemu-system-x86_64 linux.img -hdb fat:/my_directory
```

Then you access access to all the files in the `/my_directory` directory without having to copy them in a disk image or to export them via SAMBA or NFS. The default access is *read-only*.

Floppies can be emulated with the `:floppy:` option:

```
qemu-system-x86_64 linux.img -fda fat:floppy:/my_directory
```

A read/write support is available for testing (beta stage) with the `:rw:` option:

```
qemu-system-x86_64 linux.img -fda fat:floppy:rw:/my_directory
```

What you should *never* do:

- use non-ASCII filenames
- use "-snapshot" together with ":rw:"
- expect it to work when loadvm'ing
- write to the FAT directory on the host system while accessing it with the guest system

1.6.8 NBD access

QEMU can access directly to block device exported using the Network Block Device protocol.

```
qemu-system-x86_64 linux.img -hdb nbd://my_nbd_server.mydomain.org:1024/
```

If the NBD server is located on the same host, you can use an unix socket instead of an inet socket:

```
qemu-system-x86_64 linux.img -hdb nbd+unix://?socket=/tmp/my_socket
```

In this case, the block device must be exported using qemu-nbd:

```
qemu-nbd --socket=/tmp/my_socket my_disk.qcow2
```

The use of qemu-nbd allows sharing of a disk between several guests:

```
qemu-nbd --socket=/tmp/my_socket --share=2 my_disk.qcow2
```

and then you can use it with two guests:

```
qemu-system-x86_64 linux1.img -hdb nbd+unix://?socket=/tmp/my_socket
qemu-system-x86_64 linux2.img -hdb nbd+unix://?socket=/tmp/my_socket
```

If the nbd-server uses named exports (supported since NBD 2.9.18, or with QEMU's own embedded NBD server), you must specify an export name in the URI:

```
qemu-system-x86_64 -cdrom nbd://localhost/debian-500-ppc-netinst
qemu-system-x86_64 -cdrom nbd://localhost/openSUSE-11.1-ppc-netinst
```

The URI syntax for NBD is supported since QEMU 1.3. An alternative syntax is also available. Here are some example of the older syntax:

```
qemu-system-x86_64 linux.img -hdb nbd:my_nbd_server.mydomain.org:1024
qemu-system-x86_64 linux2.img -hdb nbd:unix:/tmp/my_socket
qemu-system-x86_64 -cdrom nbd:localhost:10809:exportname=debian-500-ppc-
↪netinst
```

1.6.9 Sheepdog disk images

Sheepdog is a distributed storage system for QEMU. It provides highly available block level storage volumes that can be attached to QEMU-based virtual machines.

You can create a Sheepdog disk image with the command:

```
qemu-img create sheepdog:///IMAGE SIZE
```

where *IMAGE* is the Sheepdog image name and *SIZE* is its size.

To import the existing *FILENAME* to Sheepdog, you can use a convert command.

```
qemu-img convert FILENAME sheepdog:///IMAGE
```

You can boot from the Sheepdog disk image with the command:

```
qemu-system-x86_64 sheepdog:///IMAGE
```

You can also create a snapshot of the Sheepdog image like qcow2.

```
qemu-img snapshot -c TAG sheepdog:///IMAGE
```

where *TAG* is a tag name of the newly created snapshot.

To boot from the Sheepdog snapshot, specify the tag name of the snapshot.

```
qemu-system-x86_64 sheepdog:///IMAGE#TAG
```

You can create a cloned image from the existing snapshot.

```
qemu-img create -b sheepdog:///BASE#TAG sheepdog:///IMAGE
```

where *BASE* is an image name of the source snapshot and *TAG* is its tag name.

You can use an unix socket instead of an inet socket:

```
qemu-system-x86_64 sheepdog+unix:///IMAGE?socket=PATH
```

If the Sheepdog daemon doesn't run on the local host, you need to specify one of the Sheepdog servers to connect to.

```
qemu-img create sheepdog://HOSTNAME:PORT/IMAGE SIZE
qemu-system-x86_64 sheepdog://HOSTNAME:PORT/IMAGE
```

1.6.10 iSCSI LUNs

iSCSI is a popular protocol used to access SCSI devices across a computer network.

There are two different ways iSCSI devices can be used by QEMU.

The first method is to mount the iSCSI LUN on the host, and make it appear as any other ordinary SCSI device on the host and then to access this device as a /dev/sd device from QEMU. How to do this differs between host OSes.

The second method involves using the iSCSI initiator that is built into QEMU. This provides a mechanism that works the same way regardless of which host OS you are running QEMU on. This section will describe this second method of using iSCSI together with QEMU.

In QEMU, iSCSI devices are described using special iSCSI URLs. URL syntax:

```
iscsi://[<username>[%<password>]@]<host>[:<port>]/<target-iqn-name>/<lun>
```

Username and password are optional and only used if your target is set up using CHAP authentication for access control. Alternatively the username and password can also be set via environment variables to have these not show up in the process list:

```
export LIBISCSI_CHAP_USERNAME=<username>
export LIBISCSI_CHAP_PASSWORD=<password>
iscsi://<host>/<target-iqn-name>/<lun>
```

Various session related parameters can be set via special options, either in a configuration file provided via '-readconfig' or directly on the command line.

If the initiator-name is not specified qemu will use a default name of 'iqn.2008-11.org.linux-kvm[:<uuid>'] where <uuid> is the UUID of the virtual machine. If the UUID is not specified qemu will use 'iqn.2008-11.org.linux-kvm[:<name>'] where <name> is the name of the virtual machine.

Setting a specific initiator name to use when logging in to the target:

```
-iscsi initiator-name=iqn.qemu.test:my-initiator
```

Controlling which type of header digest to negotiate with the target:

```
-iscsi header-digest=CRC32C|CRC32C-NONE|NONE-CRC32C|NONE
```

These can also be set via a configuration file:

```
[iscsi]
  user = "CHAP username"
  password = "CHAP password"
  initiator-name = "iqn.qemu.test:my-initiator"
  # header digest is one of CRC32C|CRC32C-NONE|NONE-CRC32C|NONE
  header-digest = "CRC32C"
```

Setting the target name allows different options for different targets:

```
[iscsi "iqn.target.name"]
  user = "CHAP username"
  password = "CHAP password"
  initiator-name = "iqn.qemu.test:my-initiator"
  # header digest is one of CRC32C|CRC32C-NONE|NONE-CRC32C|NONE
  header-digest = "CRC32C"
```

How to use a configuration file to set iSCSI configuration options:

```
cat >iscsi.conf <<EOF
[iscsi]
  user = "me"
  password = "my password"
  initiator-name = "iqn.qemu.test:my-initiator"
  header-digest = "CRC32C"
EOF

qemu-system-x86_64 -drive file=iscsi://127.0.0.1/iqn.qemu.test/1 \
  -readconfig iscsi.conf
```

How to set up a simple iSCSI target on loopback and access it via QEMU: this example shows how to set up an iSCSI target with one CDROM and one DISK using the Linux STGT software target. This target is available on Red Hat based systems as the package 'scsi-target-utils'.

```
tgtd --iscsi portal=127.0.0.1:3260
tgtadm --lld iscsi --op new --mode target --tid 1 -T iqn.qemu.test
tgtadm --lld iscsi --mode logicalunit --op new --tid 1 --lun 1 \
    -b /IMAGES/disk.img --device-type=disk
tgtadm --lld iscsi --mode logicalunit --op new --tid 1 --lun 2 \
    -b /IMAGES/cd.iso --device-type=cd
tgtadm --lld iscsi --op bind --mode target --tid 1 -I ALL

qemu-system-x86_64 -iscsi initiator-name=iqn.qemu.test:my-initiator \
  -boot d -drive file=iscsi://127.0.0.1/iqn.qemu.test/1 \
  -cdrom iscsi://127.0.0.1/iqn.qemu.test/2
```

1.6.11 GlusterFS disk images

GlusterFS is a user space distributed file system.

You can boot from the GlusterFS disk image with the command:

URI:

```
qemu-system-x86_64 -drive file=gluster[+TYPE]://[HOST}[:PORT]]/VOLUME/PATH
                             [?socket=...][,file.debug=9][,file.logfile=...]
```

JSON:

```
qemu-system-x86_64 'json:{"driver":"qcow2",
                          "file":{"driver":"gluster",
                                   "volume":"testvol","path":"a.img","debug":9,
↪"logfile":"...",
                                   "server":[{"type":"tcp","host":"...",
↪"port":"..."},
                                             {"type":"unix","socket":"..."}]}}'
```

gluster is the protocol.

TYPE specifies the transport type used to connect to gluster management daemon (glusterd). Valid transport types are tcp and unix. In the URI form, if a transport type isn't specified, then tcp type is assumed.

HOST specifies the server where the volume file specification for the given volume resides. This can be either a hostname or an ipv4 address. If transport type is unix, then *HOST* field should not be specified. Instead *socket* field needs to be populated with the path to unix domain socket.

PORT is the port number on which glusterd is listening. This is optional and if not specified, it defaults to port 24007. If the transport type is unix, then *PORT* should not be specified.

VOLUME is the name of the gluster volume which contains the disk image.

PATH is the path to the actual disk image that resides on gluster volume.

debug is the logging level of the gluster protocol driver. Debug levels are 0-9, with 9 being the most verbose, and 0 representing no debugging output. The default level is 4. The current logging levels defined in the gluster source are 0 - None, 1 - Emergency, 2 - Alert, 3 - Critical, 4 - Error, 5 - Warning, 6 - Notice, 7 - Info, 8 - Debug, 9 - Trace

logfile is a commandline option to mention log file path which helps in logging to the specified file and also help in persisting the gfapi logs. The default is stderr.

You can create a GlusterFS disk image with the command:

```
qemu-img create gluster://HOST/VOLUME/PATH SIZE
```

Examples

```
qemu-system-x86_64 -drive file=gluster://1.2.3.4/testvol/a.img
qemu-system-x86_64 -drive file=gluster+tcp://1.2.3.4/testvol/a.img
qemu-system-x86_64 -drive file=gluster+tcp://1.2.3.4:24007/testvol/dir/a.img
qemu-system-x86_64 -drive file=gluster+tcp://[1:2:3:4:5:6:7:8]/testvol/dir/a.
↪img
qemu-system-x86_64 -drive file=gluster+tcp://[1:2:3:4:5:6:7:8]:24007/testvol/
↪dir/a.img
qemu-system-x86_64 -drive file=gluster+tcp://server.domain.com:24007/testvol/
↪dir/a.img
qemu-system-x86_64 -drive file=gluster+unix:///testvol/dir/a.img?socket=/tmp/
↪glusterd.socket
qemu-system-x86_64 -drive file=gluster+rdma://1.2.3.4:24007/testvol/a.img
qemu-system-x86_64 -drive file=gluster://1.2.3.4/testvol/a.img,file.debug=9,
↪file.logfile=/var/log/qemu-gluster.log
qemu-system-x86_64 'json:{"driver":"qcow2",
                          "file":{"driver":"gluster",
                                   "volume":"testvol","path":"a.img",
                                   "debug":9,"logfile":"/var/log/qemu-gluster.
↪log",
                                   "server":[{"type":"tcp","host":"1.2.3.4",
↪"port":24007},
                                             {"type":"unix","socket":"/var/run/
↪glusterd.socket"}]}}'
qemu-system-x86_64 -drive driver=qcow2,file.driver=gluster,file.
↪volume=testvol,file.path=/path/a.img,
                                      file.debug=9,file.logfile=/var/log/qemu-
↪gluster.log,
                                      file.server.0.type=tcp,file.server.0.
↪host=1.2.3.4,file.server.0.port=24007,
                                      file.server.1.type=unix,file.server.1.
↪socket=/var/run/glusterd.socket
```

1.6.12 Secure Shell (ssh) disk images

You can access disk images located on a remote ssh server by using the ssh protocol:

```
qemu-system-x86_64 -drive file=ssh://[USER@]SERVER[:PORT]/PATH[?host_key_
↪check=HOST_KEY_CHECK]
```

Alternative syntax using properties:

```
qemu-system-x86_64 -drive file.driver=ssh[,file.user=USER],file.host=SERVER[,
↪file.port=PORT],file.path=PATH[,file.host_key_check=HOST_KEY_CHECK]
```

ssh is the protocol.

USER is the remote user. If not specified, then the local username is tried.

SERVER specifies the remote ssh server. Any ssh server can be used, but it must implement the sftp-server protocol. Most Unix/Linux systems should work without requiring any extra configuration.

PORT is the port number on which sshd is listening. By default the standard ssh port (22) is used.

PATH is the path to the disk image.

The optional *HOST_KEY_CHECK* parameter controls how the remote host's key is checked. The default is `yes` which means to use the local `.ssh/known_hosts` file. Setting this to `no` turns off known-hosts checking. Or you can check that the host key matches a specific fingerprint: `host_key_check=md5:78:45:8e:14:57:4f:d5:45:83:0a:0e:f3:49:82:c9:c8` (`sha1:` can also be used as a prefix, but note that OpenSSH tools only use MD5 to print fingerprints).

Currently authentication must be done using ssh-agent. Other authentication methods may be supported in future.

Note: Many ssh servers do not support an `fsync`-style operation. The ssh driver cannot guarantee that disk flush requests are obeyed, and this causes a risk of disk corruption if the remote server or network goes down during writes. The driver will print a warning when `fsync` is not supported:

```
warning: ssh server ssh.example.com:22 does not support fsync
```

With sufficiently new versions of libssh and OpenSSH, `fsync` is supported.

1.6.13 NVMe disk images

NVM Express (NVMe) storage controllers can be accessed directly by a userspace driver in QEMU. This bypasses the host kernel file system and block layers while retaining QEMU block layer functionalities, such as block jobs, I/O throttling, image formats, etc. Disk I/O performance is typically higher than with `-drive file=/dev/sda` using either thread pool or linux-aio.

The controller will be exclusively used by the QEMU process once started. To be able to share storage between multiple VMs and other applications on the host, please use the file based protocols.

Before starting QEMU, bind the host NVMe controller to the host vfio-pci driver. For example:

```
# modprobe vfio-pci
# lspci -n -s 0000:06:0d.0
06:0d.0 0401: 1102:0002 (rev 08)
# echo 0000:06:0d.0 > /sys/bus/pci/devices/0000:06:0d.0/driver/unbind
# echo 1102 0002 > /sys/bus/pci/drivers/vfio-pci/new_id

# qemu-system-x86_64 -drive file=nvme://HOST:BUS:SLOT.FUNC/NAMESPACE
```

Alternative syntax using properties:

```
qemu-system-x86_64 -drive file.driver=nvme,file.device=HOST:BUS:SLOT.FUNC,
↪file.namespace=NAMESPACE
```

HOST:*BUS*:*SLOT*.*FUNC* is the NVMe controller's PCI device address on the host.

NAMESPACE is the NVMe namespace number, starting from 1.

1.6.14 Disk image file locking

By default, QEMU tries to protect image files from unexpected concurrent access, as long as it's supported by the block protocol driver and host operating system. If multiple QEMU processes (including QEMU emulators and utilities) try to open the same image with conflicting accessing modes, all but the first one will get an error.

This feature is currently supported by the file protocol on Linux with the Open File Descriptor (OFD) locking API, and can be configured to fall back to POSIX locking if the POSIX host doesn't support Linux OFD locking.

To explicitly enable image locking, specify "locking=on" in the file protocol driver options. If OFD locking is not possible, a warning will be printed and the POSIX locking API will be used. In this case there is a risk that the lock will get silently lost when doing hot plugging and block jobs, due to the shortcomings of the POSIX locking API.

QEMU transparently handles lock handover during shared storage migration. For shared virtual disk images between multiple VMs, the "share-rw" device option should be used.

By default, the guest has exclusive write access to its disk image. If the guest can safely share the disk image with other writers the `-device ...,share-rw=on` parameter can be used. This is only safe if the guest is running software, such as a cluster file system, that coordinates disk accesses to avoid corruption.

Note that share-rw=on only declares the guest's ability to share the disk. Some QEMU features, such as image file formats, require exclusive write access to the disk image and this is unaffected by the share-rw=on option.

Alternatively, locking can be fully disabled by "locking=off" block device option. In the command line, the option is usually in the form of "file.locking=off" as the protocol driver is normally placed as a "file" child under a format driver. For example:

```
-blockdev driver=qcow2,file.filename=/path/to/image,file.locking=off,file.driver=file
```

To check if image locking is active, check the output of the "lslocks" command on host and see if there are locks held by the QEMU process on the image file. More than one byte could be locked by the QEMU instance, each byte of which reflects a particular permission that is acquired or protected by the running block driver.

1.7 Network emulation

QEMU can simulate several network cards (e.g. PCI or ISA cards on the PC target) and can connect them to a network backend on the host or an emulated hub. The various host network backends can either be used to connect the NIC of the guest to a real network (e.g. by using a TAP devices or the non-privileged user mode network stack), or to other guest instances running in another QEMU process (e.g. by using the socket host network backend).

1.7.1 Using TAP network interfaces

This is the standard way to connect QEMU to a real network. QEMU adds a virtual network device on your host (called `tapN`), and you can then configure it as if it was a real ethernet card.

Linux host

As an example, you can download the `linux-test-xxx.tar.gz` archive and copy the script `qemu-ifup` in `/etc` and configure properly `sudo` so that the command `ifconfig` contained in `qemu-ifup` can be executed as root. You must verify that your host kernel supports the TAP network interfaces: the device `/dev/net/tun` must be present.

See *Invocation* to have examples of command lines using the TAP network interfaces.

Windows host

There is a virtual ethernet driver for Windows 2000/XP systems, called TAP-Win32. But it is not included in standard QEMU for Windows, so you will need to get it separately. It is part of OpenVPN package, so download OpenVPN from : https://openvpn.net/.

1.7.2 Using the user mode network stack

By using the option `-net user` (default configuration if no `-net` option is specified), QEMU uses a completely user mode network stack (you don't need root privilege to use the virtual network). The virtual network configuration is the following:

```
guest (10.0.2.15)  <------>  Firewall/DHCP server <-----> Internet
                      |          (10.0.2.2)
                      |
                      ---->  DNS server (10.0.2.3)
                      |
                      ---->  SMB server (10.0.2.4)
```

The QEMU VM behaves as if it was behind a firewall which blocks all incoming connections. You can use a DHCP client to automatically configure the network in the QEMU VM. The DHCP server assign addresses to the hosts starting from 10.0.2.15.

In order to check that the user mode network is working, you can ping the address 10.0.2.2 and verify that you got an address in the range 10.0.2.x from the QEMU virtual DHCP server.

Note that ICMP traffic in general does not work with user mode networking. `ping`, aka. ICMP echo, to the local router (10.0.2.2) shall work, however. If you're using QEMU on Linux >= 3.0, it can use unprivileged ICMP ping sockets to allow `ping` to the Internet. The host admin has to set the ping_group_range in order to grant access to those sockets. To allow ping for GID 100 (usually users group):

```
echo 100 100 > /proc/sys/net/ipv4/ping_group_range
```

When using the built-in TFTP server, the router is also the TFTP server.

When using the `'-netdev user,hostfwd=...'` option, TCP or UDP connections can be redirected from the host to the guest. It allows for example to redirect X11, telnet or SSH connections.

1.7.3 Hubs

QEMU can simulate several hubs. A hub can be thought of as a virtual connection between several network devices. These devices can be for example QEMU virtual ethernet cards or virtual Host ethernet devices (TAP devices). You can connect guest NICs or host network backends to such a hub using the `-netdev hubport` or `-nic hubport` options. The legacy `-net` option also connects the given device to the emulated hub with ID 0 (i.e. the default hub) unless you specify a netdev with `-net nic,netdev=xxx` here.

1.7.4 Connecting emulated networks between QEMU instances

Using the `-netdev socket` (or `-nic socket` or `-net socket`) option, it is possible to create emulated networks that span several QEMU instances. See the description of the `-netdev socket` option in *Invocation* to have a basic example.

1.8 USB emulation

QEMU can emulate a PCI UHCI, OHCI, EHCI or XHCI USB controller. You can plug virtual USB devices or real host USB devices (only works with certain host operating systems). QEMU will automatically create and connect virtual USB hubs as necessary to connect multiple USB devices.

1.8.1 Connecting USB devices

USB devices can be connected with the `-device usb-...` command line option or the `device_add` monitor command. Available devices are:

`usb-mouse` Virtual Mouse. This will override the PS/2 mouse emulation when activated.

`usb-tablet` Pointer device that uses absolute coordinates (like a touchscreen). This means QEMU is able to report the mouse position without having to grab the mouse. Also overrides the PS/2 mouse emulation when activated.

`usb-storage,drive=drive_id` Mass storage device backed by drive_id (see *Disk Images*)

`usb-uas` USB attached SCSI device, see usb-storage.txt for details

`usb-bot` Bulk-only transport storage device, see usb-storage.txt for details here, too

`usb-mtp,rootdir=dir` Media transfer protocol device, using dir as root of the file tree that is presented to the guest.

`usb-host,hostbus=bus,hostaddr=addr` Pass through the host device identified by bus and addr

`usb-host,vendorid=vendor,productid=product` Pass through the host device identified by vendor and product ID

`usb-wacom-tablet` Virtual Wacom PenPartner tablet. This device is similar to the `tablet` above but it can be used with the tslib library because in addition to touch coordinates it reports touch pressure.

`usb-kbd` Standard USB keyboard. Will override the PS/2 keyboard (if present).

`usb-serial,chardev=id` Serial converter. This emulates an FTDI FT232BM chip connected to host character device id.

`usb-braille,chardev=id` Braille device. This will use BrlAPI to display the braille output on a real or fake device referenced by id.

`usb-net[,netdev=id]` Network adapter that supports CDC ethernet and RNDIS protocols. id specifies a netdev defined with `-netdev ...,id=id`. For instance, user-mode networking can be used with

```
qemu-system-x86_64 [...] -netdev user,id=net0 -device usb-net,netdev=net0
```

`usb-ccid` Smartcard reader device

`usb-audio` USB audio device

`u2f-{emulated,passthru}` Universal Second Factor device

1.8.2 Using host USB devices on a Linux host

WARNING: this is an experimental feature. QEMU will slow down when using it. USB devices requiring real time streaming (i.e. USB Video Cameras) are not supported yet.

1. If you use an early Linux 2.4 kernel, verify that no Linux driver is actually using the USB device. A simple way to do that is simply to disable the corresponding kernel module by renaming it from `mydriver.o` to `mydriver.o.disabled`.
2. Verify that `/proc/bus/usb` is working (most Linux distributions should enable it by default). You should see something like that:

```
ls /proc/bus/usb
001  devices  drivers
```

3. Since only root can access to the USB devices directly, you can either launch QEMU as root or change the permissions of the USB devices you want to use. For testing, the following suffices:

```
chown -R myuid /proc/bus/usb
```

4. Launch QEMU and do in the monitor:

```
info usbhost
  Device 1.2, speed 480 Mb/s
    Class 00: USB device 1234:5678, USB DISK
```

You should see the list of the devices you can use (Never try to use hubs, it won't work).

5. Add the device in QEMU by using:

```
device_add usb-host,vendorid=0x1234,productid=0x5678
```

Normally the guest OS should report that a new USB device is plugged. You can use the option `-device usb-host,...` to do the same.

6. Now you can try to use the host USB device in QEMU.

When relaunching QEMU, you may have to unplug and plug again the USB device to make it work again (this is a bug).

1.9 Inter-VM Shared Memory device

On Linux hosts, a shared memory device is available. The basic syntax is:

`qemu_system-x86_64 -device ivshmem-plain,memdev=hostmem`

where hostmem names a host memory backend. For a POSIX shared memory backend, use something like

```
-object memory-backend-file,size=1M,share,mem-path=/dev/shm/ivshmem,id=hostmem
```

If desired, interrupts can be sent between guest VMs accessing the same shared memory region. Interrupt support requires using a shared memory server and using a chardev socket to connect to it. The code for the shared memory server is qemu.git/contrib/ivshmem-server. An example syntax when using the shared memory server is:

```
# First start the ivshmem server once and for all
ivshmem-server -p pidfile -S path -m shm-name -l shm-size -n vectors

# Then start your qemu instances with matching arguments
qemu_system-x86_64 -device ivshmem-doorbell,vectors=vectors,chardev=id
                   -chardev socket,path=path,id=id
```

When using the server, the guest will be assigned a VM ID (>=0) that allows guests using the same server to communicate via interrupts. Guests can read their VM ID from a device register (see ivshmem-spec.txt).

1.9.1 Migration with ivshmem

With device property `master=on`, the guest will copy the shared memory on migration to the destination host. With `master=off`, the guest will not be able to migrate with the device attached. In the latter case, the device should be detached and then reattached after migration using the PCI hotplug support.

At most one of the devices sharing the same memory can be master. The master must complete migration before you plug back the other devices.

1.9.2 ivshmem and hugepages

Instead of specifying the <shm size> using POSIX shm, you may specify a memory backend that has hugepage support:

```
qemu_system-x86_64 -object memory-backend-file,size=1G,mem-path=/dev/
↪hugepages/my-shmem-file,share,id=mb1
                 -device ivshmem-plain,memdev=mb1
```

ivshmem-server also supports hugepages mount points with the `-m` memory path argument.

1.10 Direct Linux Boot

This section explains how to launch a Linux kernel inside QEMU without having to make a full bootable image. It is very useful for fast Linux kernel testing.

The syntax is:

```
qemu-system-x86_64 -kernel bzImage -hda rootdisk.img -append "root=/dev/hda"
```

Use `-kernel` to provide the Linux kernel image and `-append` to give the kernel command line arguments. The `-initrd` option can be used to provide an INITRD image.

If you do not need graphical output, you can disable it and redirect the virtual serial port and the QEMU monitor to the console with the `-nographic` option. The typical command line is:

```
qemu-system-x86_64 -kernel bzImage -hda rootdisk.img                  -append␣
↪"root=/dev/hda console=ttyS0" -nographic
```

Use Ctrl-a c to switch between the serial console and the monitor (see *Keys in the graphical frontends*).

1.11 VNC security

The VNC server capability provides access to the graphical console of the guest VM across the network. This has a number of security considerations depending on the deployment scenarios.

1.11.1 Without passwords

The simplest VNC server setup does not include any form of authentication. For this setup it is recommended to restrict it to listen on a UNIX domain socket only. For example

```
qemu-system-x86_64 [...OPTIONS...] -vnc unix:/home/joebloggs/.qemu-myvm-vnc
```

This ensures that only users on local box with read/write access to that path can access the VNC server. To securely access the VNC server from a remote machine, a combination of netcat+ssh can be used to provide a secure tunnel.

1.11.2 With passwords

The VNC protocol has limited support for password based authentication. Since the protocol limits passwords to 8 characters it should not be considered to provide high security. The password can be fairly easily brute-forced by a client making repeat connections. For this reason, a VNC server using password authentication should be restricted to only listen on the loopback interface or UNIX domain sockets. Password authentication is not supported when operating in FIPS 140-2 compliance mode as it requires the use of the DES cipher. Password authentication is requested with the `password` option, and then once QEMU is running the password is set with the monitor. Until the monitor is used to set the password all clients will be rejected.

```
qemu-system-x86_64 [...OPTIONS...] -vnc :1,password -monitor stdio
(qemu) change vnc password
Password: ****
```

```
(qemu)
```

1.11.3 With x509 certificates

The QEMU VNC server also implements the VeNCrypt extension allowing use of TLS for encryption of the session, and x509 certificates for authentication. The use of x509 certificates is strongly recommended, because TLS on its own is susceptible to man-in-the-middle attacks. Basic x509 certificate support provides a secure session, but no authentication. This allows any client to connect, and provides an encrypted session.

```
qemu-system-x86_64 [...OPTIONS...]   -object tls-creds-x509,id=tls0,dir=/etc/
↪pki/qemu,endpoint=server,verify-peer=no   -vnc :1,tls-creds=tls0 -monitor␣
↪stdio
```

In the above example `/etc/pki/qemu` should contain at least three files, `ca-cert.pem`, `server-cert.pem` and `server-key.pem`. Unprivileged users will want to use a private directory, for example `$HOME/.pki/qemu`. NB the `server-key.pem` file should be protected with file mode 0600 to only be readable by the user owning it.

1.11.4 With x509 certificates and client verification

Certificates can also provide a means to authenticate the client connecting. The server will request that the client provide a certificate, which it will then validate against the CA certificate. This is a good choice if deploying in an environment with a private internal certificate authority. It uses the same syntax as previously, but with `verify-peer` set to `yes` instead.

```
qemu-system-x86_64 [...OPTIONS...]   -object tls-creds-x509,id=tls0,dir=/etc/
↪pki/qemu,endpoint=server,verify-peer=yes   -vnc :1,tls-creds=tls0 -monitor␣
↪stdio
```

1.11.5 With x509 certificates, client verification and passwords

Finally, the previous method can be combined with VNC password authentication to provide two layers of authentication for clients.

```
qemu-system-x86_64 [...OPTIONS...]   -object tls-creds-x509,id=tls0,dir=/etc/
↪pki/qemu,endpoint=server,verify-peer=yes   -vnc :1,tls-creds=tls0,password -
↪monitor stdio
(qemu) change vnc password
Password: ****
(qemu)
```

1.11.6 With SASL authentication

The SASL authentication method is a VNC extension, that provides an easily extendable, pluggable authentication method. This allows for integration with a wide range of authentication mechanisms, such as PAM, GSSAPI/Kerberos, LDAP, SQL databases, one-time keys and more. The strength of the authentication depends on the exact mechanism configured. If the chosen mechanism also provides a SSF layer, then it will encrypt the datastream as well.

Refer to the later docs on how to choose the exact SASL mechanism used for authentication, but assuming use of one supporting SSF, then QEMU can be launched with:

```
qemu-system-x86_64 [...OPTIONS...] -vnc :1,sasl -monitor stdio
```

1.11.7 With x509 certificates and SASL authentication

If the desired SASL authentication mechanism does not supported SSF layers, then it is strongly advised to run it in combination with TLS and x509 certificates. This provides securely encrypted data stream, avoiding risk of compromising of the security credentials. This can be enabled, by combining the 'sasl' option with the aforementioned TLS + x509 options:

```
qemu-system-x86_64 [...OPTIONS...]   -object tls-creds-x509,id=tls0,dir=/etc/
↪pki/qemu,endpoint=server,verify-peer=yes   -vnc :1,tls-creds=tls0,sasl -
↪monitor stdio
```

1.11.8 Configuring SASL mechanisms

The following documentation assumes use of the Cyrus SASL implementation on a Linux host, but the principles should apply to any other SASL implementation or host. When SASL is enabled, the mechanism configuration will be loaded from system default SASL service config /etc/sasl2/qemu.conf. If running QEMU as an unprivileged user, an environment variable SASL_CONF_PATH can be used to make it search alternate locations for the service config file.

If the TLS option is enabled for VNC, then it will provide session encryption, otherwise the SASL mechanism will have to provide encryption. In the latter case the list of possible plugins that can be used is drastically reduced. In fact only the GSSAPI SASL mechanism provides an acceptable level of security by modern standards. Previous versions of QEMU referred to the DIGEST-MD5 mechanism, however, it has multiple serious flaws described in detail in RFC 6331 and thus should never be used any more. The SCRAM-SHA-1 mechanism provides a simple username/password auth facility similar to DIGEST-MD5, but does not support session encryption, so can only be used in combination with TLS.

When not using TLS the recommended configuration is

```
mech_list: gssapi
keytab: /etc/qemu/krb5.tab
```

This says to use the 'GSSAPI' mechanism with the Kerberos v5 protocol, with the server principal stored in /etc/qemu/krb5.tab. For this to work the administrator of your KDC must generate a Kerberos principal for the server, with a name of 'qemu/somehost.example.com@EXAMPLE.COM' replacing 'somehost.example.com' with the fully qualified host name of the machine running QEMU, and 'EXAMPLE.COM' with the Kerberos Realm.

When using TLS, if username+password authentication is desired, then a reasonable configuration is

```
mech_list: scram-sha-1
sasldb_path: /etc/qemu/passwd.db
```

The `saslpasswd2` program can be used to populate the `passwd.db` file with accounts.

Other SASL configurations will be left as an exercise for the reader. Note that all mechanisms, except GSSAPI, should be combined with use of TLS to ensure a secure data channel.

1.12 TLS setup for network services

Almost all network services in QEMU have the ability to use TLS for session data encryption, along with x509 certificates for simple client authentication. What follows is a description of how to generate certificates suitable for usage with QEMU, and applies to the VNC server, character devices with the TCP backend, NBD server and client, and migration server and client.

At a high level, QEMU requires certificates and private keys to be provided in PEM format. Aside from the core fields, the certificates should include various extension data sets, including v3 basic constraints data, key purpose, key usage and subject alt name.

The GnuTLS package includes a command called `certtool` which can be used to easily generate certificates and keys in the required format with expected data present. Alternatively a certificate management service may be used.

At a minimum it is necessary to setup a certificate authority, and issue certificates to each server. If using x509 certificates for authentication, then each client will also need to be issued a certificate.

Assuming that the QEMU network services will only ever be exposed to clients on a private intranet, there is no need to use a commercial certificate authority to create certificates. A self-signed CA is sufficient, and in fact likely to be more secure since it removes the ability of malicious 3rd parties to trick the CA into mis-issuing certs for impersonating your services. The only likely exception where a commercial CA might be desirable is if enabling the VNC websockets server and exposing it directly to remote browser clients. In such a case it might be useful to use a commercial CA to avoid needing to install custom CA certs in the web browsers.

The recommendation is for the server to keep its certificates in either `/etc/pki/qemu` or for unprivileged users in `$HOME/.pki/qemu`.

1.12.1 Setup the Certificate Authority

This step only needs to be performed once per organization / organizational unit. First the CA needs a private key. This key must be kept VERY secret and secure. If this key is compromised the entire trust chain of the certificates issued with it is lost.

```
# certtool --generate-privkey > ca-key.pem
```

To generate a self-signed certificate requires one core piece of information, the name of the organization. A template file `ca.info` should be populated with the desired data to avoid having to deal with interactive prompts from certtool:

```
# cat > ca.info <<EOF
cn = Name of your organization
ca
cert_signing_key
EOF
# certtool --generate-self-signed \
           --load-privkey ca-key.pem
           --template ca.info \
           --outfile ca-cert.pem
```

The `ca` keyword in the template sets the v3 basic constraints extension to indicate this certificate is for a CA, while `cert_signing_key` sets the key usage extension to indicate this will be used for signing other keys. The generated `ca-cert.pem` file should be copied to all servers and clients wishing to utilize TLS support in the VNC server. The `ca-key.pem` must not be disclosed/copied anywhere except the host responsible for issuing certificates.

1.12.2 Issuing server certificates

Each server (or host) needs to be issued with a key and certificate. When connecting the certificate is sent to the client which validates it against the CA certificate. The core pieces of information for a server certificate are the hostnames and/or IP addresses that will be used by clients when connecting. The hostname / IP address that the client specifies when connecting will be validated against the hostname(s) and IP address(es) recorded in the server certificate, and if no match is found the client will close the connection.

Thus it is recommended that the server certificate include both the fully qualified and unqualified hostnames. If the server will have permanently assigned IP address(es), and clients are likely to use them when connecting, they may also be included in the certificate. Both IPv4 and IPv6 addresses are supported. Historically certificates only included 1 hostname in the CN field, however, usage of this field for validation is now deprecated. Instead modern TLS clients will validate against the Subject Alt Name extension data, which allows for multiple entries. In the future usage of the CN field may be discontinued entirely, so providing SAN extension data is strongly recommended.

On the host holding the CA, create template files containing the information for each server, and use it to issue server certificates.

```
# cat > server-hostNNN.info <<EOF
organization = Name  of your organization
cn = hostNNN.foo.example.com
dns_name = hostNNN
dns_name = hostNNN.foo.example.com
ip_address = 10.0.1.87
ip_address = 192.8.0.92
ip_address = 2620:0:cafe::87
ip_address = 2001:24::92
tls_www_server
encryption_key
signing_key
EOF
# certtool --generate-privkey > server-hostNNN-key.pem
# certtool --generate-certificate \
           --load-ca-certificate ca-cert.pem \
           --load-ca-privkey ca-key.pem \
           --load-privkey server-hostNNN-key.pem \
           --template server-hostNNN.info \
           --outfile server-hostNNN-cert.pem
```

The `dns_name` and `ip_address` fields in the template are setting the subject alt name extension data. The `tls_www_server` keyword is the key purpose extension to indicate this certificate is intended for usage in a web server. Although QEMU network services are not in fact HTTP servers (except for VNC websockets), setting this key purpose is still recommended. The `encryption_key` and `signing_key` keyword is the key usage extension to indicate this certificate is intended for usage in the data session.

The `server-hostNNN-key.pem` and `server-hostNNN-cert.pem` files should now be securely copied to the server for which they were generated, and renamed to `server-key.pem` and `server-cert.pem` when added to the `/etc/pki/qemu` directory on the target host. The `server-key.pem` file is security sensitive and should be kept protected with file mode 0600 to prevent disclosure.

1.12.3 Issuing client certificates

The QEMU x509 TLS credential setup defaults to enabling client verification using certificates, providing a simple authentication mechanism. If this default is used, each client also needs to be issued a certificate. The client certificate contains enough metadata to uniquely identify the client with the scope of the certificate authority. The client certificate would typically include fields for organization, state, city, building, etc.

Once again on the host holding the CA, create template files containing the information for each client, and use it to issue client certificates.

```
# cat > client-hostNNN.info <<EOF
country = GB
state = London
locality = City Of London
organization = Name of your organization
cn = hostNNN.foo.example.com
```

(continues on next page)

(continued from previous page)

```
tls_www_client
encryption_key
signing_key
EOF
# certtool --generate-privkey > client-hostNNN-key.pem
# certtool --generate-certificate \
           --load-ca-certificate ca-cert.pem \
           --load-ca-privkey ca-key.pem \
           --load-privkey client-hostNNN-key.pem \
           --template client-hostNNN.info \
           --outfile client-hostNNN-cert.pem
```

The subject alt name extension data is not required for clients, so the the `dns_name` and `ip_address` fields are not included. The `tls_www_client` keyword is the key purpose extension to indicate this certificate is intended for usage in a web client. Although QEMU network clients are not in fact HTTP clients, setting this key purpose is still recommended. The `encryption_key` and `signing_key` keyword is the key usage extension to indicate this certificate is intended for usage in the data session.

The `client-hostNNN-key.pem` and `client-hostNNN-cert.pem` files should now be securely copied to the client for which they were generated, and renamed to `client-key.pem` and `client-cert.pem` when added to the `/etc/pki/qemu` directory on the target host. The `client-key.pem` file is security sensitive and should be kept protected with file mode 0600 to prevent disclosure.

If a single host is going to be using TLS in both a client and server role, it is possible to create a single certificate to cover both roles. This would be quite common for the migration and NBD services, where a QEMU process will be started by accepting a TLS protected incoming migration, and later itself be migrated out to another host. To generate a single certificate, simply include the template data from both the client and server instructions in one.

```
# cat > both-hostNNN.info <<EOF
country = GB
state = London
locality = City Of London
organization = Name of your organization
cn = hostNNN.foo.example.com
dns_name = hostNNN
dns_name = hostNNN.foo.example.com
ip_address = 10.0.1.87
ip_address = 192.8.0.92
ip_address = 2620:0:cafe::87
ip_address = 2001:24::92
tls_www_server
tls_www_client
encryption_key
signing_key
EOF
# certtool --generate-privkey > both-hostNNN-key.pem
# certtool --generate-certificate \
           --load-ca-certificate ca-cert.pem \
           --load-ca-privkey ca-key.pem \
           --load-privkey both-hostNNN-key.pem \
           --template both-hostNNN.info \
           --outfile both-hostNNN-cert.pem
```

When copying the PEM files to the target host, save them twice, once as `server-cert.pem` and `server-key.pem`, and again as `client-cert.pem` and `client-key.pem`.

1.12.4 TLS x509 credential configuration

QEMU has a standard mechanism for loading x509 credentials that will be used for network services and clients. It requires specifying the `tls-creds-x509` class name to the `--object` command line argument for the system emulators. Each set of credentials loaded should be given a unique string identifier via the `id` parameter. A single set of TLS credentials can be used for multiple network backends, so VNC, migration, NBD, character devices can all share the same credentials. Note, however, that credentials for use in a client endpoint must be loaded separately from those used in a server endpoint.

When specifying the object, the `dir` parameters specifies which directory contains the credential files. This directory is expected to contain files with the names mentioned previously, `ca-cert.pem`, `server-key.pem`, `server-cert.pem`, `client-key.pem` and `client-cert.pem` as appropriate. It is also possible to include a set of pre-generated Diffie-Hellman (DH) parameters in a file `dh-params.pem`, which can be created using the `certtool --generate-dh-params` command. If omitted, QEMU will dynamically generate DH parameters when loading the credentials.

The `endpoint` parameter indicates whether the credentials will be used for a network client or server, and determines which PEM files are loaded.

The `verify` parameter determines whether x509 certificate validation should be performed. This defaults to enabled, meaning clients will always validate the server hostname against the certificate subject alt name fields and/or CN field. It also means that servers will request that clients provide a certificate and validate them. Verification should never be turned off for client endpoints, however, it may be turned off for server endpoints if an alternative mechanism is used to authenticate clients. For example, the VNC server can use SASL to authenticate clients instead.

To load server credentials with client certificate validation enabled

```
qemu-system-x86_64 -object tls-creds-x509,id=tls0,dir=/etc/pki/qemu,
↪endpoint=server
```

while to load client credentials use

```
qemu-system-x86_64 -object tls-creds-x509,id=tls0,dir=/etc/pki/qemu,
↪endpoint=client
```

Network services which support TLS will all have a `tls-creds` parameter which expects the ID of the TLS credentials object. For example with VNC:

```
qemu-system-x86_64 -vnc 0.0.0.0:0,tls-creds=tls0
```

1.12.5 TLS Pre-Shared Keys (PSK)

Instead of using certificates, you may also use TLS Pre-Shared Keys (TLS-PSK). This can be simpler to set up than certificates but is less scalable.

Use the GnuTLS `psktool` program to generate a `keys.psk` file containing one or more usernames and random keys:

```
mkdir -m 0700 /tmp/keys
psktool -u rich -p /tmp/keys/keys.psk
```

TLS-enabled servers such as qemu-nbd can use this directory like so:

```
qemu-nbd \
  -t -x / \
  --object tls-creds-psk,id=tls0,endpoint=server,dir=/tmp/keys \
  --tls-creds tls0 \
  image.qcow2
```

When connecting from a qemu-based client you must specify the directory containing `keys.psk` and an optional username (defaults to "qemu"):

```
qemu-img info \
  --object tls-creds-psk,id=tls0,dir=/tmp/keys,username=rich,endpoint=client \
  --image-opts \
  file.driver=nbd,file.host=localhost,file.port=10809,file.tls-creds=tls0,file.
↪export=/
```

1.13 GDB usage

QEMU supports working with gdb via gdb's remote-connection facility (the "gdbstub"). This allows you to debug guest code in the same way that you might with a low-level debug facility like JTAG on real hardware. You can stop and start the virtual machine, examine state like registers and memory, and set breakpoints and watchpoints.

In order to use gdb, launch QEMU with the `-s` and `-S` options. The `-s` option will make QEMU listen for an incoming connection from gdb on TCP port 1234, and `-S` will make QEMU not start the guest until you tell it to from gdb. (If you want to specify which TCP port to use or to use something other than TCP for the gdbstub connection, use the `-gdb dev` option instead of `-s`.)

```
qemu-system-x86_64 -s -S -kernel bzImage -hda rootdisk.img -append "root=/dev/
↪hda"
```

QEMU will launch but will silently wait for gdb to connect.

Then launch gdb on the 'vmlinux' executable:

```
> gdb vmlinux
```

In gdb, connect to QEMU:

```
(gdb) target remote localhost:1234
```

Then you can use gdb normally. For example, type 'c' to launch the kernel:

```
(gdb) c
```

Here are some useful tips in order to use gdb on system code:

1. Use `info reg` to display all the CPU registers.
2. Use `x/10i $eip` to display the code at the PC position.
3. Use `set architecture i8086` to dump 16 bit code. Then use `x/10i $cs*16+$eip` to dump the code at the PC position.

Advanced debugging options:

The default single stepping behavior is step with the IRQs and timer service routines off. It is set this way because when gdb executes a single step it expects to advance beyond the current instruction. With the IRQs and timer service routines on, a single step might jump into the one of the interrupt or exception vectors instead of executing the current instruction. This means you may hit the same breakpoint a number of times before executing the instruction gdb wants to have executed. Because there are rare circumstances where you want to single step into an interrupt vector the behavior can be controlled from GDB. There are three commands you can query and set the single step behavior:

`maintenance packet qqemu.sstepbits` This will display the MASK bits used to control the single stepping IE:

```
(gdb) maintenance packet qqemu.sstepbits
sending: "qqemu.sstepbits"
received: "ENABLE=1,NOIRQ=2,NOTIMER=4"
```

maintenance packet qqemu.sstep This will display the current value of the mask used when single stepping IE:

```
(gdb) maintenance packet qqemu.sstep
sending: "qqemu.sstep"
received: "0x7"
```

maintenance packet Qqemu.sstep=HEX_VALUE This will change the single step mask, so if wanted to enable IRQs on the single step, but not timers, you would use:

```
(gdb) maintenance packet Qqemu.sstep=0x5
sending: "qemu.sstep=0x5"
received: "OK"
```

Another feature that QEMU gdbstub provides is to toggle the memory GDB works with, by default GDB will show the current process memory respecting the virtual address translation.

If you want to examine/change the physical memory you can set the gdbstub to work with the physical memory rather with the virtual one.

The memory mode can be checked by sending the following command:

maintenance packet qqemu.PhyMemMode This will return either 0 or 1, 1 indicates you are currently in the physical memory mode.

maintenance packet Qqemu.PhyMemMode:1 This will change the memory mode to physical memory.

maintenance packet Qqemu.PhyMemMode:0 This will change it back to normal memory mode.

1.14 Managed start up options

In system mode emulation, it's possible to create a VM in a paused state using the `-S` command line option. In this state the machine is completely initialized according to command line options and ready to execute VM code but VCPU threads are not executing any code. The VM state in this paused state depends on the way QEMU was started. It could be in:

- initial state (after reset/power on state)
- with direct kernel loading, the initial state could be amended to execute code loaded by QEMU in the VM's RAM and with incoming migration
- with incoming migration, initial state will be amended with the migrated machine state after migration completes

This paused state is typically used by users to query machine state and/or additionally configure the machine (by hotplugging devices) in runtime before allowing VM code to run.

However, at the `-S` pause point, it's impossible to configure options that affect initial VM creation (like: `-smp`/`-m`/`-numa` ...) or cold plug devices. The experimental `--preconfig` command line option allows pausing QEMU before the initial VM creation, in a "preconfig" state, where additional queries and configuration can be performed via QMP before moving on to the resulting configuration startup. In the preconfig state, QEMU only allows a limited set of commands over the QMP monitor, where the commands do not depend on an initialized machine, including but not limited to:

- `qmp_capabilities`

- `query-qmp-schema`
- `query-commands`
- `query-status`
- `x-exit-preconfig`

1.15 QEMU System Emulator Targets

QEMU is a generic emulator and it emulates many machines. Most of the options are similar for all machines. Specific information about the various targets are mentioned in the following sections.

Contents:

1.15.1 x86 (PC) System emulator

Peripherals

The QEMU PC System emulator simulates the following peripherals:

- i440FX host PCI bridge and PIIX3 PCI to ISA bridge
- Cirrus CLGD 5446 PCI VGA card or dummy VGA card with Bochs VESA extensions (hardware level, including all non standard modes).
- PS/2 mouse and keyboard
- 2 PCI IDE interfaces with hard disk and CD-ROM support
- Floppy disk
- PCI and ISA network adapters
- Serial ports
- IPMI BMC, either and internal or external one
- Creative SoundBlaster 16 sound card
- ENSONIQ AudioPCI ES1370 sound card
- Intel 82801AA AC97 Audio compatible sound card
- Intel HD Audio Controller and HDA codec
- Adlib (OPL2) - Yamaha YM3812 compatible chip
- Gravis Ultrasound GF1 sound card
- CS4231A compatible sound card
- PC speaker
- PCI UHCI, OHCI, EHCI or XHCI USB controller and a virtual USB-1.1 hub.

SMP is supported with up to 255 CPUs.

QEMU uses the PC BIOS from the Seabios project and the Plex86/Bochs LGPL VGA BIOS.

QEMU uses YM3812 emulation by Tatsuyuki Satoh.

QEMU uses GUS emulation (GUSEMU32 http://www.deinmeister.de/gusemu/) by Tibor “TS” Schütz.

Note that, by default, GUS shares IRQ(7) with parallel ports and so QEMU must be told to not have parallel ports to have working GUS.

```
qemu_system-x86_64 dos.img -device gus -parallel none
```

Alternatively:

```
qemu_system-x86_64 dos.img -device gus,irq=5
```

Or some other unclaimed IRQ.

CS4231A is the chip used in Windows Sound System and GUSMAX products

The PC speaker audio device can be configured using the pcspk-audiodev machine property, i.e.

```
qemu_system-x86_64 some.img -audiodev <backend>,id=<name> -machine pcspk-
↪audiodev=<name>
```

Recommendations for KVM CPU model configuration on x86 hosts

The information that follows provides recommendations for configuring CPU models on x86 hosts. The goals are to maximise performance, while protecting guest OS against various CPU hardware flaws, and optionally enabling live migration between hosts with heterogeneous CPU models.

Two ways to configure CPU models with QEMU / KVM

(1) **Host passthrough**

This passes the host CPU model features, model, stepping, exactly to the guest. Note that KVM may filter out some host CPU model features if they cannot be supported with virtualization. Live migration is unsafe when this mode is used as libvirt / QEMU cannot guarantee a stable CPU is exposed to the guest across hosts. This is the recommended CPU to use, provided live migration is not required.

(2) **Named model**

QEMU comes with a number of predefined named CPU models, that typically refer to specific generations of hardware released by Intel and AMD. These allow the guest VMs to have a degree of isolation from the host CPU, allowing greater flexibility in live migrating between hosts with differing hardware. @end table

In both cases, it is possible to optionally add or remove individual CPU features, to alter what is presented to the guest by default.

Libvirt supports a third way to configure CPU models known as “Host model”. This uses the QEMU “Named model” feature, automatically picking a CPU model that is similar the host CPU, and then adding extra features to approximate the host model as closely as possible. This does not guarantee the CPU family, stepping, etc will precisely match the host CPU, as they would with “Host passthrough”, but gives much of the benefit of passthrough, while making live migration safe.

Preferred CPU models for Intel x86 hosts

The following CPU models are preferred for use on Intel hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

`Cascadelake-Server`, `Cascadelake-Server-noTSX` Intel Xeon Processor (Cascade Lake, 2019), with “stepping” levels 6 or 7 only. (The Cascade Lake Xeon processor with *stepping 5 is vulnerable to MDS variants*.)

`Skylake-Server`, `Skylake-Server-IBRS`, `Skylake-Server-IBRS-noTSX` Intel Xeon Processor (Skylake, 2016)

`Skylake-Client`, `Skylake-Client-IBRS`, `Skylake-Client-noTSX-IBRS}` Intel Core Processor (Skylake, 2015)

`Broadwell`, `Broadwell-IBRS`, `Broadwell-noTSX`, `Broadwell-noTSX-IBRS` Intel Core Processor (Broadwell, 2014)

`Haswell`, `Haswell-IBRS`, `Haswell-noTSX`, `Haswell-noTSX-IBRS` Intel Core Processor (Haswell, 2013)

`IvyBridge`, `IvyBridge-IBR` Intel Xeon E3-12xx v2 (Ivy Bridge, 2012)

`SandyBridge`, `SandyBridge-IBRS` Intel Xeon E312xx (Sandy Bridge, 2011)

`Westmere`, `Westmere-IBRS` Westmere E56xx/L56xx/X56xx (Nehalem-C, 2010)

`Nehalem`, `Nehalem-IBRS` Intel Core i7 9xx (Nehalem Class Core i7, 2008)

`Penryn` Intel Core 2 Duo P9xxx (Penryn Class Core 2, 2007)

`Conroe` Intel Celeron_4x0 (Conroe/Merom Class Core 2, 2006)

Important CPU features for Intel x86 hosts

The following are important CPU features that should be used on Intel x86 hosts, when available in the host CPU. Some of them require explicit configuration to enable, as they are not included by default in some, or all, of the named CPU models listed above. In general all of these features are included if using "Host passthrough" or "Host model".

`pcid` Recommended to mitigate the cost of the Meltdown (CVE-2017-5754) fix.

Included by default in Haswell, Broadwell & Skylake Intel CPU models.

Should be explicitly turned on for Westmere, SandyBridge, and IvyBridge Intel CPU models. Note that some desktop/mobile Westmere CPUs cannot support this feature.

`spec-ctrl` Required to enable the Spectre v2 (CVE-2017-5715) fix.

Included by default in Intel CPU models with -IBRS suffix.

Must be explicitly turned on for Intel CPU models without -IBRS suffix.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

`stibp` Required to enable stronger Spectre v2 (CVE-2017-5715) fixes in some operating systems.

Must be explicitly turned on for all Intel CPU models.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

`ssbd` Required to enable the CVE-2018-3639 fix.

Not included by default in any Intel CPU model.

Must be explicitly turned on for all Intel CPU models.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

`pdpe1gb` Recommended to allow guest OS to use 1GB size pages.

Not included by default in any Intel CPU model.

Should be explicitly turned on for all Intel CPU models.

Note that not all CPU hardware will support this feature.

`md-clear` Required to confirm the MDS (CVE-2018-12126, CVE-2018-12127, CVE-2018-12130, CVE-2019-11091) fixes.

Not included by default in any Intel CPU model.

Must be explicitly turned on for all Intel CPU models.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

`mds-no` Recommended to inform the guest OS that the host is *not* vulnerable to any of the MDS variants ([MFBDS] CVE-2018-12130, [MLPDS] CVE-2018-12127, [MSBDS] CVE-2018-12126).

This is an MSR (Model-Specific Register) feature rather than a CPUID feature, so it will not appear in the Linux `/proc/cpuinfo` in the host or guest. Instead, the host kernel uses it to populate the MDS vulnerability file in `sysfs`.

So it should only be enabled for VMs if the host reports @code{Not affected} in the `/sys/devices/system/cpu/vulnerabilities/mds` file.

`taa-no` Recommended to inform that the guest that the host is `not` vulnerable to CVE-2019-11135, TSX Asynchronous Abort (TAA).

This too is an MSR feature, so it does not show up in the Linux `/proc/cpuinfo` in the host or guest.

It should only be enabled for VMs if the host reports `Not affected` in the `/sys/devices/system/cpu/vulnerabilities/tsx_async_abort` file.

`tsx-ctrl` Recommended to inform the guest that it can disable the Intel TSX (Transactional Synchronization Extensions) feature; or, if the processor is vulnerable, use the Intel VERW instruction (a processor-level instruction that performs checks on memory access) as a mitigation for the TAA vulnerability. (For details, refer to Intel's deep dive into MDS.)

Expose this to the guest OS if and only if: (a) the host has TSX enabled; *and* (b) the guest has `rtm` CPU flag enabled.

By disabling TSX, KVM-based guests can avoid paying the price of mitigating TSX-based attacks.

Note that `tsx-ctrl` too is an MSR feature, so it does not show up in the Linux `/proc/cpuinfo` in the host or guest.

To validate that Intel TSX is indeed disabled for the guest, there are two ways: (a) check for the *absence* of `rtm` in the guest's `/proc/cpuinfo`; or (b) the `/sys/devices/system/cpu/vulnerabilities/tsx_async_abort` file in the guest should report `Mitigation:  TSX disabled`.

Preferred CPU models for AMD x86 hosts

The following CPU models are preferred for use on Intel hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

`EPYC`, `EPYC-IBPB` AMD EPYC Processor (2017)

`Opteron_G5` AMD Opteron 63xx class CPU (2012)

`Opteron_G4` AMD Opteron 62xx class CPU (2011)

`Opteron_G3` AMD Opteron 23xx (Gen 3 Class Opteron, 2009)

`Opteron_G2` AMD Opteron 22xx (Gen 2 Class Opteron, 2006)

`Opteron_G1` AMD Opteron 240 (Gen 1 Class Opteron, 2004)

Important CPU features for AMD x86 hosts

The following are important CPU features that should be used on AMD x86 hosts, when available in the host CPU. Some of them require explicit configuration to enable, as they are not included by default in some, or all, of the named CPU models listed above. In general all of these features are included if using "Host passthrough" or "Host model".

`ibpb` Required to enable the Spectre v2 (CVE-2017-5715) fix.

Included by default in AMD CPU models with -IBPB suffix.

Must be explicitly turned on for AMD CPU models without -IBPB suffix.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

`stibp` Required to enable stronger Spectre v2 (CVE-2017-5715) fixes in some operating systems.

Must be explicitly turned on for all AMD CPU models.

Requires the host CPU microcode to support this feature before it can be used for guest CPUs.

`virt-ssbd` Required to enable the CVE-2018-3639 fix

Not included by default in any AMD CPU model.

Must be explicitly turned on for all AMD CPU models.

This should be provided to guests, even if amd-ssbd is also provided, for maximum guest compatibility.

Note for some QEMU / libvirt versions, this must be force enabled when when using "Host model", because this is a virtual feature that doesn't exist in the physical host CPUs.

`amd-ssbd` Required to enable the CVE-2018-3639 fix

Not included by default in any AMD CPU model.

Must be explicitly turned on for all AMD CPU models.

This provides higher performance than `virt-ssbd` so should be exposed to guests whenever available in the host. `virt-ssbd` should none the less also be exposed for maximum guest compatibility as some kernels only know about `virt-ssbd`.

`amd-no-ssb` Recommended to indicate the host is not vulnerable CVE-2018-3639

Not included by default in any AMD CPU model.

Future hardware generations of CPU will not be vulnerable to CVE-2018-3639, and thus the guest should be told not to enable its mitigations, by exposing amd-no-ssb. This is mutually exclusive with virt-ssbd and amd-ssbd.

`pdpe1gb` Recommended to allow guest OS to use 1GB size pages

Not included by default in any AMD CPU model.

Should be explicitly turned on for all AMD CPU models.

Note that not all CPU hardware will support this feature.

Default x86 CPU models

The default QEMU CPU models are designed such that they can run on all hosts. If an application does not wish to do perform any host compatibility checks before launching guests, the default is guaranteed to work.

The default CPU models will, however, leave the guest OS vulnerable to various CPU hardware flaws, so their use is strongly discouraged. Applications should follow the earlier guidance to setup a better CPU configuration, with host passthrough recommended if live migration is not needed.

qemu32, qemu64 QEMU Virtual CPU version 2.5+ (32 & 64 bit variants)

`qemu64` is used for x86_64 guests and `qemu32` is used for i686 guests, when no `-cpu` argument is given to QEMU, or no `<cpu>` is provided in libvirt XML.

Other non-recommended x86 CPUs

The following CPUs models are compatible with most AMD and Intel x86 hosts, but their usage is discouraged, as they expose a very limited featureset, which prevents guests having optimal performance.

kvm32, kvm64 Common KVM processor (32 & 64 bit variants).

Legacy models just for historical compatibility with ancient QEMU versions.

486, athlon, phenom, coreduo, core2duo, n270, pentium, pentium2, pentium3 Various very old x86 CPU models, mostly predating the introduction of hardware assisted virtualization, that should thus not be required for running virtual machines.

Syntax for configuring CPU models

The examples below illustrate the approach to configuring the various CPU models / features in QEMU and libvirt.

QEMU command line

Host passthrough:

```
qemu-system-x86_64 -cpu host
```

Host passthrough with feature customization:

```
qemu-system-x86_64 -cpu host,-vmx,...
```

Named CPU models:

```
qemu-system-x86_64 -cpu Westmere
```

Named CPU models with feature customization:

```
qemu-system-x86_64 -cpu Westmere,+pcid,...
```

Libvirt guest XML

Host passthrough:

```
<cpu mode='host-passthrough'/>
```

Host passthrough with feature customization:

```
<cpu mode='host-passthrough'>
    <feature name="vmx" policy="disable"/>
    ...
</cpu>
```

Host model:

```
<cpu mode='host-model'/>
```

Host model with feature customization:

```
<cpu mode='host-model'>
    <feature name="vmx" policy="disable"/>
    ...
</cpu>
```

Named model:

```
<cpu mode='custom'>
    <model name="Westmere"/>
</cpu>
```

Named model with feature customization:

```
<cpu mode='custom'>
    <model name="Westmere"/>
    <feature name="pcid" policy="require"/>
    ...
</cpu>
```

OS requirements

On x86_64 hosts, the default set of CPU features enabled by the KVM accelerator require the host to be running Linux v4.5 or newer. Red Hat Enterprise Linux 7 is also supported, since the required functionality was backported.

1.15.2 PowerPC System emulator

Use the executable `qemu-system-ppc` to simulate a complete 40P (PREP) or PowerMac PowerPC system.

QEMU emulates the following PowerMac peripherals:

- UniNorth or Grackle PCI Bridge
- PCI VGA compatible card with VESA Bochs Extensions
- 2 PMAC IDE interfaces with hard disk and CD-ROM support
- NE2000 PCI adapters
- Non Volatile RAM
- VIA-CUDA with ADB keyboard and mouse.

QEMU emulates the following 40P (PREP) peripherals:

- PCI Bridge
- PCI VGA compatible card with VESA Bochs Extensions
- 2 IDE interfaces with hard disk and CD-ROM support
- Floppy disk
- PCnet network adapters
- Serial port
- PREP Non Volatile RAM
- PC compatible keyboard and mouse.

Since version 0.9.1, QEMU uses OpenBIOS https://www.openbios.org/ for the g3beige and mac99 PowerMac and the 40p machines. OpenBIOS is a free (GPL v2) portable firmware implementation. The goal is to implement a 100% IEEE 1275-1994 (referred to as Open Firmware) compliant firmware.

More information is available at http://perso.magic.fr/l_indien/qemu-ppc/.

1.15.3 Sparc32 System emulator

Use the executable `qemu-system-sparc` to simulate the following Sun4m architecture machines:

- SPARCstation 4
- SPARCstation 5
- SPARCstation 10
- SPARCstation 20
- SPARCserver 600MP
- SPARCstation LX
- SPARCstation Voyager
- SPARCclassic
- SPARCbook

The emulation is somewhat complete. SMP up to 16 CPUs is supported, but Linux limits the number of usable CPUs to 4.

QEMU emulates the following sun4m peripherals:

- IOMMU
- TCX or cgthree Frame buffer
- Lance (Am7990) Ethernet
- Non Volatile RAM M48T02/M48T08
- Slave I/O: timers, interrupt controllers, Zilog serial ports, keyboard and power/reset logic
- ESP SCSI controller with hard disk and CD-ROM support
- Floppy drive (not on SS-600MP)
- CS4231 sound device (only on SS-5, not working yet)

The number of peripherals is fixed in the architecture. Maximum memory size depends on the machine type, for SS-5 it is 256MB and for others 2047MB.

Since version 0.8.2, QEMU uses OpenBIOS https://www.openbios.org/. OpenBIOS is a free (GPL v2) portable firmware implementation. The goal is to implement a 100% IEEE 1275-1994 (referred to as Open Firmware) compliant firmware.

A sample Linux 2.6 series kernel and ram disk image are available on the QEMU web site. There are still issues with NetBSD and OpenBSD, but most kernel versions work. Please note that currently older Solaris kernels don't work probably due to interface issues between OpenBIOS and Solaris.

1.15.4 Sparc64 System emulator

Use the executable `qemu-system-sparc64` to simulate a Sun4u (UltraSPARC PC-like machine), Sun4v (T1 PC-like machine), or generic Niagara (T1) machine. The Sun4u emulator is mostly complete, being able to run Linux, NetBSD and OpenBSD in headless (-nographic) mode. The Sun4v emulator is still a work in progress.

The Niagara T1 emulator makes use of firmware and OS binaries supplied in the S10image/ directory of the OpenSPARC T1 project http://download.oracle.com/technetwork/systems/opensparc/OpenSPARCT1_Arch.1.5.tar.bz2 and is able to boot the disk.s10hw2 Solaris image.

```
qemu-system-sparc64 -M niagara -L /path-to/S10image/ \
                    -nographic -m 256 \
                    -drive if=pflash,readonly=on,file=/S10image/disk.s10hw2
```

QEMU emulates the following peripherals:

- UltraSparc IIi APB PCI Bridge
- PCI VGA compatible card with VESA Bochs Extensions
- PS/2 mouse and keyboard
- Non Volatile RAM M48T59
- PC-compatible serial ports
- 2 PCI IDE interfaces with hard disk and CD-ROM support
- Floppy disk

1.15.5 MIPS System emulator

Four executables cover simulation of 32 and 64-bit MIPS systems in both endian options, `qemu-system-mips`, `qemu-system-mipsel` `qemu-system-mips64` and `qemu-system-mips64el`. Five different machine types are emulated:

- A generic ISA PC-like machine “mips”
- The MIPS Malta prototype board “malta”
- An ACER Pica “pica61”. This machine needs the 64-bit emulator.
- MIPS emulator pseudo board “mipssim”
- A MIPS Magnum R4000 machine “magnum”. This machine needs the 64-bit emulator.

The generic emulation is supported by Debian ‘Etch’ and is able to install Debian into a virtual disk image. The following devices are emulated:

- A range of MIPS CPUs, default is the 24Kf
- PC style serial port
- PC style IDE disk
- NE2000 network card

The Malta emulation supports the following devices:

- Core board with MIPS 24Kf CPU and Galileo system controller
- PIIX4 PCI/USB/SMbus controller
- The Multi-I/O chip’s serial device

- PCI network cards (PCnet32 and others)
- Malta FPGA serial device
- Cirrus (default) or any other PCI VGA graphics card

The Boston board emulation supports the following devices:

- Xilinx FPGA, which includes a PCIe root port and an UART
- Intel EG20T PCH connects the I/O peripherals, but only the SATA bus is emulated

The ACER Pica emulation supports:

- MIPS R4000 CPU
- PC-style IRQ and DMA controllers
- PC Keyboard
- IDE controller

The MIPS Magnum R4000 emulation supports:

- MIPS R4000 CPU
- PC-style IRQ controller
- PC Keyboard
- SCSI controller
- G364 framebuffer

The Fuloong 2E emulation supports:

- Loongson 2E CPU
- Bonito64 system controller as North Bridge
- VT82C686 chipset as South Bridge
- RTL8139D as a network card chipset

The mipssim pseudo board emulation provides an environment similar to what the proprietary MIPS emulator uses for running Linux. It supports:

- A range of MIPS CPUs, default is the 24Kf
- PC style serial port
- MIPSnet network emulation

Supported CPU model configurations on MIPS hosts

QEMU supports variety of MIPS CPU models:

Supported CPU models for MIPS32 hosts

The following CPU models are supported for use on MIPS32 hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

`mips32r6-generic` MIPS32 Processor (Release 6, 2015)

P5600 MIPS32 Processor (P5600, 2014)

M14K, M14Kc MIPS32 Processor (M14K, 2009)

74Kf MIPS32 Processor (74K, 2007)

34Kf MIPS32 Processor (34K, 2006)

24Kc, 24KEc, 24Kf MIPS32 Processor (24K, 2003)

4Kc, 4Km, 4KEcR1, 4KEmR1, 4KEc, 4KEm MIPS32 Processor (4K, 1999)

Supported CPU models for MIPS64 hosts

The following CPU models are supported for use on MIPS64 hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

I6400 MIPS64 Processor (Release 6, 2014)

Loongson-2F MIPS64 Processor (Loongson 2, 2008)

Loongson-2E MIPS64 Processor (Loongson 2, 2006)

mips64dspr2 MIPS64 Processor (Release 2, 2006)

MIPS64R2-generic, 5KEc, 5KEf MIPS64 Processor (Release 2, 2002)

20Kc MIPS64 Processor (20K, 2000

5Kc, 5Kf MIPS64 Processor (5K, 1999)

VR5432 MIPS64 Processor (VR, 1998)

R4000 MIPS64 Processor (MIPS III, 1991)

Supported CPU models for nanoMIPS hosts

The following CPU models are supported for use on nanoMIPS hosts. Administrators / applications are recommended to use the CPU model that matches the generation of the host CPUs in use. In a deployment with a mixture of host CPU models between machines, if live migration compatibility is required, use the newest CPU model that is compatible across all desired hosts.

I7200 MIPS I7200 (nanoMIPS, 2018)

Preferred CPU models for MIPS hosts

The following CPU models are preferred for use on different MIPS hosts:

MIPS III R4000

MIPS32R2 34Kf

MIPS64R6 I6400

nanoMIPS I7200

nanoMIPS System emulator

Executable `qemu-system-mipsel` also covers simulation of 32-bit nanoMIPS system in little endian mode:

- nanoMIPS I7200 CPU

Example of `qemu-system-mipsel` usage for nanoMIPS is shown below:

Download `<disk_image_file>` from https://mipsdistros.mips.com/LinuxDistro/nanomips/buildroot/index.html.

Download `<kernel_image_file>` from https://mipsdistros.mips.com/LinuxDistro/nanomips/kernels/v4.15.18-432-gb2eb9a8b07a1-20180627102142/index.html.

Start system emulation of Malta board with nanoMIPS I7200 CPU:

```
qemu-system-mipsel -cpu I7200 -kernel <kernel_image_file> \
    -M malta -serial stdio -m <memory_size> -hda <disk_image_file> \
    -append "mem=256m@0x0 rw console=ttyS0 vga=cirrus vesa=0x111 root=/dev/sda"
```

1.15.6 Arm System emulator

QEMU can emulate both 32-bit and 64-bit Arm CPUs. Use the `qemu-system-aarch64` executable to simulate a 64-bit Arm machine. You can use either `qemu-system-arm` or `qemu-system-aarch64` to simulate a 32-bit Arm machine: in general, command lines that work for `qemu-system-arm` will behave the same when used with `qemu-system-aarch64`.

QEMU has generally good support for Arm guests. It has support for nearly fifty different machines. The reason we support so many is that Arm hardware is much more widely varying than x86 hardware. Arm CPUs are generally built into "system-on-chip" (SoC) designs created by many different companies with different devices, and these SoCs are then built into machines which can vary still further even if they use the same SoC. Even with fifty boards QEMU does not cover more than a small fraction of the Arm hardware ecosystem.

The situation for 64-bit Arm is fairly similar, except that we don't implement so many different machines.

As well as the more common "A-profile" CPUs (which have MMUs and will run Linux) QEMU also supports "M-profile" CPUs such as the Cortex-M0, Cortex-M4 and Cortex-M33 (which are microcontrollers used in very embedded boards). For most boards the CPU type is fixed (matching what the hardware has), so typically you don't need to specify the CPU type by hand, except for special cases like the `virt` board.

Choosing a board model

For QEMU's Arm system emulation, you must specify which board model you want to use with the `-M` or `--machine` option; there is no default.

Because Arm systems differ so much and in fundamental ways, typically operating system or firmware images intended to run on one machine will not run at all on any other. This is often surprising for new users who are used to the x86 world where every system looks like a standard PC. (Once the kernel has booted, most userspace software cares much less about the detail of the hardware.)

If you already have a system image or a kernel that works on hardware and you want to boot with QEMU, check whether QEMU lists that machine in its `-machine help` output. If it is listed, then you can probably use that board model. If it is not listed, then unfortunately your image will almost certainly not boot on QEMU. (You might be able to extract the filesystem and use that with a different kernel which boots on a system that QEMU does emulate.)

If you don't care about reproducing the idiosyncrasies of a particular bit of hardware, such as small amount of RAM, no PCI or other hard disk, etc., and just want to run Linux, the best option is to use the `virt` board. This is a platform which doesn't correspond to any real hardware and is designed for use in virtual machines. You'll need to compile

Linux with a suitable configuration for running on the `virt` board. `virt` supports PCI, virtio, recent CPUs and large amounts of RAM. It also supports 64-bit CPUs.

Board-specific documentation

Unfortunately many of the Arm boards QEMU supports are currently undocumented; you can get a complete list by running `qemu-system-aarch64 --machine help`.

Arm Integrator/CP (`integratorcp`)

The Arm Integrator/CP board is emulated with the following devices:

- ARM926E, ARM1026E, ARM946E, ARM1136 or Cortex-A8 CPU
- Two PL011 UARTs
- SMC 91c111 Ethernet adapter
- PL110 LCD controller
- PL050 KMI with PS/2 keyboard and mouse.
- PL181 MultiMedia Card Interface with SD card.

Arm MPS2 boards (`mps2-an385`, `mps2-an505`, `mps2-an511`, `mps2-an521`)

These board models all use Arm M-profile CPUs.

The Arm MPS2 and MPS2+ dev boards are FPGA based (the 2+ has a bigger FPGA but is otherwise the same as the 2). Since the CPU itself and most of the devices are in the FPGA, the details of the board as seen by the guest depend significantly on the FPGA image.

QEMU models the following FPGA images:

`mps2-an385` Cortex-M3 as documented in ARM Application Note AN385

`mps2-an511` Cortex-M3 'DesignStart' as documented in AN511

`mps2-an505` Cortex-M33 as documented in ARM Application Note AN505

`mps2-an521` Dual Cortex-M33 as documented in Application Note AN521

Differences between QEMU and real hardware:

- AN385 remapping of low 16K of memory to either ZBT SSRAM1 or to block RAM is unimplemented (QEMU always maps this to ZBT SSRAM1, as if zbt_boot_ctrl is always zero)
- QEMU provides a LAN9118 ethernet rather than LAN9220; the only guest visible difference is that the LAN9118 doesn't support checksum offloading

Arm Musca boards (`musca-a`, `musca-b1`)

The Arm Musca development boards are a reference implementation of a system using the SSE-200 Subsystem for Embedded. They are dual Cortex-M33 systems.

QEMU provides models of the A and B1 variants of this board.

Unimplemented devices:

- SPI
- I^2C
- I^2S
- PWM
- QSPI
- Timer
- SCC
- GPIO
- eFlash
- MHU
- PVT
- SDIO
- CryptoCell

Note that (like the real hardware) the Musca-A machine is asymmetric: CPU 0 does not have the FPU or DSP extensions, but CPU 1 does. Also like the real hardware, the memory maps for the A and B1 variants differ significantly, so guest software must be built for the right variant.

Arm Realview boards (`realview-eb`, `realview-eb-mpcore`, `realview-pb-a8`, `realview-pbx-a9`)

Several variants of the Arm RealView baseboard are emulated, including the EB, PB-A8 and PBX-A9. Due to interactions with the bootloader, only certain Linux kernel configurations work out of the box on these boards.

Kernels for the PB-A8 board should have CONFIG_REALVIEW_HIGH_PHYS_OFFSET enabled in the kernel, and expect 512M RAM. Kernels for The PBX-A9 board should have CONFIG_SPARSEMEM enabled, CONFIG_REALVIEW_HIGH_PHYS_OFFSET disabled and expect 1024M RAM.

The following devices are emulated:

- ARM926E, ARM1136, ARM11MPCore, Cortex-A8 or Cortex-A9 MPCore CPU
- Arm AMBA Generic/Distributed Interrupt Controller
- Four PL011 UARTs
- SMC 91c111 or SMSC LAN9118 Ethernet adapter
- PL110 LCD controller
- PL050 KMI with PS/2 keyboard and mouse
- PCI host bridge
- PCI OHCI USB controller
- LSI53C895A PCI SCSI Host Bus Adapter with hard disk and CD-ROM devices
- PL181 MultiMedia Card Interface with SD card.

Arm Versatile boards (`versatileab`, `versatilepb`)

The Arm Versatile baseboard is emulated with the following devices:

- ARM926E, ARM1136 or Cortex-A8 CPU
- PL190 Vectored Interrupt Controller
- Four PL011 UARTs
- SMC 91c111 Ethernet adapter
- PL110 LCD controller
- PL050 KMI with PS/2 keyboard and mouse.
- PCI host bridge. Note the emulated PCI bridge only provides access to PCI memory space. It does not provide access to PCI IO space. This means some devices (eg. ne2k_pci NIC) are not usable, and others (eg. rtl8139 NIC) are only usable when the guest drivers use the memory mapped control registers.
- PCI OHCI USB controller.
- LSI53C895A PCI SCSI Host Bus Adapter with hard disk and CD-ROM devices.
- PL181 MultiMedia Card Interface with SD card.

Arm Versatile Express boards (`vexpress-a9`, `vexpress-a15`)

QEMU models two variants of the Arm Versatile Express development board family:

- `vexpress-a9` models the combination of the Versatile Express motherboard and the CoreTile Express A9x4 daughterboard
- `vexpress-a15` models the combination of the Versatile Express motherboard and the CoreTile Express A15x2 daughterboard

Note that as this hardware does not have PCI, IDE or SCSI, the only available storage option is emulated SD card.

Implemented devices:

- PL041 audio
- PL181 SD controller
- PL050 keyboard and mouse
- PL011 UARTs
- SP804 timers
- I2C controller
- PL031 RTC
- PL111 LCD display controller
- Flash memory
- LAN9118 ethernet

Unimplemented devices:

- SP810 system control block
- PCI-express
- USB controller (Philips ISP1761)

- Local DAP ROM
- CoreSight interfaces
- PL301 AXI interconnect
- SCC
- System counter
- HDLCD controller (`vexpress-a15`)
- SP805 watchdog
- PL341 dynamic memory controller
- DMA330 DMA controller
- PL354 static memory controller
- BP147 TrustZone Protection Controller
- TrustZone Address Space Controller

Other differences between the hardware and the QEMU model:

- QEMU will default to creating one CPU unless you pass a different `-smp` argument
- QEMU allows the amount of RAM provided to be specified with the `-m` argument
- QEMU defaults to providing a CPU which does not provide either TrustZone or the Virtualization Extensions: if you want these you must enable them with `-machine secure=on` and `-machine virtualization=on`
- QEMU provides 4 virtio-mmio virtio transports; these start at address `0x10013000` for `vexpress-a9` and at `0x1c130000` for `vexpress-a15`, and have IRQs from 40 upwards. If a dtb is provided on the command line then QEMU will edit it to include suitable entries describing these transports for the guest.

Aspeed family boards (`*-bmc`, `ast2500-evb`, `ast2600-evb`)

The QEMU Aspeed machines model BMCs of various OpenPOWER systems and Aspeed evaluation boards. They are based on different releases of the Aspeed SoC : the AST2400 integrating an ARM926EJ-S CPU (400MHz), the AST2500 with an ARM1176JZS CPU (800MHz) and more recently the AST2600 with dual cores ARM Cortex A7 CPUs (1.2GHz).

The SoC comes with RAM, Gigabit ethernet, USB, SD/MMC, USB, SPI, I2C, etc.

AST2400 SoC based machines :

- `palmetto-bmc` OpenPOWER Palmetto POWER8 BMC

AST2500 SoC based machines :

- `ast2500-evb` Aspeed AST2500 Evaluation board
- `romulus-bmc` OpenPOWER Romulus POWER9 BMC
- `witherspoon-bmc` OpenPOWER Witherspoon POWER9 BMC
- `sonorapass-bmc` OCP SonoraPass BMC
- `swift-bmc` OpenPOWER Swift BMC POWER9

AST2600 SoC based machines :

- `ast2600-evb` Aspeed AST2600 Evaluation board (Cortex A7)

- `tacoma-bmc` OpenPOWER Witherspoon POWER9 AST2600 BMC

Supported devices

- SMP (for the AST2600 Cortex-A7)
- Interrupt Controller (VIC)
- Timer Controller
- RTC Controller
- I2C Controller
- System Control Unit (SCU)
- SRAM mapping
- X-DMA Controller (basic interface)
- Static Memory Controller (SMC or FMC) - Only SPI Flash support
- SPI Memory Controller
- USB 2.0 Controller
- SD/MMC storage controllers
- SDRAM controller (dummy interface for basic settings and training)
- Watchdog Controller
- GPIO Controller (Master only)
- UART
- Ethernet controllers

Missing devices

- Coprocessor support
- ADC (out of tree implementation)
- PWM and Fan Controller
- LPC Bus Controller
- Slave GPIO Controller
- Super I/O Controller
- Hash/Crypto Engine
- PCI-Express 1 Controller
- Graphic Display Controller
- PECI Controller
- MCTP Controller
- Mailbox Controller
- Virtual UART

- eSPI Controller
- I3C Controller

Boot options

The Aspeed machines can be started using the -kernel option to load a Linux kernel or from a firmare image which can be downloaded from the OpenPOWER jenkins :

> https://openpower.xyz/

The image should be attached as an MTD drive. Run :

```
$ qemu-system-arm -M romulus-bmc -nic user \
      -drive file=flash-romulus,format=raw,if=mtd -nographic
```

Canon A1100 (`canon-a1100`)

This machine is a model of the Canon PowerShot A1100 camera, which uses the DIGIC SoC. This model is based on reverse engineering efforts by the contributors to the CHDK and Magic Lantern projects.

The emulation is incomplete. In particular it can't be used to run the original camera firmware, but it can successfully run an experimental version of the barebox bootloader.

Freecom MusicPal (`musicpal`)

The Freecom MusicPal internet radio emulation includes the following elements:

- Marvell MV88W8618 Arm core.
- 32 MB RAM, 256 KB SRAM, 8 MB flash.
- Up to 2 16550 UARTs
- MV88W8xx8 Ethernet controller
- MV88W8618 audio controller, WM8750 CODEC and mixer
- 128x64 display with brightness control
- 2 buttons, 2 navigation wheels with button function

Gumstix Connex and Verdex (`connex`, `verdex`)

These machines model the Gumstix Connex and Verdex boards. The Connex has a PXA255 CPU and the Verdex has a PXA270.

Implemented devices:

- NOR flash
- SMC91C111 ethernet
- Interrupt controller
- DMA
- Timer

- GPIO
- MMC/SD card
- Fast infra-red communications port (FIR)
- LCD controller
- Synchronous serial ports (SPI)
- PCMCIA interface
- I2C
- I2S

Nokia N800 and N810 tablets (`n800`, `n810`)

Nokia N800 and N810 internet tablets (known also as RX-34 and RX-44 / 48) emulation supports the following elements:

- Texas Instruments OMAP2420 System-on-chip (ARM1136 core)
- RAM and non-volatile OneNAND Flash memories
- Display connected to EPSON remote framebuffer chip and OMAP on-chip display controller and a LS041y3 MIPI DBI-C controller
- TI TSC2301 (in N800) and TI TSC2005 (in N810) touchscreen controllers driven through SPI bus
- National Semiconductor LM8323-controlled qwerty keyboard driven through I^{2}C bus
- Secure Digital card connected to OMAP MMC/SD host
- Three OMAP on-chip UARTs and on-chip STI debugging console
- Mentor Graphics “Inventra” dual-role USB controller embedded in a TI TUSB6010 chip - only USB host mode is supported
- TI TMP105 temperature sensor driven through I^{2}C bus
- TI TWL92230C power management companion with an RTC on I^{2}C bus
- Nokia RETU and TAHVO multi-purpose chips with an RTC, connected through CBUS

Orange Pi PC (`orangepi-pc`)

The Xunlong Orange Pi PC is an Allwinner H3 System on Chip based embedded computer with mainline support in both U-Boot and Linux. The board comes with a Quad Core Cortex-A7 @ 1.3GHz, 1GiB RAM, 100Mbit ethernet, USB, SD/MMC, USB, HDMI and various other I/O.

Supported devices

The Orange Pi PC machine supports the following devices:

- SMP (Quad Core Cortex-A7)
- Generic Interrupt Controller configuration
- SRAM mappings
- SDRAM controller

- Real Time Clock
- Timer device (re-used from Allwinner A10)
- UART
- SD/MMC storage controller
- EMAC ethernet
- USB 2.0 interfaces
- Clock Control Unit
- System Control module
- Security Identifier device

Limitations

Currently, Orange Pi PC does *not* support the following features:

- Graphical output via HDMI, GPU and/or the Display Engine
- Audio output
- Hardware Watchdog

Also see the 'unimplemented' array in the Allwinner H3 SoC module for a complete list of unimplemented I/O devices: `./hw/arm/allwinner-h3.c`

Boot options

The Orange Pi PC machine can start using the standard -kernel functionality for loading a Linux kernel or ELF executable. Additionally, the Orange Pi PC machine can also emulate the BootROM which is present on an actual Allwinner H3 based SoC, which loads the bootloader from a SD card, specified via the -sd argument to qemu-system-arm.

Machine-specific options

The following machine-specific options are supported:

- allwinner-rtc.base-year=YYYY

  The Allwinner RTC device is automatically created by the Orange Pi PC machine and uses a default base year value which can be overridden using the 'base-year' property. The base year is the actual represented year when the RTC year value is zero. This option can be used in case the target operating system driver uses a different base year value. The minimum value for the base year is 1900.

- allwinner-sid.identifier=abcd1122-a000-b000-c000-12345678ffff

  The Security Identifier value can be read by the guest. For example, U-Boot uses it to determine a unique MAC address.

The above machine-specific options can be specified in qemu-system-arm via the '-global' argument, for example:

```
$ qemu-system-arm -M orangepi-pc -sd mycard.img \
     -global allwinner-rtc.base-year=2000
```

Running mainline Linux

Mainline Linux kernels from 4.19 up to latest master are known to work. To build a Linux mainline kernel that can be booted by the Orange Pi PC machine, simply configure the kernel using the sunxi_defconfig configuration:

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make mrproper
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make sunxi_defconfig
```

To be able to use USB storage, you need to manually enable the corresponding configuration item. Start the kconfig configuration tool:

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make menuconfig
```

Navigate to the following item, enable it and save your configuration:

> Device Drivers > USB support > USB Mass Storage support

Build the Linux kernel with:

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make
```

To boot the newly build linux kernel in QEMU with the Orange Pi PC machine, use:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
    -kernel /path/to/linux/arch/arm/boot/zImage \
    -append 'console=ttyS0,115200' \
    -dtb /path/to/linux/arch/arm/boot/dts/sun8i-h3-orangepi-pc.dtb
```

Orange Pi PC images

Note that the mainline kernel does not have a root filesystem. You may provide it with an official Orange Pi PC image from the official website:

> http://www.orangepi.org/downloadresources/

Another possibility is to run an Armbian image for Orange Pi PC which can be downloaded from:

> https://www.armbian.com/orange-pi-pc/

Alternatively, you can also choose to build you own image with buildroot using the orangepi_pc_defconfig. Also see https://buildroot.org for more information.

When using an image as an SD card, it must be resized to a power of two. This can be done with the qemu-img command. It is recommended to only increase the image size instead of shrinking it to a power of two, to avoid loss of data. For example, to prepare a downloaded Armbian image, first extract it and then increase its size to one gigabyte as follows:

```
$ qemu-img resize Armbian_19.11.3_Orangepipc_bionic_current_5.3.9.img 1G
```

You can choose to attach the selected image either as an SD card or as USB mass storage. For example, to boot using the Orange Pi PC Debian image on SD card, simply add the -sd argument and provide the proper root= kernel parameter:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
    -kernel /path/to/linux/arch/arm/boot/zImage \
    -append 'console=ttyS0,115200 root=/dev/mmcblk0p2' \
    -dtb /path/to/linux/arch/arm/boot/dts/sun8i-h3-orangepi-pc.dtb \
    -sd OrangePi_pc_debian_stretch_server_linux5.3.5_v1.0.img
```

To attach the image as an USB mass storage device to the machine, simply append to the command:

```
-drive if=none,id=stick,file=myimage.img \
-device usb-storage,bus=usb-bus.0,drive=stick
```

Instead of providing a custom Linux kernel via the -kernel command you may also choose to let the Orange Pi PC machine load the bootloader from SD card, just like a real board would do using the BootROM. Simply pass the selected image via the -sd argument and remove the -kernel, -append, -dbt and -initrd arguments:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
     -sd Armbian_19.11.3_Orangepipc_buster_current_5.3.9.img
```

Note that both the official Orange Pi PC images and Armbian images start a lot of userland programs via systemd. Depending on the host hardware and OS, they may be slow to emulate, especially due to emulating the 4 cores. To help reduce the performance slow down due to emulating the 4 cores, you can give the following kernel parameters via U-Boot (or via -append):

```
=> setenv extraargs 'systemd.default_timeout_start_sec=9000 loglevel=7 nosmp␣
↪console=ttyS0,115200'
```

Running U-Boot

U-Boot mainline can be build and configured using the orangepi_pc_defconfig using similar commands as describe above for Linux. Note that it is recommended for development/testing to select the following configuration setting in U-Boot:

> Device Tree Control > Provider for DTB for DT Control > Embedded DTB

To start U-Boot using the Orange Pi PC machine, provide the u-boot binary to the -kernel argument:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
    -kernel /path/to/uboot/u-boot -sd disk.img
```

Use the following U-boot commands to load and boot a Linux kernel from SD card:

```
=> setenv bootargs console=ttyS0,115200
=> ext2load mmc 0 0x42000000 zImage
=> ext2load mmc 0 0x43000000 sun8i-h3-orangepi-pc.dtb
=> bootz 0x42000000 - 0x43000000
```

Running NetBSD

The NetBSD operating system also includes support for Allwinner H3 based boards, including the Orange Pi PC. NetBSD 9.0 is known to work best for the Orange Pi PC board and provides a fully working system with serial console, networking and storage. For the Orange Pi PC machine, get the 'evbarm-earmv7hf' based image from:

> https://cdn.netbsd.org/pub/NetBSD/NetBSD-9.0/evbarm-earmv7hf/binary/gzimg/armv7.img.gz

The image requires manually installing U-Boot in the image. Build U-Boot with the orangepi_pc_defconfig configuration as described in the previous section. Next, unzip the NetBSD image and write the U-Boot binary including SPL using:

```
$ gunzip armv7.img.gz
$ dd if=/path/to/u-boot-sunxi-with-spl.bin of=armv7.img bs=1024 seek=8 conv=notrunc
```

Finally, before starting the machine the SD image must be extended such that the size of the SD image is a power of two and that the NetBSD kernel will not conclude the NetBSD partition is larger than the emulated SD card:

```
$ qemu-img resize armv7.img 2G
```

Start the machine using the following command:

```
$ qemu-system-arm -M orangepi-pc -nic user -nographic \
      -sd armv7.img -global allwinner-rtc.base-year=2000
```

At the U-Boot stage, interrupt the automatic boot process by pressing a key and set the following environment variables before booting:

```
=> setenv bootargs root=ld0a
=> setenv kernel netbsd-GENERIC.ub
=> setenv fdtfile dtb/sun8i-h3-orangepi-pc.dtb
=> setenv bootcmd 'fatload mmc 0:1 ${kernel_addr_r} ${kernel}; fatload mmc 0:1 ${fdt_
↪addr_r} ${fdtfile}; fdt addr ${fdt_addr_r}; bootm ${kernel_addr_r} - ${fdt_addr_r}'
```

Optionally you may save the environment variables to SD card with 'saveenv'. To continue booting simply give the 'boot' command and NetBSD boots.

Orange Pi PC acceptance tests

The Orange Pi PC machine has several acceptance tests included. To run the whole set of tests, build QEMU from source and simply provide the following command:

```
$ AVOCADO_ALLOW_LARGE_STORAGE=yes avocado --show=app,console run \
   -t machine:orangepi-pc tests/acceptance/boot_linux_console.py
```

Palm Tungsten|E PDA (`cheetah`)

The Palm Tungsten|E PDA (codename "Cheetah") emulation includes the following elements:

- Texas Instruments OMAP310 System-on-chip (ARM925T core)
- ROM and RAM memories (ROM firmware image can be loaded with -option-rom)
- On-chip LCD controller
- On-chip Real Time Clock
- TI TSC2102i touchscreen controller / analog-digital converter / Audio CODEC, connected through MicroWire and I^2S busses
- GPIO-connected matrix keypad
- Secure Digital card connected to OMAP MMC/SD host
- Three on-chip UARTs

Sharp XScale-based PDA models (`akita`, `borzoi`, `spitz`, `terrier`)

The XScale-based clamshell PDA models ("Spitz", "Akita", "Borzoi" and "Terrier") emulation includes the following peripherals:

- Intel PXA270 System-on-chip (ARMv5TE core)

- NAND Flash memory
- IBM/Hitachi DSCM microdrive in a PXA PCMCIA slot - not in “Akita”
- On-chip OHCI USB controller
- On-chip LCD controller
- On-chip Real Time Clock
- TI ADS7846 touchscreen controller on SSP bus
- Maxim MAX1111 analog-digital converter on I^{2}C bus
- GPIO-connected keyboard controller and LEDs
- Secure Digital card connected to PXA MMC/SD host
- Three on-chip UARTs
- WM8750 audio CODEC on I^{2}C and I^{2}S busses

Sharp Zaurus SL-5500 (`collie`)

This machine is a model of the Sharp Zaurus SL-5500, which was a 1990s PDA based on the StrongARM SA1110.

Implemented devices:

- NOR flash
- Interrupt controller
- Timer
- RTC
- GPIO
- Peripheral Pin Controller (PPC)
- UARTs
- Synchronous Serial Ports (SSP)

Siemens SX1 (`sx1`, `sx1-v1`)

The Siemens SX1 models v1 and v2 (default) basic emulation. The emulation includes the following elements:

- Texas Instruments OMAP310 System-on-chip (ARM925T core)
- ROM and RAM memories (ROM firmware image can be loaded with -pflash) V1 1 Flash of 16MB and 1 Flash of 8MB V2 1 Flash of 32MB
- On-chip LCD controller
- On-chip Real Time Clock
- Secure Digital card connected to OMAP MMC/SD host
- Three on-chip UARTs

Stellaris boards (`lm3s6965evb`, `lm3s811evb`)

The Luminary Micro Stellaris LM3S811EVB emulation includes the following devices:

- Cortex-M3 CPU core.
- 64k Flash and 8k SRAM.
- Timers, UARTs, ADC and I^{2}C interface.
- OSRAM Pictiva 96x16 OLED with SSD0303 controller on I^{2}C bus.

The Luminary Micro Stellaris LM3S6965EVB emulation includes the following devices:

- Cortex-M3 CPU core.
- 256k Flash and 64k SRAM.
- Timers, UARTs, ADC, I^{2}C and SSI interfaces.
- OSRAM Pictiva 128x64 OLED with SSD0323 controller connected via SSI.

'virt' generic virtual platform (`virt`)

The *virt* board is a platform which does not correspond to any real hardware; it is designed for use in virtual machines. It is the recommended board type if you simply want to run a guest such as Linux and do not care about reproducing the idiosyncrasies and limitations of a particular bit of real-world hardware.

This is a "versioned" board model, so as well as the `virt` machine type itself (which may have improvements, bugfixes and other minor changes between QEMU versions) a version is provided that guarantees to have the same behaviour as that of previous QEMU releases, so that VM migration will work between QEMU versions. For instance the `virt-5.0` machine type will behave like the `virt` machine from the QEMU 5.0 release, and migration should work between `virt-5.0` of the 5.0 release and `virt-5.0` of the 5.1 release. Migration is not guaranteed to work between different QEMU releases for the non-versioned `virt` machine type.

Supported devices

The virt board supports:

- PCI/PCIe devices
- Flash memory
- One PL011 UART
- An RTC
- The fw_cfg device that allows a guest to obtain data from QEMU
- A PL061 GPIO controller
- An optional SMMUv3 IOMMU
- hotpluggable DIMMs
- hotpluggable NVDIMMs
- An MSI controller (GICv2M or ITS). GICv2M is selected by default along with GICv2. ITS is selected by default with GICv3 (>= virt-2.7). Note that ITS is not modeled in TCG mode.
- 32 virtio-mmio transport devices
- running guests using the KVM accelerator on aarch64 hardware

- large amounts of RAM (at least 255GB, and more if using highmem)
- many CPUs (up to 512 if using a GICv3 and highmem)
- Secure-World-only devices if the CPU has TrustZone:
  - A second PL011 UART
  - A secure flash memory
  - 16MB of secure RAM

Supported guest CPU types:

- `cortex-a7` (32-bit)
- `cortex-a15` (32-bit; the default)
- `cortex-a53` (64-bit)
- `cortex-a57` (64-bit)
- `cortex-a72` (64-bit)
- `host` (with KVM only)
- `max` (same as `host` for KVM; best possible emulation with TCG)

Note that the default is `cortex-a15`, so for an AArch64 guest you must specify a CPU type.

Graphics output is available, but unlike the x86 PC machine types there is no default display device enabled: you should select one from the Display devices section of "-device help". The recommended option is `virtio-gpu-pci`; this is the only one which will work correctly with KVM. You may also need to ensure your guest kernel is configured with support for this; see below.

Machine-specific options

The following machine-specific options are supported:

secure Set `on`/`off` to enable/disable emulating a guest CPU which implements the Arm Security Extensions (TrustZone). The default is `off`.

virtualization Set `on`/`off` to enable/disable emulating a guest CPU which implements the Arm Virtualization Extensions. The default is `off`.

mte Set `on`/`off` to enable/disable emulating a guest CPU which implements the Arm Memory Tagging Extensions. The default is `off`.

highmem Set `on`/`off` to enable/disable placing devices and RAM in physical address space above 32 bits. The default is `on` for machine types later than `virt-2.12`.

gic-version Specify the version of the Generic Interrupt Controller (GIC) to provide. Valid values are:

`2` GICv2

`3` GICv3

`host` Use the same GIC version the host provides, when using KVM

`max` Use the best GIC version possible (same as host when using KVM; currently same as `3`` for TCG, but this may change in future)

its Set `on`/`off` to enable/disable ITS instantiation. The default is `on` for machine types later than `virt-2.7`.

iommu Set the IOMMU type to create for the guest. Valid values are:

none Don't create an IOMMU (the default)

smmuv3 Create an SMMUv3

ras Set `on`/`off` to enable/disable reporting host memory errors to a guest using ACPI and guest external abort exceptions. The default is off.

Linux guest kernel configuration

The 'defconfig' for Linux arm and arm64 kernels should include the right device drivers for virtio and the PCI controller; however some older kernel versions, especially for 32-bit Arm, did not have everything enabled by default. If you're not seeing PCI devices that you expect, then check that your guest config has:

```
CONFIG_PCI=y
CONFIG_VIRTIO_PCI=y
CONFIG_PCI_HOST_GENERIC=y
```

If you want to use the `virtio-gpu-pci` graphics device you will also need:

```
CONFIG_DRM=y
CONFIG_DRM_VIRTIO_GPU=y
```

Hardware configuration information for bare-metal programming

The `virt` board automatically generates a device tree blob ("dtb") which it passes to the guest. This provides information about the addresses, interrupt lines and other configuration of the various devices in the system. Guest code can rely on and hard-code the following addresses:

- Flash memory starts at address 0x0000_0000
- RAM starts at 0x4000_0000

All other information about device locations may change between QEMU versions, so guest code must look in the DTB.

QEMU supports two types of guest image boot for `virt`, and the way for the guest code to locate the dtb binary differs:

- For guests using the Linux kernel boot protocol (this means any non-ELF file passed to the QEMU `-kernel` option) the address of the DTB is passed in a register (`r2` for 32-bit guests, or `x0` for 64-bit guests)
- For guests booting as "bare-metal" (any other kind of boot), the DTB is at the start of RAM (0x4000_0000)

Xilinx Versal Virt (`xlnx-versal-virt`)

Xilinx Versal is a family of heterogeneous multi-core SoCs (System on Chip) that combine traditional hardened CPUs and I/O peripherals in a Processing System (PS) with runtime programmable FPGA logic (PL) and an Artificial Intelligence Engine (AIE).

More details here: https://www.xilinx.com/products/silicon-devices/acap/versal.html

The family of Versal SoCs share a single architecture but come in different parts with different speed grades, amounts of PL and other differences.

The Xilinx Versal Virt board in QEMU is a model of a virtual board (does not exist in reality) with a virtual Versal SoC without I/O limitations. Currently, we support the following cores and devices:

Implemented CPU cores:

- 2 ACPUs (ARM Cortex-A72)

Implemented devices:

- Interrupt controller (ARM GICv3)
- 2 UARTs (ARM PL011)
- An RTC (Versal built-in)
- 2 GEMs (Cadence MACB Ethernet MACs)
- 8 ADMA (Xilinx zDMA) channels
- 2 SD Controllers
- OCM (256KB of On Chip Memory)
- DDR memory

QEMU does not yet model any other devices, including the PL and the AI Engine.

Other differences between the hardware and the QEMU model:

- QEMU allows the amount of DDR memory provided to be specified with the `-m` argument. If a DTB is provided on the command line then QEMU will edit it to include suitable entries describing the Versal DDR memory ranges.
- QEMU provides 8 virtio-mmio virtio transports; these start at address `0xa0000000` and have IRQs from 111 and upwards.

Running

If the user provides an Operating System to be loaded, we expect users to use the `-kernel` command line option.

Users can load firmware or boot-loaders with the `-device loader` options.

When loading an OS, QEMU generates a DTB and selects an appropriate address where it gets loaded. This DTB will be passed to the kernel in register x0.

If there's no `-kernel` option, we generate a DTB and place it at 0x1000 for boot-loaders or firmware to pick it up.

If users want to provide their own DTB, they can use the `-dtb` option. These DTBs will have their memory nodes modified to match QEMU's selected ram_size option before they get passed to the kernel or FW.

When loading an OS, we turn on QEMU's PSCI implementation with SMC as the PSCI conduit. When there's no `-kernel` option, we assume the user provides EL3 firmware to handle PSCI.

A few examples:

Direct Linux boot of a generic ARM64 upstream Linux kernel:

```
$ qemu-system-aarch64 -M xlnx-versal-virt -m 2G \
    -serial mon:stdio -display none \
    -kernel arch/arm64/boot/Image \
    -nic user -nic user \
    -device virtio-rng-device,bus=virtio-mmio-bus.0 \
    -drive if=none,index=0,file=hd0.qcow2,id=hd0,snapshot \
    -drive file=qemu_sd.qcow2,if=sd,index=0,snapshot \
    -device virtio-blk-device,drive=hd0 -append root=/dev/vda
```

Direct Linux boot of PetaLinux 2019.2:

```
$ qemu-system-aarch64  -M xlnx-versal-virt -m 2G \
    -serial mon:stdio -display none \
    -kernel petalinux-v2019.2/Image \
    -append "rdinit=/sbin/init console=ttyAMA0,115200n8 earlycon=pl011,mmio,
↪0xFF000000,115200n8" \
    -net nic,model=cadence_gem,netdev=net0 -netdev user,id=net0 \
    -device virtio-rng-device,bus=virtio-mmio-bus.0,rng=rng0 \
    -object rng-random,filename=/dev/urandom,id=rng0
```

Boot PetaLinux 2019.2 via ARM Trusted Firmware (2018.3 because the 2019.2 version of ATF tries to configure the CCI which we don't model) and U-boot:

```
$ qemu-system-aarch64 -M xlnx-versal-virt -m 2G \
    -serial stdio -display none \
    -device loader,file=petalinux-v2018.3/bl31.elf,cpu-num=0 \
    -device loader,file=petalinux-v2019.2/u-boot.elf \
    -device loader,addr=0x20000000,file=petalinux-v2019.2/Image \
    -nic user -nic user \
    -device virtio-rng-device,bus=virtio-mmio-bus.0,rng=rng0 \
    -object rng-random,filename=/dev/urandom,id=rng0
```

Run the following at the U-Boot prompt:

```
Versal>
fdt addr $fdtcontroladdr
fdt move $fdtcontroladdr 0x40000000
fdt set /timer clock-frequency <0x3dfd240>
setenv bootargs "rdinit=/sbin/init maxcpus=1 console=ttyAMA0,115200n8 earlycon=pl011,
↪mmio,0xFF000000,115200n8"
booti 20000000 - 40000000
fdt addr $fdtcontroladdr
```

Boot Linux as DOM0 on Xen via U-Boot:

```
$ qemu-system-aarch64 -M xlnx-versal-virt -m 4G \
    -serial stdio -display none \
    -device loader,file=petalinux-v2019.2/u-boot.elf,cpu-num=0 \
    -device loader,addr=0x30000000,file=linux/2018-04-24/xen \
    -device loader,addr=0x40000000,file=petalinux-v2019.2/Image \
    -nic user -nic user \
    -device virtio-rng-device,bus=virtio-mmio-bus.0,rng=rng0 \
    -object rng-random,filename=/dev/urandom,id=rng0
```

Run the following at the U-Boot prompt:

```
Versal>
fdt addr $fdtcontroladdr
fdt move $fdtcontroladdr 0x20000000
fdt set /timer clock-frequency <0x3dfd240>
fdt set /chosen xen,xen-bootargs "console=dtuart dtuart=/uart@ff000000 dom0_mem=640M␣
↪bootscrub=0 maxcpus=1 timer_slop=0"
fdt set /chosen xen,dom0-bootargs "rdinit=/sbin/init clk_ignore_unused console=hvc0␣
↪maxcpus=1"
fdt mknode /chosen dom0
fdt set /chosen/dom0 compatible "xen,multiboot-module"
fdt set /chosen/dom0 reg <0x00000000 0x40000000 0x0 0x03100000>
booti 30000000 - 20000000
```

Boot Linux as Dom0 on Xen via ARM Trusted Firmware and U-Boot:

```
$ qemu-system-aarch64 -M xlnx-versal-virt -m 4G \
    -serial stdio -display none \
    -device loader,file=petalinux-v2018.3/bl31.elf,cpu-num=0 \
    -device loader,file=petalinux-v2019.2/u-boot.elf \
    -device loader,addr=0x30000000,file=linux/2018-04-24/xen \
    -device loader,addr=0x40000000,file=petalinux-v2019.2/Image \
    -nic user -nic user \
    -device virtio-rng-device,bus=virtio-mmio-bus.0,rng=rng0 \
    -object rng-random,filename=/dev/urandom,id=rng0
```

Run the following at the U-Boot prompt:

```
Versal>
fdt addr $fdtcontroladdr
fdt move $fdtcontroladdr 0x20000000
fdt set /timer clock-frequency <0x3dfd240>
fdt set /chosen xen,xen-bootargs "console=dtuart dtuart=/uart@ff000000 dom0_mem=640M␣
↪bootscrub=0 maxcpus=1 timer_slop=0"
fdt set /chosen xen,dom0-bootargs "rdinit=/sbin/init clk_ignore_unused console=hvc0␣
↪maxcpus=1"
fdt mknode /chosen dom0
fdt set /chosen/dom0 compatible "xen,multiboot-module"
fdt set /chosen/dom0 reg <0x00000000 0x40000000 0x0 0x03100000>
booti 30000000 - 20000000
```

Arm CPU features

Arm CPU Features

CPU features are optional features that a CPU of supporting type may choose to implement or not. In QEMU, optional CPU features have corresponding boolean CPU proprieties that, when enabled, indicate that the feature is implemented, and, conversely, when disabled, indicate that it is not implemented. An example of an Arm CPU feature is the Performance Monitoring Unit (PMU). CPU types such as the Cortex-A15 and the Cortex-A57, which respectively implement Arm architecture reference manuals ARMv7-A and ARMv8-A, may both optionally implement PMUs. For example, if a user wants to use a Cortex-A15 without a PMU, then the *-cpu* parameter should contain *pmu=off* on the QEMU command line, i.e. *-cpu cortex-a15,pmu=off*.

As not all CPU types support all optional CPU features, then whether or not a CPU property exists depends on the CPU type. For example, CPUs that implement the ARMv8-A architecture reference manual may optionally support the AArch32 CPU feature, which may be enabled by disabling the *aarch64* CPU property. A CPU type such as the Cortex-A15, which does not implement ARMv8-A, will not have the *aarch64* CPU property.

QEMU's support may be limited for some CPU features, only partially supporting the feature or only supporting the feature under certain configurations. For example, the *aarch64* CPU feature, which, when disabled, enables the optional AArch32 CPU feature, is only supported when using the KVM accelerator and when running on a host CPU type that supports the feature. While *aarch64* currently only works with KVM, it could work with TCG. CPU features that are specific to KVM are prefixed with "kvm-" and are described in "KVM VCPU Features".

CPU Feature Probing

Determining which CPU features are available and functional for a given CPU type is possible with the *query-cpu-model-expansion* QMP command. Below are some examples where *scripts/qmp/qmp-shell* (see the top comment block in the script for usage) is used to issue the QMP commands.

1. Determine which CPU features are available for the *max* CPU type (Note, we started QEMU with qemu-system-aarch64, so *max* is implementing the ARMv8-A reference manual in this case):

```
(QEMU) query-cpu-model-expansion type=full model={"name":"max"}
{ "return": {
  "model": { "name": "max", "props": {
  "sve1664": true, "pmu": true, "sve1792": true, "sve1920": true,
  "sve128": true, "aarch64": true, "sve1024": true, "sve": true,
  "sve640": true, "sve768": true, "sve1408": true, "sve256": true,
  "sve1152": true, "sve512": true, "sve384": true, "sve1536": true,
  "sve896": true, "sve1280": true, "sve2048": true
}}}}
```

We see that the *max* CPU type has the *pmu*, *aarch64*, *sve*, and many *sve<N>* CPU features. We also see that all the CPU features are enabled, as they are all *true*. (The *sve<N>* CPU features are all optional SVE vector lengths (see "SVE CPU Properties"). While with TCG all SVE vector lengths can be supported, when KVM is in use it's more likely that only a few lengths will be supported, if SVE is supported at all.)

(2) Let's try to disable the PMU:

```
(QEMU) query-cpu-model-expansion type=full model={"name":"max","props":{"pmu
↪":false}}
{ "return": {
  "model": { "name": "max", "props": {
  "sve1664": true, "pmu": false, "sve1792": true, "sve1920": true,
  "sve128": true, "aarch64": true, "sve1024": true, "sve": true,
  "sve640": true, "sve768": true, "sve1408": true, "sve256": true,
  "sve1152": true, "sve512": true, "sve384": true, "sve1536": true,
  "sve896": true, "sve1280": true, "sve2048": true
}}}}
```

We see it worked, as *pmu* is now *false*.

(3) Let's try to disable *aarch64*, which enables the AArch32 CPU feature:

```
(QEMU) query-cpu-model-expansion type=full model={"name":"max","props":{"aarch64
↪":false}}
{"error": {
 "class": "GenericError", "desc":
 "'aarch64' feature cannot be disabled unless KVM is enabled and 32-bit EL1 is␣
↪supported"
}}
```

It looks like this feature is limited to a configuration we do not currently have.

(4) Let's disable *sve* and see what happens to all the optional SVE vector lengths:

```
(QEMU) query-cpu-model-expansion type=full model={"name":"max","props":{"sve
↪":false}}
{ "return": {
  "model": { "name": "max", "props": {
  "sve1664": false, "pmu": true, "sve1792": false, "sve1920": false,
  "sve128": false, "aarch64": true, "sve1024": false, "sve": false,
  "sve640": false, "sve768": false, "sve1408": false, "sve256": false,
  "sve1152": false, "sve512": false, "sve384": false, "sve1536": false,
  "sve896": false, "sve1280": false, "sve2048": false
}}}}
```

As expected they are now all *false*.

(5) Let's try probing CPU features for the Cortex-A15 CPU type:

```
(QEMU) query-cpu-model-expansion type=full model={"name":"cortex-a15"}
{"return": {"model": {"name": "cortex-a15", "props": {"pmu": true}}}}
```

Only the *pmu* CPU feature is available.

A note about CPU feature dependencies

It's possible for features to have dependencies on other features. I.e. it may be possible to change one feature at a time without error, but when attempting to change all features at once an error could occur depending on the order they are processed. It's also possible changing all at once doesn't generate an error, because a feature's dependencies are satisfied with other features, but the same feature cannot be changed independently without error. For these reasons callers should always attempt to make their desired changes all at once in order to ensure the collection is valid.

A note about CPU models and KVM

Named CPU models generally do not work with KVM. There are a few cases that do work, e.g. using the named CPU model *cortex-a57* with KVM on a seattle host, but mostly if KVM is enabled the *host* CPU type must be used. This means the guest is provided all the same CPU features as the host CPU type has. And, for this reason, the *host* CPU type should enable all CPU features that the host has by default. Indeed it's even a bit strange to allow disabling CPU features that the host has when using the *host* CPU type, but in the absence of CPU models it's the best we can do if we want to launch guests without all the host's CPU features enabled.

Enabling KVM also affects the *query-cpu-model-expansion* QMP command. The affect is not only limited to specific features, as pointed out in example (3) of "CPU Feature Probing", but also to which CPU types may be expanded. When KVM is enabled, only the *max*, *host*, and current CPU type may be expanded. This restriction is necessary as it's not possible to know all CPU types that may work with KVM, but it does impose a small risk of users experiencing unexpected errors. For example on a seattle, as mentioned above, the *cortex-a57* CPU type is also valid when KVM is enabled. Therefore a user could use the *host* CPU type for the current type, but then attempt to query *cortex-a57*, however that query will fail with our restrictions. This shouldn't be an issue though as management layers and users have been preferring the *host* CPU type for use with KVM for quite some time. Additionally, if the KVM-enabled QEMU instance running on a seattle host is using the *cortex-a57* CPU type, then querying *cortex-a57* will work.

Using CPU Features

After determining which CPU features are available and supported for a given CPU type, then they may be selectively enabled or disabled on the QEMU command line with that CPU type:

```
$ qemu-system-aarch64 -M virt -cpu max,pmu=off,sve=on,sve128=on,sve256=on
```

The example above disables the PMU and enables the first two SVE vector lengths for the *max* CPU type. Note, the *sve=on* isn't actually necessary, because, as we observed above with our probe of the *max* CPU type, *sve* is already on by default. Also, based on our probe of defaults, it would seem we need to disable many SVE vector lengths, rather than only enabling the two we want. This isn't the case, because, as disabling many SVE vector lengths would be quite verbose, the *sve<N>* CPU properties have special semantics (see "SVE CPU Property Parsing Semantics").

KVM VCPU Features

KVM VCPU features are CPU features that are specific to KVM, such as paravirt features or features that enable CPU virtualization extensions. The features' CPU properties are only available when KVM is enabled and are named with the prefix "kvm-". KVM VCPU features may be probed, enabled, and disabled in the same way as other CPU features. Below is the list of KVM VCPU features and their descriptions.

kvm-no-adjvtime By default kvm-no-adjvtime is disabled. This means that by default the virtual time adjustment is enabled (vtime is not *not* adjusted).

When virtual time adjustment is enabled each time the VM transitions back to running state the VCPU's virtual counter is updated to ensure stopped time is not counted. This avoids time jumps surprising guest OSes and applications, as long as they use the virtual counter for timekeeping. However it has the side effect of the virtual and physical counters diverging. All timekeeping based on the virtual counter will appear to lag behind any timekeeping that does not subtract VM stopped time. The guest may resynchronize its virtual counter with other time sources as needed.

Enable kvm-no-adjvtime to disable virtual time adjustment, also restoring the legacy (pre-5.0) behavior.

SVE CPU Properties

There are two types of SVE CPU properties: *sve* and *sve<N>*. The first is used to enable or disable the entire SVE feature, just as the *pmu* CPU property completely enables or disables the PMU. The second type is used to enable or disable specific vector lengths, where *N* is the number of bits of the length. The *sve<N>* CPU properties have special dependencies and constraints, see "SVE CPU Property Dependencies and Constraints" below. Additionally, as we want all supported vector lengths to be enabled by default, then, in order to avoid overly verbose command lines (command lines full of *sve<N>=off*, for all *N* not wanted), we provide the parsing semantics listed in "SVE CPU Property Parsing Semantics".

SVE CPU Property Dependencies and Constraints

1) At least one vector length must be enabled when *sve* is enabled.
2) If a vector length *N* is enabled, then, when KVM is enabled, all smaller, host supported vector lengths must also be enabled. If KVM is not enabled, then only all the smaller, power-of-two vector lengths must be enabled. E.g. with KVM if the host supports all vector lengths up to 512-bits (128, 256, 384, 512), then if *sve512* is enabled, the 128-bit vector length, 256-bit vector length, and 384-bit vector length must also be enabled. Without KVM, the 384-bit vector length would not be required.
3) If KVM is enabled then only vector lengths that the host CPU type support may be enabled. If SVE is not supported by the host, then no *sve** properties may be enabled.

SVE CPU Property Parsing Semantics

1) If SVE is disabled (*sve=off*), then which SVE vector lengths are enabled or disabled is irrelevant to the guest, as the entire SVE feature is disabled and that disables all vector lengths for the guest. However QEMU will still track any *sve<N>* CPU properties provided by the user. If later an *sve=on* is provided, then the guest will get only the enabled lengths. If no *sve=on* is provided and there are explicitly enabled vector lengths, then an error is generated.
2) If SVE is enabled (*sve=on*), but no *sve<N>* CPU properties are provided, then all supported vector lengths are enabled, which when KVM is not in use means including the non-power-of-two lengths, and, when KVM is in use, it means all vector lengths supported by the host processor.
3) If SVE is enabled, then an error is generated when attempting to disable the last enabled vector length (see constraint (1) of "SVE CPU Property Dependencies and Constraints").

4) If one or more vector lengths have been explicitly enabled and at at least one of the dependency lengths of the maximum enabled length has been explicitly disabled, then an error is generated (see constraint (2) of "SVE CPU Property Dependencies and Constraints").
5) When KVM is enabled, if the host does not support SVE, then an error is generated when attempting to enable any *sve** properties (see constraint (3) of "SVE CPU Property Dependencies and Constraints").
6) When KVM is enabled, if the host does support SVE, then an error is generated when attempting to enable any vector lengths not supported by the host (see constraint (3) of "SVE CPU Property Dependencies and Constraints").
7) If one or more *sve<N>* CPU properties are set *off*, but no *sve<N>*, CPU properties are set *on*, then the specified vector lengths are disabled but the default for any unspecified lengths remains enabled. When KVM is not enabled, disabling a power-of-two vector length also disables all vector lengths larger than the power-of-two length. When KVM is enabled, then disabling any supported vector length also disables all larger vector lengths (see constraint (2) of "SVE CPU Property Dependencies and Constraints").
8) If one or more *sve<N>* CPU properties are set to *on*, then they are enabled and all unspecified lengths default to disabled, except for the required lengths per constraint (2) of "SVE CPU Property Dependencies and Constraints", which will even be auto-enabled if they were not explicitly enabled.
9) If SVE was disabled (*sve=off*), allowing all vector lengths to be explicitly disabled (i.e. avoiding the error specified in (3) of "SVE CPU Property Parsing Semantics"), then if later an *sve=on* is provided an error will be generated. To avoid this error, one must enable at least one vector length prior to enabling SVE.

SVE CPU Property Examples

1) Disable SVE:

```
$ qemu-system-aarch64 -M virt -cpu max,sve=off
```

2) Implicitly enable all vector lengths for the *max* CPU type:

```
$ qemu-system-aarch64 -M virt -cpu max
```

3) When KVM is enabled, implicitly enable all host CPU supported vector lengths with the *host* CPU type:

```
$ qemu-system-aarch64 -M virt,accel=kvm -cpu host
```

4) Only enable the 128-bit vector length:

```
$ qemu-system-aarch64 -M virt -cpu max,sve128=on
```

5) Disable the 512-bit vector length and all larger vector lengths, since 512 is a power-of-two. This results in all the smaller, uninitialized lengths (128, 256, and 384) defaulting to enabled:

```
$ qemu-system-aarch64 -M virt -cpu max,sve512=off
```

6) Enable the 128-bit, 256-bit, and 512-bit vector lengths:

```
$ qemu-system-aarch64 -M virt -cpu max,sve128=on,sve256=on,sve512=on
```

7) The same as (6), but since the 128-bit and 256-bit vector lengths are required for the 512-bit vector length to be enabled, then allow them to be auto-enabled:

```
$ qemu-system-aarch64 -M virt -cpu max,sve512=on
```

8) Do the same as (7), but by first disabling SVE and then re-enabling it:

```
$ qemu-system-aarch64 -M virt -cpu max,sve=off,sve512=on,sve=on
```

9) Force errors regarding the last vector length:

```
$ qemu-system-aarch64 -M virt -cpu max,sve128=off
$ qemu-system-aarch64 -M virt -cpu max,sve=off,sve128=off,sve=on
```

SVE CPU Property Recommendations

The examples in "SVE CPU Property Examples" exhibit many ways to select vector lengths which developers may find useful in order to avoid overly verbose command lines. However, the recommended way to select vector lengths is to explicitly enable each desired length. Therefore only example's (1), (4), and (6) exhibit recommended uses of the properties.

1.15.7 ColdFire System emulator

Use the executable `qemu-system-m68k` to simulate a ColdFire machine. The emulator is able to boot a uClinux kernel.

The M5208EVB emulation includes the following devices:

- MCF5208 ColdFire V2 Microprocessor (ISA A+ with EMAC).
- Three Two on-chip UARTs.
- Fast Ethernet Controller (FEC)

The AN5206 emulation includes the following devices:

- MCF5206 ColdFire V2 Microprocessor.
- Two on-chip UARTs.

1.15.8 Xtensa System emulator

Two executables cover simulation of both Xtensa endian options, `qemu-system-xtensa` and `qemu-system-xtensaeb`. Two different machine types are emulated:

- Xtensa emulator pseudo board "sim"
- Avnet LX60/LX110/LX200 board

The sim pseudo board emulation provides an environment similar to one provided by the proprietary Tensilica ISS. It supports:

- A range of Xtensa CPUs, default is the DC232B
- Console and filesystem access via semihosting calls

The Avnet LX60/LX110/LX200 emulation supports:

- A range of Xtensa CPUs, default is the DC232B
- 16550 UART
- OpenCores 10/100 Mbps Ethernet MAC

1.15.9 s390x System emulator

QEMU can emulate z/Architecture (in particular, 64 bit) s390x systems via the `qemu-system-s390x` binary. Only one machine type, `s390-ccw-virtio`, is supported (with versioning for compatibility handling).

When using KVM as accelerator, QEMU can emulate CPUs up to the generation of the host. When using the default cpu model with TCG as accelerator, QEMU will emulate a subset of z13 cpu features that should be enough to run distributions built for the z13.

Device support

QEMU will not emulate most of the traditional devices found under LPAR or z/VM; virtio devices (especially using virtio-ccw) make up the bulk of the available devices. Passthrough of host devices via vfio-pci, vfio-ccw, or vfio-ap is also available.

Adjunct Processor (AP) Device

Contents

Introduction

The IBM Adjunct Processor (AP) Cryptographic Facility is comprised of three AP instructions and from 1 to 256 PCIe cryptographic adapter cards. These AP devices provide cryptographic functions to all CPUs assigned to a linux system running in an IBM Z system LPAR.

On s390x, AP adapter cards are exposed via the AP bus. This document describes how those cards may be made available to KVM guests using the VFIO mediated device framework.

AP Architectural Overview

In order understand the terminology used in the rest of this document, let's start with some definitions:

- AP adapter

  An AP adapter is an IBM Z adapter card that can perform cryptographic functions. There can be from 0 to 256 adapters assigned to an LPAR depending on the machine model. Adapters assigned to the LPAR in which a linux host is running will be available to the linux host. Each adapter is identified by a number from 0 to 255; however, the maximum adapter number allowed is determined by machine model. When installed, an AP adapter is accessed by AP instructions executed by any CPU.

- AP domain

  An adapter is partitioned into domains. Each domain can be thought of as a set of hardware registers for processing AP instructions. An adapter can hold up to 256 domains; however, the maximum domain number allowed is determined by machine model. Each domain is identified by a number from 0 to 255. Domains can be further classified into two types:

  - Usage domains are domains that can be accessed directly to process AP commands
  - Control domains are domains that are accessed indirectly by AP commands sent to a usage domain to control or change the domain; for example, to set a secure private key for the domain.

- AP Queue

  An AP queue is the means by which an AP command-request message is sent to an AP usage domain inside a specific AP. An AP queue is identified by a tuple comprised of an AP adapter ID (APID) and an AP queue index (APQI). The APQI corresponds to a given usage domain number within the adapter. This tuple forms an AP Queue Number (APQN) uniquely identifying an AP queue. AP instructions include a field containing the APQN to identify the AP queue to which the AP command-request message is to be sent for processing.

- AP Instructions:

  There are three AP instructions:

  - NQAP: to enqueue an AP command-request message to a queue
  - DQAP: to dequeue an AP command-reply message from a queue
  - PQAP: to administer the queues

  AP instructions identify the domain that is targeted to process the AP command; this must be one of the usage domains. An AP command may modify a domain that is not one of the usage domains, but the modified domain must be one of the control domains.

Start Interpretive Execution (SIE) Instruction

A KVM guest is started by executing the Start Interpretive Execution (SIE) instruction. The SIE state description is a control block that contains the state information for a KVM guest and is supplied as input to the SIE instruction. The SIE state description contains a satellite control block called the Crypto Control Block (CRYCB). The CRYCB contains three fields to identify the adapters, usage domains and control domains assigned to the KVM guest:

- The AP Mask (APM) field is a bit mask that identifies the AP adapters assigned to the KVM guest. Each bit in the mask, from left to right, corresponds to an APID from 0-255. If a bit is set, the corresponding adapter is valid for use by the KVM guest.
- The AP Queue Mask (AQM) field is a bit mask identifying the AP usage domains assigned to the KVM guest. Each bit in the mask, from left to right, corresponds to an AP queue index (APQI) from 0-255. If a bit is set, the corresponding queue is valid for use by the KVM guest.

- The AP Domain Mask field is a bit mask that identifies the AP control domains assigned to the KVM guest. The ADM bit mask controls which domains can be changed by an AP command-request message sent to a usage domain from the guest. Each bit in the mask, from left to right, corresponds to a domain from 0-255. If a bit is set, the corresponding domain can be modified by an AP command-request message sent to a usage domain.

If you recall from the description of an AP Queue, AP instructions include an APQN to identify the AP adapter and AP queue to which an AP command-request message is to be sent (NQAP and PQAP instructions), or from which a command-reply message is to be received (DQAP instruction). The validity of an APQN is defined by the matrix calculated from the APM and AQM; it is the cross product of all assigned adapter numbers (APM) with all assigned queue indexes (AQM). For example, if adapters 1 and 2 and usage domains 5 and 6 are assigned to a guest, the APQNs (1,5), (1,6), (2,5) and (2,6) will be valid for the guest.

The APQNs can provide secure key functionality - i.e., a private key is stored on the adapter card for each of its domains - so each APQN must be assigned to at most one guest or the linux host.

Example 1: Valid configuration

	Guest1	Guest2
adapters	1, 2	1, 2
domains	5, 6	7

This is valid because both guests have a unique set of APQNs:

- Guest1 has APQNs (1,5), (1,6), (2,5) and (2,6);
- Guest2 has APQNs (1,7) and (2,7).

Example 2: Valid configuration

	Guest1	Guest2
adapters	1, 2	3, 4
domains	5, 6	5, 6

This is also valid because both guests have a unique set of APQNs:

- Guest1 has APQNs (1,5), (1,6), (2,5), (2,6);
- Guest2 has APQNs (3,5), (3,6), (4,5), (4,6)

Example 3: Invalid configuration

	Guest1	Guest2
adapters	1, 2	1
domains	5, 6	6, 7

This is an invalid configuration because both guests have access to APQN (1,6).

AP Matrix Configuration on Linux Host

A linux system is a guest of the LPAR in which it is running and has access to the AP resources configured for the LPAR. The LPAR's AP matrix is configured via its Activation Profile which can be edited on the HMC. When the linux system is started, the AP bus will detect the AP devices assigned to the LPAR and create the following in sysfs:

```
/sys/bus/ap
... [devices]
...... xx.yyyy
...... ...
...... cardxx
...... ...
```

Where:

cardxx is AP adapter number xx (in hex)

xx.yyyy is an APQN with xx specifying the APID and yyyy specifying the APQI

For example, if AP adapters 5 and 6 and domains 4, 71 (0x47), 171 (0xab) and 255 (0xff) are configured for the LPAR, the sysfs representation on the linux host system would look like this:

```
/sys/bus/ap
... [devices]
...... 05.0004
...... 05.0047
...... 05.00ab
...... 05.00ff
...... 06.0004
...... 06.0047
...... 06.00ab
...... 06.00ff
...... card05
...... card06
```

A set of default device drivers are also created to control each type of AP device that can be assigned to the LPAR on which a linux host is running:

```
/sys/bus/ap
... [drivers]
...... [cex2acard]        for Crypto Express 2/3 accelerator cards
...... [cex2aqueue]       for AP queues served by Crypto Express 2/3
                          accelerator cards
...... [cex4card]         for Crypto Express 4/5/6 accelerator and coprocessor
                          cards
...... [cex4queue]        for AP queues served by Crypto Express 4/5/6
                          accelerator and coprocessor cards
...... [pcixcccard]       for Crypto Express 2/3 coprocessor cards
...... [pcixccqueue]      for AP queues served by Crypto Express 2/3
                          coprocessor cards
```

Binding AP devices to device drivers

There are two sysfs files that specify bitmasks marking a subset of the APQN range as 'usable by the default AP queue device drivers' or 'not usable by the default device drivers' and thus available for use by the alternate device driver(s). The sysfs locations of the masks are:

```
/sys/bus/ap/apmask
/sys/bus/ap/aqmask
```

The `apmask` is a 256-bit mask that identifies a set of AP adapter IDs (APID). Each bit in the mask, from left to right (i.e., from most significant to least significant bit in big endian order), corresponds to an APID from 0-255. If a bit is set, the APID is marked as usable only by the default AP queue device drivers; otherwise, the APID is usable by the vfio_ap device driver.

The `aqmask` is a 256-bit mask that identifies a set of AP queue indexes (APQI). Each bit in the mask, from left to right (i.e., from most significant to least significant bit in big endian order), corresponds to an APQI from 0-255. If a bit is set, the APQI is marked as usable only by the default AP queue device drivers; otherwise, the APQI is usable by the vfio_ap device driver.

Take, for example, the following mask:

```
0x7dffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

It indicates:

> 1, 2, 3, 4, 5, and 7-255 belong to the default drivers' pool, and 0 and 6 belong to the vfio_ap device driver's pool.

The APQN of each AP queue device assigned to the linux host is checked by the AP bus against the set of APQNs derived from the cross product of APIDs and APQIs marked as usable only by the default AP queue device drivers. If a match is detected, only the default AP queue device drivers will be probed; otherwise, the vfio_ap device driver will be probed.

By default, the two masks are set to reserve all APQNs for use by the default AP queue device drivers. There are two ways the default masks can be changed:

1. The sysfs mask files can be edited by echoing a string into the respective sysfs mask file in one of two formats:

   - An absolute hex string starting with 0x - like "0x12345678" - sets the mask. If the given string is shorter than the mask, it is padded with 0s on the right; for example, specifying a mask value of 0x41 is the same as specifying:

     ```
     0x4100000000000000000000000000000000000000000000000000000000000000
     ```

     Keep in mind that the mask reads from left to right (i.e., most significant to least significant bit in big endian order), so the mask above identifies device numbers 1 and 7 (`01000001`).

     If the string is longer than the mask, the operation is terminated with an error (EINVAL).

   - Individual bits in the mask can be switched on and off by specifying each bit number to be switched in a comma separated list. Each bit number string must be prepended with a (+) or minus (−) to indicate the corresponding bit is to be switched on (+) or off (−). Some valid values are:

     ```
     "+0"    switches bit 0 on
     "-13"   switches bit 13 off
     "+0x41" switches bit 65 on
     "-0xff" switches bit 255 off
     ```

     The following example:

     ```
     +0,-6,+0x47,-0xf0
     ```

     Switches bits 0 and 71 (0x47) on Switches bits 6 and 240 (0xf0) off

     Note that the bits not specified in the list remain as they were before the operation.

2. The masks can also be changed at boot time via parameters on the kernel command line like this:

   ```
   ap.apmask=0xffff ap.aqmask=0x40
   ```

   This would create the following masks:

   apmask:

   ```
   0xffff000000000000000000000000000000000000000000000000000000000000
   ```

   aqmask:

   ```
   0x4000000000000000000000000000000000000000000000000000000000000000
   ```

   Resulting in these two pools:

   ```
   default drivers pool:    adapter 0-15, domain 1
   alternate drivers pool:  adapter 16-255, domains 0, 2-255
   ```

Configuring an AP matrix for a linux guest

The sysfs interfaces for configuring an AP matrix for a guest are built on the VFIO mediated device framework. To configure an AP matrix for a guest, a mediated matrix device must first be created for the `/sys/devices/vfio_ap/matrix` device. When the vfio_ap device driver is loaded, it registers with the VFIO mediated device framework. When the driver registers, the sysfs interfaces for creating mediated matrix devices is created:

```
/sys/devices
... [vfio_ap]
......[matrix]
......... [mdev_supported_types]
............ [vfio_ap-passthrough]
............... create
............... [devices]
```

A mediated AP matrix device is created by writing a UUID to the attribute file named `create`, for example:

```
uuidgen > create
```

or

```
echo $uuid > create
```

When a mediated AP matrix device is created, a sysfs directory named after the UUID is created in the `devices` subdirectory:

```
/sys/devices
... [vfio_ap]
......[matrix]
......... [mdev_supported_types]
............ [vfio_ap-passthrough]
............... create
............... [devices]
.................. [$uuid]
```

There will also be three sets of attribute files created in the mediated matrix device's sysfs directory to configure an AP matrix for the KVM guest:

```
/sys/devices
... [vfio_ap]
......[matrix]
......... [mdev_supported_types]
............ [vfio_ap-passthrough]
............... create
............... [devices]
.................. [$uuid]
..................... assign_adapter
..................... assign_control_domain
..................... assign_domain
..................... matrix
..................... unassign_adapter
..................... unassign_control_domain
..................... unassign_domain
```

assign_adapter To assign an AP adapter to the mediated matrix device, its APID is written to the `assign_adapter` file. This may be done multiple times to assign more than one adapter. The APID may be specified using conventional semantics as a decimal, hexadecimal, or octal number. For example, to assign adapters 4, 5 and 16 to a mediated matrix device in decimal, hexadecimal and octal respectively:

```
echo 4 > assign_adapter
echo 0x5 > assign_adapter
echo 020 > assign_adapter
```

In order to successfully assign an adapter:

- The adapter number specified must represent a value from 0 up to the maximum adapter number allowed by the machine model. If an adapter number higher than the maximum is specified, the operation will terminate with an error (ENODEV).
- All APQNs that can be derived from the adapter ID being assigned and the IDs of the previously assigned domains must be bound to the vfio_ap device driver. If no domains have yet been assigned, then there must be at least one APQN with the specified APID bound to the vfio_ap driver. If no such APQNs are bound to the driver, the operation will terminate with an error (EADDRNOTAVAIL).
- No APQN that can be derived from the adapter ID and the IDs of the previously assigned domains can be assigned to another mediated matrix device. If an APQN is assigned to another mediated matrix device, the operation will terminate with an error (EADDRINUSE).

unassign_adapter To unassign an AP adapter, its APID is written to the `unassign_adapter` file. This may also be done multiple times to unassign more than one adapter.

assign_domain To assign a usage domain, the domain number is written into the `assign_domain` file. This may be done multiple times to assign more than one usage domain. The domain number is specified using conventional semantics as a decimal, hexadecimal, or octal number. For example, to assign usage domains 4, 8, and 71 to a mediated matrix device in decimal, hexadecimal and octal respectively:

```
echo 4 > assign_domain
echo 0x8 > assign_domain
echo 0107 > assign_domain
```

In order to successfully assign a domain:

- The domain number specified must represent a value from 0 up to the maximum domain number allowed by the machine model. If a domain number higher than the maximum is specified, the operation will terminate with an error (ENODEV).

- All APQNs that can be derived from the domain ID being assigned and the IDs of the previously assigned adapters must be bound to the vfio_ap device driver. If no domains have yet been assigned, then there must be at least one APQN with the specified APQI bound to the vfio_ap driver. If no such APQNs are bound to the driver, the operation will terminate with an error (EADDRNOTAVAIL).
- No APQN that can be derived from the domain ID being assigned and the IDs of the previously assigned adapters can be assigned to another mediated matrix device. If an APQN is assigned to another mediated matrix device, the operation will terminate with an error (EADDRINUSE).

`unassign_domain` To unassign a usage domain, the domain number is written into the `unassign_domain` file. This may be done multiple times to unassign more than one usage domain.

`assign_control_domain` To assign a control domain, the domain number is written into the `assign_control_domain` file. This may be done multiple times to assign more than one control domain. The domain number may be specified using conventional semantics as a decimal, hexadecimal, or octal number. For example, to assign control domains 4, 8, and 71 to a mediated matrix device in decimal, hexadecimal and octal respectively:

```
echo 4 > assign_domain
echo 0x8 > assign_domain
echo 0107 > assign_domain
```

In order to successfully assign a control domain, the domain number specified must represent a value from 0 up to the maximum domain number allowed by the machine model. If a control domain number higher than the maximum is specified, the operation will terminate with an error (ENODEV).

`unassign_control_domain` To unassign a control domain, the domain number is written into the `unassign_domain` file. This may be done multiple times to unassign more than one control domain.

Notes: No changes to the AP matrix will be allowed while a guest using the mediated matrix device is running. Attempts to assign an adapter, domain or control domain will be rejected and an error (EBUSY) returned.

Starting a Linux Guest Configured with an AP Matrix

To provide a mediated matrix device for use by a guest, the following option must be specified on the QEMU command line:

```
-device vfio_ap,sysfsdev=$path-to-mdev
```

The sysfsdev parameter specifies the path to the mediated matrix device. There are a number of ways to specify this path:

```
/sys/devices/vfio_ap/matrix/$uuid
/sys/bus/mdev/devices/$uuid
/sys/bus/mdev/drivers/vfio_mdev/$uuid
/sys/devices/vfio_ap/matrix/mdev_supported_types/vfio_ap-passthrough/devices/$uuid
```

When the linux guest is started, the guest will open the mediated matrix device's file descriptor to get information about the mediated matrix device. The `vfio_ap` device driver will update the APM, AQM, and ADM fields in the guest's CRYCB with the adapter, usage domain and control domains assigned via the mediated matrix device's sysfs attribute files. Programs running on the linux guest will then:

1. Have direct access to the APQNs derived from the cross product of the AP adapter numbers (APID) and queue indexes (APQI) specified in the APM and AQM fields of the guests's CRYCB respectively. These APQNs identify the AP queues that are valid for use by the guest; meaning, AP commands can be sent by the guest to any of these queues for processing.

2. Have authorization to process AP commands to change a control domain identified in the ADM field of the guest's CRYCB. The AP command must be sent to a valid APQN (see 1 above).

CPU model features:

Three CPU model features are available for controlling guest access to AP facilities:

1. AP facilities feature

   The AP facilities feature indicates that AP facilities are installed on the guest. This feature will be exposed for use only if the AP facilities are installed on the host system. The feature is s390-specific and is represented as a parameter of the -cpu option on the QEMU command line:

   ```
   qemu-system-s390x -cpu $model,ap=on|off
   ```

   Where:

   $model is the CPU model defined for the guest (defaults to the model of the host system if not specified).

   ap=on|off indicates whether AP facilities are installed (on) or not (off). The default for CPU models zEC12 or newer is `ap=on`. AP facilities must be installed on the guest if a vfio-ap device (`-device vfio-ap,sysfsdev=$path`) is configured for the guest, or the guest will fail to start.

2. Query Configuration Information (QCI) facility

   The QCI facility is used by the AP bus running on the guest to query the configuration of the AP facilities. This facility will be available only if the QCI facility is installed on the host system. The feature is s390-specific and is represented as a parameter of the -cpu option on the QEMU command line:

   ```
   qemu-system-s390x -cpu $model,apqci=on|off
   ```

   Where:

   $model is the CPU model defined for the guest

   apqci=on|off indicates whether the QCI facility is installed (on) or not (off). The default for CPU models zEC12 or newer is `apqci=on`; for older models, QCI will not be installed.

   If QCI is installed (`apqci=on`) but AP facilities are not (`ap=off`), an error message will be logged, but the guest will be allowed to start. It makes no sense to have QCI installed if the AP facilities are not; this is considered an invalid configuration.

   If the QCI facility is not installed, APQNs with an APQI greater than 15 will not be detected by the AP bus running on the guest.

3. Adjunct Process Facility Test (APFT) facility

   The APFT facility is used by the AP bus running on the guest to test the AP facilities available for a given AP queue. This facility will be available only if the APFT facility is installed on the host system. The feature is s390-specific and is represented as a parameter of the -cpu option on the QEMU command line:

   ```
   qemu-system-s390x -cpu $model,apft=on|off
   ```

   Where:

   $model is the CPU model defined for the guest (defaults to the model of the host system if not specified).

   apft=on|off indicates whether the APFT facility is installed (on) or not (off). The default for CPU models zEC12 and newer is `apft=on` for older models, APFT will not be installed.

If APFT is installed (`apft=on`) but AP facilities are not (`ap=off`), an error message will be logged, but the guest will be allowed to start. It makes no sense to have APFT installed if the AP facilities are not; this is considered an invalid configuration.

It also makes no sense to turn APFT off because the AP bus running on the guest will not detect CEX4 and newer devices without it. Since only CEX4 and newer devices are supported for guest usage, no AP devices can be made accessible to a guest started without APFT installed.

Hot plug a vfio-ap device into a running guest

Only one vfio-ap device can be attached to the virtual machine's ap-bus, so a vfio-ap device can be hot plugged if and only if no vfio-ap device is attached to the bus already, whether via the QEMU command line or a prior hot plug action.

To hot plug a vfio-ap device, use the QEMU `device_add` command:

```
(qemu) device_add vfio-ap,sysfsdev="$path-to-mdev",id="$id"
```

Where the `$path-to-mdev` value specifies the absolute path to a mediated device to which AP resources to be used by the guest have been assigned. `$id` is the name value for the optional id parameter.

Note that on Linux guests, the AP devices will be created in the `/sys/bus/ap/devices` directory when the AP bus subsequently performs its periodic scan, so there may be a short delay before the AP devices are accessible on the guest.

The command will fail if:

- A vfio-ap device has already been attached to the virtual machine's ap-bus.
- The CPU model features for controlling guest access to AP facilities are not enabled (see 'CPU model features' subsection in the previous section).

Hot unplug a vfio-ap device from a running guest

A vfio-ap device can be unplugged from a running KVM guest if a vfio-ap device has been attached to the virtual machine's ap-bus via the QEMU command line or a prior hot plug action.

To hot unplug a vfio-ap device, use the QEMU `device_del` command:

```
(qemu) device_del "$id"
```

Where `$id` is the same id that was specified at device creation.

On a Linux guest, the AP devices will be removed from the `/sys/bus/ap/devices` directory on the guest when the AP bus subsequently performs its periodic scan, so there may be a short delay before the AP devices are no longer accessible by the guest.

The command will fail if the `$path-to-mdev` specified on the `device_del` command does not match the value specified when the vfio-ap device was attached to the virtual machine's ap-bus.

Example: Configure AP Matrices for Three Linux Guests

Let's now provide an example to illustrate how KVM guests may be given access to AP facilities. For this example, we will show how to configure three guests such that executing the lszcrypt command on the guests would look like this:

Guest1:

```
CARD.DOMAIN TYPE  MODE
------------------------------
05          CEX5C CCA-Coproc
05.0004     CEX5C CCA-Coproc
05.00ab     CEX5C CCA-Coproc
06          CEX5A Accelerator
06.0004     CEX5A Accelerator
06.00ab     CEX5C CCA-Coproc
```

Guest2:

```
CARD.DOMAIN TYPE  MODE
------------------------------
05          CEX5A Accelerator
05.0047     CEX5A Accelerator
05.00ff     CEX5A Accelerator
```

Guest3:

```
CARD.DOMAIN TYPE  MODE
------------------------------
06          CEX5A Accelerator
06.0047     CEX5A Accelerator
06.00ff     CEX5A Accelerator
```

These are the steps:

1. Install the vfio_ap module on the linux host. The dependency chain for the vfio_ap module is:

   - iommu
   - s390
   - zcrypt
   - vfio
   - vfio_mdev
   - vfio_mdev_device
   - KVM

   To build the vfio_ap module, the kernel build must be configured with the following Kconfig elements selected:

   - IOMMU_SUPPORT
   - S390
   - ZCRYPT
   - S390_AP_IOMMU
   - VFIO
   - VFIO_MDEV
   - VFIO_MDEV_DEVICE
   - KVM

   If using make menuconfig select the following to build the vfio_ap module::

      -> Device Drivers

-> IOMMU Hardware Support select S390 AP IOMMU Support

-> VFIO Non-Privileged userspace driver framework

-> Mediated device driver framework -> VFIO driver for Mediated devices

-> I/O subsystem -> VFIO support for AP devices

2. Secure the AP queues to be used by the three guests so that the host can not access them. To secure the AP queues 05.0004, 05.0047, 05.00ab, 05.00ff, 06.0004, 06.0047, 06.00ab, and 06.00ff for use by the vfio_ap device driver, the corresponding APQNs must be removed from the default queue drivers pool as follows:

```
echo -5,-6 > /sys/bus/ap/apmask

echo -4,-0x47,-0xab,-0xff > /sys/bus/ap/aqmask
```

This will result in AP queues 05.0004, 05.0047, 05.00ab, 05.00ff, 06.0004, 06.0047, 06.00ab, and 06.00ff getting bound to the vfio_ap device driver. The sysfs directory for the vfio_ap device driver will now contain symbolic links to the AP queue devices bound to it:

```
/sys/bus/ap
... [drivers]
...... [vfio_ap]
......... [05.0004]
......... [05.0047]
......... [05.00ab]
......... [05.00ff]
......... [06.0004]
......... [06.0047]
......... [06.00ab]
......... [06.00ff]
```

Keep in mind that only type 10 and newer adapters (i.e., CEX4 and later) can be bound to the vfio_ap device driver. The reason for this is to simplify the implementation by not needlessly complicating the design by supporting older devices that will go out of service in the relatively near future, and for which there are few older systems on which to test.

The administrator, therefore, must take care to secure only AP queues that can be bound to the vfio_ap device driver. The device type for a given AP queue device can be read from the parent card's sysfs directory. For example, to see the hardware type of the queue 05.0004:

```
cat /sys/bus/ap/devices/card05/hwtype
```

The hwtype must be 10 or higher (CEX4 or newer) in order to be bound to the vfio_ap device driver.

3. Create the mediated devices needed to configure the AP matrixes for the three guests and to provide an interface to the vfio_ap driver for use by the guests:

```
/sys/devices/vfio_ap/matrix/
... [mdev_supported_types]
...... [vfio_ap-passthrough] (passthrough mediated matrix device type)
......... create
......... [devices]
```

To create the mediated devices for the three guests:

```
uuidgen > create
uuidgen > create
uuidgen > create
```

or

```
echo $uuid1 > create
echo $uuid2 > create
echo $uuid3 > create
```

This will create three mediated devices in the [devices] subdirectory named after the UUID used to create the mediated device. We'll call them $uuid1, $uuid2 and $uuid3 and this is the sysfs directory structure after creation:

```
/sys/devices/vfio_ap/matrix/
... [mdev_supported_types]
...... [vfio_ap-passthrough]
......... [devices]
............ [$uuid1]
............... assign_adapter
............... assign_control_domain
............... assign_domain
............... matrix
............... unassign_adapter
............... unassign_control_domain
............... unassign_domain

............ [$uuid2]
............... assign_adapter
............... assign_control_domain
............... assign_domain
............... matrix
............... unassign_adapter
............... unassign_control_domain
............... unassign_domain

............ [$uuid3]
............... assign_adapter
............... assign_control_domain
............... assign_domain
............... matrix
............... unassign_adapter
............... unassign_control_domain
............... unassign_domain
```

4. The administrator now needs to configure the matrixes for the mediated devices $uuid1 (for Guest1), $uuid2 (for Guest2) and $uuid3 (for Guest3).

   This is how the matrix is configured for Guest1:

   ```
   echo 5 > assign_adapter
   echo 6 > assign_adapter
   echo 4 > assign_domain
   echo 0xab > assign_domain
   ```

   Control domains can similarly be assigned using the assign_control_domain sysfs file.

   If a mistake is made configuring an adapter, domain or control domain, you can use the `unassign_xxx` interfaces to unassign the adapter, domain or control domain.

   To display the matrix configuration for Guest1:

```
cat matrix
```

The output will display the APQNs in the format `xx.yyyy`, where xx is the adapter number and yyyy is the domain number. The output for Guest1 will look like this:

```
05.0004
05.00ab
06.0004
06.00ab
```

This is how the matrix is configured for Guest2:

```
echo 5 > assign_adapter
echo 0x47 > assign_domain
echo 0xff > assign_domain
```

This is how the matrix is configured for Guest3:

```
echo 6 > assign_adapter
echo 0x47 > assign_domain
echo 0xff > assign_domain
```

5. Start Guest1:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on -device vfio-ap,
↪sysfsdev=/sys/devices/vfio_ap/matrix/$uuid1 ...
```

7. Start Guest2:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on -device vfio-ap,
↪sysfsdev=/sys/devices/vfio_ap/matrix/$uuid2 ...
```

7. Start Guest3:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on -device vfio-ap,
↪sysfsdev=/sys/devices/vfio_ap/matrix/$uuid3 ...
```

When the guest is shut down, the mediated matrix devices may be removed.

Using our example again, to remove the mediated matrix device $uuid1:

```
/sys/devices/vfio_ap/matrix/
... [mdev_supported_types]
...... [vfio_ap-passthrough]
......... [devices]
............ [$uuid1]
............... remove


echo 1 > remove
```

This will remove all of the mdev matrix device's sysfs structures including the mdev device itself. To recreate and reconfigure the mdev matrix device, all of the steps starting with step 3 will have to be performed again. Note that the remove will fail if a guest using the mdev is still running.

It is not necessary to remove an mdev matrix device, but one may want to remove it if no guest will use it during the remaining lifetime of the linux host. If the mdev matrix device is removed, one may want to also reconfigure the pool of adapters and queues reserved for use by the default drivers.

Limitations

- The KVM/kernel interfaces do not provide a way to prevent restoring an APQN to the default drivers pool of a queue that is still assigned to a mediated device in use by a guest. It is incumbent upon the administrator to ensure there is no mediated device in use by a guest to which the APQN is assigned lest the host be given access to the private data of the AP queue device, such as a private key configured specifically for the guest.
- Dynamically assigning AP resources to or unassigning AP resources from a mediated matrix device - see *Configuring an AP matrix for a linux guest* section above - while a running guest is using it is currently not supported.
- Live guest migration is not supported for guests using AP devices. If a guest is using AP devices, the vfio-ap device configured for the guest must be unplugged before migrating the guest (see *Hot unplug a vfio-ap device from a running guest* section above.)

The virtual channel subsystem

QEMU implements a virtual channel subsystem with subchannels, (mostly functionless) channel paths, and channel devices (virtio-ccw, 3270, and devices passed via vfio-ccw). It supports multiple subchannel sets (MSS) and multiple channel subsystems extended (MCSS-E).

All channel devices support the `devno` property, which takes a parameter in the form `<cssid>.<ssid>.<device number>`.

The default channel subsystem image id (`<cssid>`) is `0xfe`. Devices in there will show up in channel subsystem image `0` to guests that do not enable MCSS-E. Note that devices with a different cssid will not be visible if the guest OS does not enable MCSS-E (which is true for all supported guest operating systems today).

Supported values for the subchannel set id (`<ssid>`) range from `0-3`. Devices with a ssid that is not `0` will not be visible if the guest OS does not enable MSS (any Linux version that supports virtio also enables MSS). Any device may be put into any subchannel set, there is no restriction by device type.

The device number can range from `0-0xffff`.

If the `devno` property is not specified for a device, QEMU will choose the next free device number in subchannel set 0, skipping to the next subchannel set if no more device numbers are free.

QEMU places a device at the first free subchannel in the specified subchannel set. If a device is hotunplugged and later replugged, it may appear at a different subchannel. (This is similar to how z/VM works.)

Examples

- a virtio-net device, cssid/ssid/devno automatically assigned:

  ```
  -device virtio-net-ccw
  ```

  In a Linux guest (without default devices and no other devices specified prior to this one), this will show up as `0.0.0000` under subchannel `0.0.0000`.

  The auto-assigned-properties in QEMU (as seen via e.g. `info qtree`) would be `dev_id = "fe.0.0000"` and `subch_id = "fe.0.0000"`.

- a virtio-rng device in subchannel set `0`:

  ```
  -device virtio-rng-ccw,devno=fe.0.0042
  ```

  If added to the same Linux guest as above, it would show up as `0.0.0042` under subchannel `0.0.0001`.

  The properties for the device would be `dev_id = "fe.0.0042"` and `subch_id = "fe.0.0001"`.

- a virtio-gpu device in subchannel set 2:

  ```
  -device virtio-gpu-ccw,devno=fe.2.1111
  ```

  If added to the same Linux guest as above, it would show up as `0.2.1111` under subchannel `0.2.0000`.

  The properties for the device would be `dev_id = "fe.2.1111"` and `subch_id = "fe.2.0000"`.

- a virtio-mouse device in a non-standard channel subsystem image:

  ```
  -device virtio-mouse-ccw,devno=2.0.2222
  ```

  This would not show up in a standard Linux guest.

  The properties for the device would be `dev_id = "2.0.2222"` and `subch_id = "2.0.0000"`.

- a virtio-keyboard device in another non-standard channel subsystem image:

  ```
  -device virtio-keyboard-ccw,devno=0.0.1234
  ```

  This would not show up in a standard Linux guest, either, as `0` is not the standard channel subsystem image id.

  The properties for the device would be `dev_id = "0.0.1234"` and `subch_id = "0.0.0000"`.

3270 devices

The 3270 is the classic 'green-screen' console of the mainframes (see the IBM 3270 Wikipedia article).

The 3270 data stream is not implemented within QEMU; the device only provides TN3270 (a telnet extension; see RFC 854 and RFC 1576) and leaves the heavy lifting to an external 3270 terminal emulator (such as `x3270`) to make a single 3270 device available to a guest. Note that this supports basic features only.

To provide a 3270 device to a guest, create a `x-terminal3270` linked to a `tn3270` chardev. The guest will see a 3270 channel device. In order to actually be able to use it, attach the `x3270` emulator to the chardev.

Example configuration

- Make sure that 3270 support is enabled in the guest's Linux kernel. You need `CONFIG_TN3270` and at least one of `CONFIG_TN3270_TTY` (for additional ttys) or `CONFIG_TN3270_CONSOLE` (for a 3270 console).
- Add a `tn3270` chardev and a `x-terminal3270` to the QEMU command line:

  ```
  -chardev socket,id=ch0,host=0.0.0.0,port=2300,nowait,server,tn3270
  -device x-terminal3270,chardev=ch0,devno=fe.0.000a,id=terminal0
  ```

- Start the guest. In the guest, use `chccwdev -e 0.0.000a` to enable the device.
- On the host, start the `x3270` emulator:

  ```
  x3270 <host>:2300
  ```

- In the guest, locate the 3270 device node under `/dev/3270/` (say, `tty1`) and start a getty on it:

  ```
  systemctl start serial-getty@3270-tty1.service
  ```

  This should get you an additional tty for logging into the guest.

- If you want to use the 3270 device as the Linux kernel console instead of an additional tty, you can also append `conmode=3270 condev=000a` to the guest's kernel command line. The kernel then should use the 3270 as console after the next boot.

Restrictions

3270 support is very basic. In particular:

- Only one 3270 device is supported.
- It has only been tested with Linux guests and the x3270 emulator.
- TLS/SSL is not supported.
- Resizing on reattach is not supported.
- Multiple commands in one inbound buffer (for example, when the reset key is pressed while the network is slow) are not supported.

Subchannel passthrough via vfio-ccw

vfio-ccw (based upon the mediated vfio device infrastructure) allows to make certain I/O subchannels and their devices available to a guest. The host will not interact with those subchannels/devices any more.

Note that while vfio-ccw should work with most non-QDIO devices, only ECKD DASDs have really been tested.

Example configuration

Step 1: configure the host device

As every mdev is identified by a uuid, the first step is to obtain one:

```
[root@host ~]# uuidgen
7e270a25-e163-4922-af60-757fc8ed48c6
```

Note: it is recommended to use the `mdevctl` tool for actually configuring the host device.

To define the same device as configured below to be started automatically, use

```
[root@host ~]# driverctl -b css set-override 0.0.0313 vfio_ccw
[root@host ~]# mdevctl define -u 7e270a25-e163-4922-af60-757fc8ed48c6 \
               -p 0.0.0313 -t vfio_ccw-io -a
```

If using `mdevctl` is not possible or wanted, follow the manual procedure below.

- Locate the subchannel for the device (in this example, `0.0.2b09`):

  ```
  [root@host ~]# lscss | grep 0.0.2b09 | awk '{print $2}'
  0.0.0313
  ```

- Unbind the subchannel (in this example, `0.0.0313`) from the standard I/O subchannel driver and bind it to the vfio-ccw driver:

  ```
  [root@host ~]# echo 0.0.0313 > /sys/bus/css/devices/0.0.0313/driver/unbind
  [root@host ~]# echo 0.0.0313 > /sys/bus/css/drivers/vfio_ccw/bind
  ```

- Create the mediated device (identified by the uuid):

```
[root@host ~]# echo "7e270a25-e163-4922-af60-757fc8ed48c6" > \
/sys/bus/css/devices/0.0.0313/mdev_supported_types/vfio_ccw-io/create
```

Step 2: configure QEMU

- Reference the created mediated device and (optionally) pick a device id to be presented in the guest (here, `fe.0.1234`, which will end up visible in the guest as `0.0.1234`:

```
-device vfio-ccw,devno=fe.0.1234,sysfsdev=\
/sys/bus/mdev/devices/7e270a25-e163-4922-af60-757fc8ed48c6
```

- Start the guest. The device (here, `0.0.1234`) should now be usable:

```
[root@guest ~]# lscss -d 0.0.1234
Device   Subchan.  DevType CU Type Use  PIM PAM POM  CHPID
----------------------------------------------------------------------
0.0.1234 0.0.0007  3390/0e 3990/e9      f0  f0  ff   1a2a3a0a 00000000
[root@guest ~]# chccwdev -e 0.0.1234
Setting device 0.0.1234 online
Done
[root@guest ~]# dmesg -t
(...)
dasd-eckd 0.0.1234: A channel path to the device has become operational
dasd-eckd 0.0.1234: New DASD 3390/0E (CU 3990/01) with 10017 cylinders, 15 heads,␣
↪224 sectors
dasd-eckd 0.0.1234: DASD with 4 KB/block, 7212240 KB total size, 48 KB/track,␣
↪compatible disk layout
dasda:VOL1/  0X2B09: dasda1
```

Architectural features

Boot devices on s390x

Booting with bootindex parameter

For classical mainframe guests (i.e. LPAR or z/VM installations), you always have to explicitly specify the disk where you want to boot from (or "IPL" from, in s390x-speak – IPL means "Initial Program Load"). In particular, there can also be only one boot device according to the architecture specification, thus specifying multiple boot devices is not possible (yet).

So for booting an s390x guest in QEMU, you should always mark the device where you want to boot from with the `bootindex` property, for example:

```
qemu-system-s390x -drive if=none,id=dr1,file=guest.qcow2 \
                  -device virtio-blk,drive=dr1,bootindex=1
```

For booting from a CD-ROM ISO image (which needs to include El-Torito boot information in order to be bootable), it is recommended to specify a `scsi-cd` device, for example like this:

```
qemu-system-s390x -blockdev file,node-name=c1,filename=... \
                  -device virtio-scsi \
                  -device scsi-cd,drive=c1,bootindex=1
```

Note that you really have to use the `bootindex` property to select the boot device. The old-fashioned `-boot order=...` command of QEMU (and also `-boot once=...`) is not supported on s390x.

Booting without bootindex parameter

The QEMU guest firmware (the so-called s390-ccw bios) has also some rudimentary support for scanning through the available block devices. So in case you did not specify a boot device with the `bootindex` property, there is still a chance that it finds a bootable device on its own and starts a guest operating system from it. However, this scanning algorithm is still very rough and may be incomplete, so that it might fail to detect a bootable device in many cases. It is really recommended to always specify the boot device with the `bootindex` property instead.

This also means that you should avoid the classical short-cut commands like `-hda`, `-cdrom` or `-drive if=virtio`, since it is not possible to specify the `bootindex` with these commands. Note that the convenience `-cdrom` option even does not give you a real (virtio-scsi) CD-ROM device on s390x. Due to technical limitations in the QEMU code base, you will get a virtio-blk device with this parameter instead, which might not be the right device type for installing a Linux distribution via ISO image. It is recommended to specify a CD-ROM device via `-device scsi-cd` (as mentioned above) instead.

Booting from a network device

Beside the normal guest firmware (which is loaded from the file `s390-ccw.img` in the data directory of QEMU, or via the `-bios` option), QEMU ships with a small TFTP network bootloader firmware for virtio-net-ccw devices, too. This firmware is loaded from a file called `s390-netboot.img` in the QEMU data directory. In case you want to load it from a different filename instead, you can specify it via the `-global s390-ipl.netboot_fw=filename` command line option.

The `bootindex` property is especially important for booting via the network. If you don't specify the the `bootindex` property here, the network bootloader firmware code won't get loaded into the guest memory so that the network boot will fail. For a successful network boot, try something like this:

```
qemu-system-s390x -netdev user,id=n1,tftp=...,bootfile=... \
                  -device virtio-net-ccw,netdev=n1,bootindex=1
```

The network bootloader firmware also has basic support for pxelinux.cfg-style configuration files. See the PXELINUX Configuration page for details how to set up the configuration file on your TFTP server. The supported configuration file entries are `DEFAULT`, `LABEL`, `KERNEL`, `INITRD` and `APPEND` (see the Syslinux Config file syntax for more information).

Protected Virtualization on s390x

The memory and most of the registers of Protected Virtual Machines (PVMs) are encrypted or inaccessible to the hypervisor, effectively prohibiting VM introspection when the VM is running. At rest, PVMs are encrypted and can only be decrypted by the firmware, represented by an entity called Ultravisor, of specific IBM Z machines.

Prerequisites

To run PVMs, a machine with the Protected Virtualization feature, as indicated by the Ultravisor Call facility (stfle bit 158), is required. The Ultravisor needs to be initialized at boot by setting *prot_virt=1* on the host's kernel command line.

Running PVMs requires using the KVM hypervisor.

If those requirements are met, the capability *KVM_CAP_S390_PROTECTED* will indicate that KVM can support PVMs on that LPAR.

QEMU Settings

To indicate to the VM that it can transition into protected mode, the *Unpack facility* (stfle bit 161 represented by the feature *unpack*/*S390_FEAT_UNPACK*) needs to be part of the cpu model of the VM.

All I/O devices need to use the IOMMU. Passthrough (vfio) devices are currently not supported.

Host huge page backings are not supported. However guests can use huge pages as indicated by its facilities.

Boot Process

A secure guest image can either be loaded from disk or supplied on the QEMU command line. Booting from disk is done by the unmodified s390-ccw BIOS. I.e., the bootmap is interpreted, multiple components are read into memory and control is transferred to one of the components (zipl stage3). Stage3 does some fixups and then transfers control to some program residing in guest memory, which is normally the OS kernel. The secure image has another component prepended (stage3a) that uses the new diag308 subcodes 8 and 10 to trigger the transition into secure mode.

Booting from the image supplied on the QEMU command line requires that the file passed via -kernel has the same memory layout as would result from the disk boot. This memory layout includes the encrypted components (kernel, initrd, cmdline), the stage3a loader and metadata. In case this boot method is used, the command line options -initrd and -cmdline are ineffective. The preparation of a PVM image is done via the *genprotimg* tool from the s390-tools collection.

1.15.10 RX System emulator

Use the executable `qemu-system-rx` to simulate RX target (GDB simulator). This target emulated following devices.

- R5F562N8 MCU
  - On-chip memory (ROM 512KB, RAM 96KB)
  - Interrupt Control Unit (ICUa)
  - 8Bit Timer x 1CH (TMR0,1)
  - Compare Match Timer x 2CH (CMT0,1)
  - Serial Communication Interface x 1CH (SCI0)
- External memory 16MByte

Example of `qemu-system-rx` usage for RX is shown below:

Download `<u-boot_image_file>` from https://osdn.net/users/ysato/pf/qemu/dl/u-boot.bin.gz

Start emulation of rx-virt:: qemu-system-rx -M gdbsim-r5f562n8 -bios <u-boot_image_file>

Download `kernel_image_file` from https://osdn.net/users/ysato/pf/qemu/dl/zImage

Download `device_tree_blob` from https://osdn.net/users/ysato/pf/qemu/dl/rx-virt.dtb

Start emulation of rx-virt::

qemu-system-rx -M gdbsim-r5f562n8 -kernel <kernel_image_file> -dtb <device_tree_blob> -append “earlycon”

1.15.11 AVR System emulator

Use the executable `qemu-system-avr` to emulate a AVR 8 bit based machine. These can have one of the following cores: avr1, avr2, avr25, avr3, avr31, avr35, avr4, avr5, avr51, avr6, avrtiny, xmega2, xmega3, xmega4, xmega5, xmega6 and xmega7.

As for now it supports few Arduino boards for educational and testing purposes. These boards use a ATmega controller, which model is limited to USART & 16-bit timer devices, enought to run FreeRTOS based applications (like https://github.com/seharris/qemu-avr-tests/blob/master/free-rtos/Demo/AVR_ATMega2560_GCC/demo.elf).

Following are examples of possible usages, assuming demo.elf is compiled for AVR cpu

- Continuous non interrupted execution:

  ```
  qemu-system-avr -machine mega2560 -bios demo.elf
  ```

- Continuous non interrupted execution with serial output into telnet window:

  ```
  qemu-system-avr -M mega2560 -bios demo.elf -nographic \
                  -serial tcp::5678,server,nowait
  ```

  and then in another shell:

  ```
  telnet localhost 5678
  ```

- Debugging wit GDB debugger:

  ```
  qemu-system-avr -machine mega2560 -bios demo.elf -s -S
  ```

  and then in another shell:

  ```
  avr-gdb demo.elf
  ```

  and then within GDB shell:

  ```
  target remote :1234
  ```

- Print out executed instructions (that have not been translated by the JIT compiler yet):

  ```
  qemu-system-avr -machine mega2560 -bios demo.elf -d in_asm
  ```

1.16 Security

1.16.1 Overview

This chapter explains the security requirements that QEMU is designed to meet and principles for securely deploying QEMU.

1.16.2 Security Requirements

QEMU supports many different use cases, some of which have stricter security requirements than others. The community has agreed on the overall security requirements that users may depend on. These requirements define what is considered supported from a security perspective.

Virtualization Use Case

The virtualization use case covers cloud and virtual private server (VPS) hosting, as well as traditional data center and desktop virtualization. These use cases rely on hardware virtualization extensions to execute guest code safely on the physical CPU at close-to-native speed.

The following entities are untrusted, meaning that they may be buggy or malicious:

- Guest
- User-facing interfaces (e.g. VNC, SPICE, WebSocket)
- Network protocols (e.g. NBD, live migration)
- User-supplied files (e.g. disk images, kernels, device trees)
- Passthrough devices (e.g. PCI, USB)

Bugs affecting these entities are evaluated on whether they can cause damage in real-world use cases and treated as security bugs if this is the case.

Non-virtualization Use Case

The non-virtualization use case covers emulation using the Tiny Code Generator (TCG). In principle the TCG and device emulation code used in conjunction with the non-virtualization use case should meet the same security requirements as the virtualization use case. However, for historical reasons much of the non-virtualization use case code was not written with these security requirements in mind.

Bugs affecting the non-virtualization use case are not considered security bugs at this time. Users with non-virtualization use cases must not rely on QEMU to provide guest isolation or any security guarantees.

1.16.3 Architecture

This section describes the design principles that ensure the security requirements are met.

Guest Isolation

Guest isolation is the confinement of guest code to the virtual machine. When guest code gains control of execution on the host this is called escaping the virtual machine. Isolation also includes resource limits such as throttling of CPU, memory, disk, or network. Guests must be unable to exceed their resource limits.

QEMU presents an attack surface to the guest in the form of emulated devices. The guest must not be able to gain control of QEMU. Bugs in emulated devices could allow malicious guests to gain code execution in QEMU. At this point the guest has escaped the virtual machine and is able to act in the context of the QEMU process on the host.

Guests often interact with other guests and share resources with them. A malicious guest must not gain control of other guests or access their data. Disk image files and network traffic must be protected from other guests unless explicitly shared between them by the user.

Principle of Least Privilege

The principle of least privilege states that each component only has access to the privileges necessary for its function. In the case of QEMU this means that each process only has access to resources belonging to the guest.

The QEMU process should not have access to any resources that are inaccessible to the guest. This way the guest does not gain anything by escaping into the QEMU process since it already has access to those same resources from within the guest.

Following the principle of least privilege immediately fulfills guest isolation requirements. For example, guest A only has access to its own disk image file `a.img` and not guest B's disk image file `b.img`.

In reality certain resources are inaccessible to the guest but must be available to QEMU to perform its function. For example, host system calls are necessary for QEMU but are not exposed to guests. A guest that escapes into the QEMU process can then begin invoking host system calls.

New features must be designed to follow the principle of least privilege. Should this not be possible for technical reasons, the security risk must be clearly documented so users are aware of the trade-off of enabling the feature.

Isolation mechanisms

Several isolation mechanisms are available to realize this architecture of guest isolation and the principle of least privilege. With the exception of Linux seccomp, these mechanisms are all deployed by management tools that launch QEMU, such as libvirt. They are also platform-specific so they are only described briefly for Linux here.

The fundamental isolation mechanism is that QEMU processes must run as unprivileged users. Sometimes it seems more convenient to launch QEMU as root to give it access to host devices (e.g. `/dev/net/tun`) but this poses a huge security risk. File descriptor passing can be used to give an otherwise unprivileged QEMU process access to host devices without running QEMU as root. It is also possible to launch QEMU as a non-root user and configure UNIX groups for access to `/dev/kvm`, `/dev/net/tun`, and other device nodes. Some Linux distros already ship with UNIX groups for these devices by default.

- SELinux and AppArmor make it possible to confine processes beyond the traditional UNIX process and file permissions model. They restrict the QEMU process from accessing processes and files on the host system that are not needed by QEMU.
- Resource limits and cgroup controllers provide throughput and utilization limits on key resources such as CPU time, memory, and I/O bandwidth.
- Linux namespaces can be used to make process, file system, and other system resources unavailable to QEMU. A namespaced QEMU process is restricted to only those resources that were granted to it.
- Linux seccomp is available via the QEMU `--sandbox` option. It disables system calls that are not needed by QEMU, thereby reducing the host kernel attack surface.

1.16.4 Sensitive configurations

There are aspects of QEMU that can have security implications which users & management applications must be aware of.

Monitor console (QMP and HMP)

The monitor console (whether used with QMP or HMP) provides an interface to dynamically control many aspects of QEMU's runtime operation. Many of the commands exposed will instruct QEMU to access content on the host file system and/or trigger spawning of external processes.

For example, the `migrate` command allows for the spawning of arbitrary processes for the purpose of tunnelling the migration data stream. The `blockdev-add` command instructs QEMU to open arbitrary files, exposing their content to the guest as a virtual disk.

Unless QEMU is otherwise confined using technologies such as SELinux, AppArmor, or Linux namespaces, the monitor console should be considered to have privileges equivalent to those of the user account QEMU is running under.

It is further important to consider the security of the character device backend over which the monitor console is exposed. It needs to have protection against malicious third parties which might try to make unauthorized connections, or perform man-in-the-middle attacks. Many of the character device backends do not satisfy this requirement and so must not be used for the monitor console.

The general recommendation is that the monitor console should be exposed over a UNIX domain socket backend to the local host only. Use of the TCP based character device backend is inappropriate unless configured to use both TLS encryption and authorization control policy on client connections.

In summary, the monitor console is considered a privileged control interface to QEMU and as such should only be made accessible to a trusted management application or user.

1.17 Deprecated features

In general features are intended to be supported indefinitely once introduced into QEMU. In the event that a feature needs to be removed, it will be listed in this section. The feature will remain functional for 2 releases prior to actual removal. Deprecated features may also generate warnings on the console when QEMU starts up, or if activated via a monitor command, however, this is not a mandatory requirement.

Prior to the 2.10.0 release there was no official policy on how long features would be deprecated prior to their removal, nor any documented list of which features were deprecated. Thus any features deprecated prior to 2.10.0 will be treated as if they were first deprecated in the 2.10.0 release.

What follows is a list of all features currently marked as deprecated.

1.17.1 System emulator command line arguments

`-machine enforce-config-section=on|off` (since 3.1)

The `enforce-config-section` parameter is replaced by the `-global migration.send-configuration={on|off}` option.

`-no-kvm` (since 1.3.0)

The `-no-kvm` argument is now a synonym for setting `-accel tcg`.

`-usbdevice` (since 2.10.0)

The `-usbdevice DEV` argument is now a synonym for setting the `-device usb-DEV` argument instead. The deprecated syntax would automatically enable USB support on the machine type. If using the new syntax, USB support must be explicitly enabled via the `-machine usb=on` argument.

`-drive file=json:{...{'driver':'file'}}` (since 3.0)

The 'file' driver for drives is no longer appropriate for character or host devices and will only accept regular files (S_IFREG). The correct driver for these file types is 'host_cdrom' or 'host_device' as appropriate.

`-smp` (invalid topologies) (since 3.1)

CPU topology properties should describe whole machine topology including possible CPUs.

However, historically it was possible to start QEMU with an incorrect topology where $n <= sockets * cores * threads < maxcpus$, which could lead to an incorrect topology enumeration by the guest. Support for invalid topologies will be removed, the user must ensure topologies described with -smp include all possible cpus, i.e. *sockets* * *cores* * *threads* = *maxcpus*.

`-vnc acl` (since 4.0.0)

The `acl` option to the `-vnc` argument has been replaced by the `tls-authz` and `sasl-authz` options.

`QEMU_AUDIO_` environment variables and `-audio-help` (since 4.0)

The `-audiodev` argument is now the preferred way to specify audio backend settings instead of environment variables. To ease migration to the new format, the `-audiodev-help` option can be used to convert the current values of the environment variables to `-audiodev` options.

Creating sound card devices and vnc without `audiodev=` property (since 4.2)

When not using the deprecated legacy audio config, each sound card should specify an `audiodev=` property. Additionally, when using vnc, you should specify an `audiodev=` propery if you plan to transmit audio through the VNC protocol.

Creating sound card devices using `-soundhw` (since 5.1)

Sound card devices should be created using `-device` instead. The names are the same for most devices. The exceptions are `hda` which needs two devices (`-device intel-hda -device hda-duplex`) and `pcspk` which can be activated using `-machine pcspk-audiodev=<name>`.

`-mon ...,control=readline,pretty=on|off` (since 4.1)

The `pretty=on|off` switch has no effect for HMP monitors, but is silently ignored. Using the switch with HMP monitors will become an error in the future.

`-realtime` (since 4.1)

The `-realtime mlock=on|off` argument has been replaced by the `-overcommit mem-lock=on|off` argument.

`-numa` node (without memory specified) (since 4.1)

Splitting RAM by default between NUMA nodes has the same issues as `mem` parameter described above with the difference that the role of the user plays QEMU using implicit generic or board specific splitting rule. Use `memdev` with *memory-backend-ram* backend or `mem` (if it's supported by used machine type) to define mapping explictly instead.

`-mem-path` fallback to RAM (since 4.1)

Currently if guest RAM allocation from file pointed by `mem-path` fails, QEMU falls back to allocating from RAM, which might result in unpredictable behavior since the backing file specified by the user is ignored. In the future, users will be responsible for making sure the backing storage specified with `-mem-path` can actually provide the guest RAM configured with `-m` and QEMU will fail to start up if RAM allocation is unsuccessful.

RISC-V `-bios` (since 5.1)

QEMU 4.1 introduced support for the -bios option in QEMU for RISC-V for the RISC-V virt machine and sifive_u machine. QEMU 4.1 had no changes to the default behaviour to avoid breakages.

QEMU 5.1 changes the default behaviour from `-bios none` to `-bios default`.

QEMU 5.1 has three options:

1. **`-bios default` - This is the current default behavior if no -bios option** is included. This option will load the default OpenSBI firmware automatically. The firmware is included with the QEMU release and no user interaction is required. All a user needs to do is specify the kernel they want to boot with the -kernel option
2. **`-bios none` - QEMU will not automatically load any firmware. It is up** to the user to load all the images they need.
3. `-bios <file>` - Tells QEMU to load the specified file as the firmwrae.

`-tb-size` option (since 5.0)

QEMU 5.0 introduced an alternative syntax to specify the size of the translation block cache, `-accel tcg,tb-size=`. The new syntax deprecates the previously available `-tb-size` option.

`-show-cursor` option (since 5.0)

Use `-display sdl,show-cursor=on` or `-display gtk,show-cursor=on` instead.

`Configuring floppies with ``-global`

Use `-device floppy,...` instead:

```
-global isa-fdc.driveA=...
-global sysbus-fdc.driveA=...
-global SUNW,fdtwo.drive=...
```

become

```
-device floppy,unit=0,drive=...
```

and

```
-global isa-fdc.driveB=...
-global sysbus-fdc.driveB=...
```

become

```
-device floppy,unit=1,drive=...
```

`-drive` with bogus interface type

Drives with interface types other than `if=none` are for onboard devices. It is possible to use drives the board doesn't pick up with -device. This usage is now deprecated. Use `if=none` instead.

1.17.2 QEMU Machine Protocol (QMP) commands

`change` (since 2.5.0)

Use `blockdev-change-medium` or `change-vnc-password` instead.

`blockdev-open-tray`, `blockdev-close-tray` argument `device` (since 2.8.0)

Use argument `id` instead.

`eject` argument `device` (since 2.8.0)

Use argument `id` instead.

`blockdev-change-medium` argument `device` (since 2.8.0)

Use argument `id` instead.

`block_set_io_throttle` argument `device` (since 2.8.0)

Use argument `id` instead.

`migrate_set_downtime` and `migrate_set_speed` (since 2.8.0)

Use `migrate-set-parameters` instead.

`query-named-block-nodes` result `encryption_key_missing` (since 2.10.0)

Always false.

`query-block` result `inserted.encryption_key_missing` (since 2.10.0)

Always false.

`blockdev-add` empty string argument `backing` (since 2.10.0)

Use argument value `null` instead.

`migrate-set-cache-size` and `query-migrate-cache-size` (since 2.11.0)

Use `migrate-set-parameters` and `query-migrate-parameters` instead.

`block-commit` arguments `base` and `top` (since 3.1.0)

Use arguments `base-node` and `top-node` instead.

`object-add` option `props` (since 5.0)

Specify the properties for the object as top-level arguments instead.

`query-named-block-nodes` and `query-block` result dirty-bitmaps[i].status (since 4.0)

The `status` field of the `BlockDirtyInfo` structure, returned by these commands is deprecated. Two new boolean fields, `recording` and `busy` effectively replace it.

`query-block` result field `dirty-bitmaps` (Since 4.2)

The `dirty-bitmaps` field of the `BlockInfo` structure, returned by the query-block command is itself now deprecated. The `dirty-bitmaps` field of the `BlockDeviceInfo` struct should be used instead, which is the type of the `inserted` field in query-block replies, as well as the type of array items in query-named-block-nodes.

Since the `dirty-bitmaps` field is optionally present in both the old and new locations, clients must use introspection to learn where to anticipate the field if/when it does appear in command output.

`query-cpus` (since 2.12.0)

The `query-cpus` command is replaced by the `query-cpus-fast` command.

`query-cpus-fast` `arch` output member (since 3.0.0)

The `arch` output member of the `query-cpus-fast` command is replaced by the `target` output member.

`cpu-add` (since 4.0)

Use `device_add` for hotplugging vCPUs instead of `cpu-add`. See documentation of `query-hotpluggable-cpus` for additional details.

`query-events` (since 4.0)

The `query-events` command has been superseded by the more powerful and accurate `query-qmp-schema` command.

chardev client socket with `wait` option (since 4.0)

Character devices creating sockets in client mode should not specify the ‘wait’ field, which is only applicable to sockets in server mode

1.17.3 Human Monitor Protocol (HMP) commands

`cpu-add` (since 4.0)

Use `device_add` for hotplugging vCPUs instead of `cpu-add`. See documentation of `query-hotpluggable-cpus` for additional details.

`acl_show`, `acl_reset`, `acl_policy`, `acl_add`, `acl_remove` (since 4.0.0)

The `acl_show`, `acl_reset`, `acl_policy`, `acl_add`, and `acl_remove` commands are deprecated with no replacement. Authorization for VNC should be performed using the pluggable QAuthZ objects.

1.17.4 System emulator CPUS

`compat` property of server class POWER CPUs (since 5.0)

The `compat` property used to set backwards compatibility modes for the processor has been deprecated. The `max-cpu-compat` property of the `pseries` machine type should be used instead.

KVM guest support on 32-bit Arm hosts (since 5.0)

The Linux kernel has dropped support for allowing 32-bit Arm systems to host KVM guests as of the 5.7 kernel. Accordingly, QEMU is deprecating its support for this configuration and will remove it in a future version. Running 32-bit guests on a 64-bit Arm host remains supported.

1.17.5 System emulator devices

`ide-drive` (since 4.2)

The ‘ide-drive’ device is deprecated. Users should use ‘ide-hd’ or ‘ide-cd’ as appropriate to get an IDE hard disk or CD-ROM as needed.

`scsi-disk` (since 4.2)

The ‘scsi-disk’ device is deprecated. Users should use ‘scsi-hd’ or ‘scsi-cd’ as appropriate to get a SCSI hard disk or CD-ROM as needed.

1.17.6 System emulator machines

mips `r4k` platform (since 5.0)

This machine type is very old and unmaintained. Users should use the `malta` machine type instead.

mips `fulong2e` machine (since 5.1)

This machine has been renamed `fuloong2e`.

`pc-1.0`, `pc-1.1`, `pc-1.2` and `pc-1.3` (since 5.0)

These machine types are very old and likely can not be used for live migration from old QEMU versions anymore. A newer machine type should be used instead.

1.17.7 Device options

Emulated device options

`-device virtio-blk,scsi=on|off` (since 5.0.0)

The virtio-blk SCSI passthrough feature is a legacy VIRTIO feature. VIRTIO 1.0 and later do not support it because the virtio-scsi device was introduced for full SCSI support. Use virtio-scsi instead when SCSI passthrough is required.

Note this also applies to `-device virtio-blk-pci,scsi=on|off`, which is an alias.

Block device options

`"backing": ""` (since 2.12.0)

In order to prevent QEMU from automatically opening an image's backing chain, use `"backing": null` instead.

`rbd` keyvalue pair encoded filenames: `""` (since 3.1.0)

Options for `rbd` should be specified according to its runtime options, like other block drivers. Legacy parsing of keyvalue pair encoded filenames is useful to open images with the old format for backing files; These image files should be updated to use the current format.

Example of legacy encoding:

```
json:{"file.driver":"rbd", "file.filename":"rbd:rbd/name"}
```

The above, converted to the current supported format:

```
json:{"file.driver":"rbd", "file.pool":"rbd", "file.image":"name"}
```

1.17.8 linux-user mode CPUs

`tilegx` CPUs (since 5.1.0)

The `tilegx` guest CPU support (which was only implemented in linux-user mode) is deprecated and will be removed in a future version of QEMU. Support for this CPU was removed from the upstream Linux kernel in 2018, and has also been dropped from glibc.

1.17.9 Related binaries

qemu-img amend to adjust backing file (since 5.1)

The use of `qemu-img amend` to modify the name or format of a qcow2 backing image is deprecated; this functionality was never fully documented or tested, and interferes with other amend operations that need access to the original backing image (such as deciding whether a v3 zero cluster may be left unallocated when converting to a v2 image). Rather, any changes to the backing chain should be performed with `qemu-img rebase -u` either before or after the remaining changes being performed by amend, as appropriate.

qemu-img backing file without format (since 5.1)

The use of `qemu-img create`, `qemu-img rebase`, or `qemu-img convert` to create or modify an image that depends on a backing file now recommends that an explicit backing format be provided. This is for safety: if QEMU probes a different format than what you thought, the data presented to the guest will be corrupt; similarly, presenting a raw image to a guest allows a potential security exploit if a future probe sees a non-raw image based on guest writes.

To avoid the warning message, or even future refusal to create an unsafe image, you must pass `-o backing_fmt=` (or the shorthand `-F` during create) to specify the intended backing format. You may use `qemu-img rebase -u` to retroactively add a backing format to an existing image. However, be aware that there are already potential security risks to blindly using `qemu-img info` to probe the format of an untrusted backing image, when deciding what format to add into an existing image.

1.17.10 Backwards compatibility

Runnability guarantee of CPU models (since 4.1.0)

Previous versions of QEMU never changed existing CPU models in ways that introduced additional host software or hardware requirements to the VM. This allowed management software to safely change the machine type of an existing VM without introducing new requirements ("runnability guarantee"). This prevented CPU models from being updated to include CPU vulnerability mitigations, leaving guests vulnerable in the default configuration.

The CPU model runnability guarantee won't apply anymore to existing CPU models. Management software that needs runnability guarantees must resolve the CPU model aliases using te `alias-of` field returned by the `query-cpu-definitions` QMP command.

While those guarantees are kept, the return value of `query-cpu-definitions` will have existing CPU model aliases point to a version that doesn't break runnability guarantees (specifically, version 1 of those CPU models). In future QEMU versions, aliases will point to newer CPU model versions depending on the machine type, so management software must resolve CPU model aliases before starting a virtual machine.

1.18 Recently removed features

What follows is a record of recently removed, formerly deprecated features that serves as a record for users who have encountered trouble after a recent upgrade.

1.18.1 System emulator command line arguments

`-net ...,name=`*name* (removed in 5.1)

The `name` parameter of the `-net` option was a synonym for the `id` parameter, which should now be used instead.

1.18.2 QEMU Machine Protocol (QMP) commands

`block-dirty-bitmap-add` "autoload" parameter (since 4.2.0)

The "autoload" parameter has been ignored since 2.12.0. All bitmaps are automatically loaded from qcow2 images.

1.18.3 Human Monitor Protocol (HMP) commands

The `hub_id` parameter of `hostfwd_add` / `hostfwd_remove` (removed in 5.0)

The `[hub_id name]` parameter tuple of the 'hostfwd_add' and 'hostfwd_remove' HMP commands has been replaced by `netdev_id`.

1.18.4 Guest Emulator ISAs

RISC-V ISA privledge specification version 1.09.1 (removed in 5.1)

The RISC-V ISA privledge specification version 1.09.1 has been removed. QEMU supports both the newer version 1.10.0 and the ratified version 1.11.0, these should be used instead of the 1.09.1 version.

1.18.5 System emulator CPUS

RISC-V ISA Specific CPUs (removed in 5.1)

The RISC-V cpus with the ISA version in the CPU name have been removed. The four CPUs are: `rv32gcsu-v1.9.1`, `rv32gcsu-v1.10.0`, `rv64gcsu-v1.9.1` and `rv64gcsu-v1.10.0`. Instead the version can be specified via the CPU `priv_spec` option when using the `rv32` or `rv64` CPUs.

RISC-V no MMU CPUs (removed in 5.1)

The RISC-V no MMU cpus have been removed. The two CPUs: `rv32imacu-nommu` and `rv64imacu-nommu` can no longer be used. Instead the MMU status can be specified via the CPU `mmu` option when using the `rv32` or `rv64` CPUs.

1.18.6 System emulator machines

`spike_v1.9.1` and `spike_v1.10` (removed in 5.1)

The version specific Spike machines have been removed in favour of the generic `spike` machine. If you need to specify an older version of the RISC-V spec you can use the `-cpu rv64gcsu,priv_spec=v1.10.0` command line argument.

1.18.7 Related binaries

`qemu-nbd --partition` (removed in 5.0)

The `qemu-nbd --partition $digit` code (also spelled `-P`) could only handle MBR partitions, and never correctly handled logical partitions beyond partition 5. Exporting a partition can still be done by utilizing the `--image-opts` option with a raw blockdev using the `offset` and `size` parameters layered on top of any other existing blockdev. For example, if partition 1 is 100MiB long starting at 1MiB, the old command:

```
qemu-nbd -t -P 1 -f qcow2 file.qcow2
```

can be rewritten as:

```
qemu-nbd -t --image-opts driver=raw,offset=1M,size=100M,file.driver=qcow2,file.file.
↪driver=file,file.file.filename=file.qcow2
```

`qemu-img convert -n -o` (removed in 5.1)

All options specified in `-o` are image creation options, so they are now rejected when used with `-n` to skip image creation.

`qemu-img create -b bad file $size` (removed in 5.1)

When creating an image with a backing file that could not be opened, `qemu-img create` used to issue a warning about the failure but proceed with the image creation if an explicit size was provided. However, as the `-u` option exists for this purpose, it is safer to enforce that any failure to open the backing image (including if the backing file is missing or an incorrect format was specified) is an error when `-u` is not used.

1.18.8 Command line options

`-numa node,mem=`*size* (removed in 5.1)

The parameter `mem` of `-numa node` was used to assign a part of guest RAM to a NUMA node. But when using it, it's impossible to manage a specified RAM chunk on the host side (like bind it to a host node, setting bind policy, ...), so the guest ends up with the fake NUMA configuration with suboptiomal performance. However since 2014 there is an alternative way to assign RAM to a NUMA node using parameter `memdev`, which does the same as `mem` and adds means to actually manage node RAM on the host side. Use parameter `memdev` with *memory-backend-ram* backend as replacement for parameter `mem` to achieve the same fake NUMA effect or a properly configured *memory-backend-file* backend to actually benefit from NUMA configuration. New machine versions (since 5.1) will not accept the option but it will still work with old machine types. User can check the QAPI schema to see if the legacy option is supported by looking at MachineInfo::numa-mem-supported property.

1.18.9 Block devices

VXHS backend (removed in 5.1)

The VXHS code does not compile since v2.12.0. It was removed in 5.1.

1.19 Supported build platforms

QEMU aims to support building and executing on multiple host OS platforms. This appendix outlines which platforms are the major build targets. These platforms are used as the basis for deciding upon the minimum required versions of 3rd party software QEMU depends on. The supported platforms are the targets for automated testing performed by the project when patches are submitted for review, and tested before and after merge.

If a platform is not listed here, it does not imply that QEMU won't work. If an unlisted platform has comparable software versions to a listed platform, there is every expectation that it will work. Bug reports are welcome for problems encountered on unlisted platforms unless they are clearly older vintage than what is described here.

Note that when considering software versions shipped in distros as support targets, QEMU considers only the version number, and assumes the features in that distro match the upstream release with the same version. In other words, if a distro backports extra features to the software in their distro, QEMU upstream code will not add explicit support for those backports, unless the feature is auto-detectable in a manner that works for the upstream releases too.

The Repology site https://repology.org is a useful resource to identify currently shipped versions of software in various operating systems, though it does not cover all distros listed below.

1.19.1 Linux OS

For distributions with frequent, short-lifetime releases, the project will aim to support all versions that are not end of life by their respective vendors. For the purposes of identifying supported software versions, the project will look at Fedora, Ubuntu, and openSUSE distros. Other short- lifetime distros will be assumed to ship similar software versions.

For distributions with long-lifetime releases, the project will aim to support the most recent major version at all times. Support for the previous major version will be dropped 2 years after the new major version is released, or when it reaches "end of life". For the purposes of identifying supported software versions, the project will look at RHEL, Debian, Ubuntu LTS, and SLES distros. Other long-lifetime distros will be assumed to ship similar software versions.

1.19.2 Windows

The project supports building with current versions of the MinGW toolchain, hosted on Linux.

1.19.3 macOS

The project supports building with the two most recent versions of macOS, with the current Homebrew package set available.

1.19.4 FreeBSD

The project aims to support all versions which are not end of life.

1.19.5 NetBSD

The project aims to support the most recent major version at all times. Support for the previous major version will be dropped 2 years after the new major version is released.

1.19.6 OpenBSD

The project aims to support all versions which are not end of life.

1.20 License

QEMU is a trademark of Fabrice Bellard.

QEMU is released under the GNU General Public License, version 2. Parts of QEMU have specific licenses, see file LICENSE.

CHAPTER 2

QEMU User Mode Emulation User's Guide

This manual is the overall guide for users using QEMU for user-mode emulation. In this mode, QEMU can launch processes compiled for one CPU on another CPU.

Contents:

2.1 QEMU User space emulator

2.1.1 Supported Operating Systems

The following OS are supported in user space emulation:

- Linux (referred as qemu-linux-user)
- BSD (referred as qemu-bsd-user)

2.1.2 Features

QEMU user space emulation has the following notable features:

System call translation: QEMU includes a generic system call translator. This means that the parameters of the system calls can be converted to fix endianness and 32/64-bit mismatches between hosts and targets. IOCTLs can be converted too.

POSIX signal handling: QEMU can redirect to the running program all signals coming from the host (such as `SIGALRM`), as well as synthesize signals from virtual CPU exceptions (for example `SIGFPE` when the program executes a division by zero).

QEMU relies on the host kernel to emulate most signal system calls, for example to emulate the signal mask. On Linux, QEMU supports both normal and real-time signals.

Threading: On Linux, QEMU can emulate the `clone` syscall and create a real host thread (with a separate virtual CPU) for each emulated thread. Note that not all targets currently emulate atomic operations correctly. x86 and Arm use a global lock in order to preserve their semantics.

QEMU was conceived so that ultimately it can emulate itself. Although it is not very useful, it is an important test to show the power of the emulator.

2.1.3 Linux User space emulator

Quick Start

In order to launch a Linux process, QEMU needs the process executable itself and all the target (x86) dynamic libraries used by it.

- On x86, you can just try to launch any process by using the native libraries:

  ```
  qemu-i386 -L / /bin/ls
  ```

  `-L /` tells that the x86 dynamic linker must be searched with a `/` prefix.

- Since QEMU is also a linux process, you can launch QEMU with QEMU (NOTE: you can only do that if you compiled QEMU from the sources):

  ```
  qemu-i386 -L / qemu-i386 -L / /bin/ls
  ```

- On non x86 CPUs, you need first to download at least an x86 glibc (`qemu-runtime-i386-XXX-.tar.gz` on the QEMU web page). Ensure that `LD_LIBRARY_PATH` is not set:

  ```
  unset LD_LIBRARY_PATH
  ```

  Then you can launch the precompiled `ls` x86 executable:

  ```
  qemu-i386 tests/i386/ls
  ```

  You can look at `scripts/qemu-binfmt-conf.sh` so that QEMU is automatically launched by the Linux kernel when you try to launch x86 executables. It requires the `binfmt_misc` module in the Linux kernel.

- The x86 version of QEMU is also included. You can try weird things such as:

  ```
  qemu-i386 /usr/local/qemu-i386/bin/qemu-i386 \
            /usr/local/qemu-i386/bin/ls-i386
  ```

Wine launch

- Ensure that you have a working QEMU with the x86 glibc distribution (see previous section). In order to verify it, you must be able to do:

  ```
  qemu-i386 /usr/local/qemu-i386/bin/ls-i386
  ```

- Download the binary x86 Wine install (`qemu-XXX-i386-wine.tar.gz` on the QEMU web page).
- Configure Wine on your account. Look at the provided script `/usr/local/qemu-i386/bin/wine-conf.sh`. Your previous `${HOME}/.wine` directory is saved to `${HOME}/.wine.org`.
- Then you can try the example `putty.exe`:

  ```
  qemu-i386 /usr/local/qemu-i386/wine/bin/wine \
            /usr/local/qemu-i386/wine/c/Program\ Files/putty.exe
  ```

Command line options

```
qemu-i386 [-h] [-d] [-L path] [-s size] [-cpu model] [-g port] [-B offset] [-R size]␣
↪program [arguments...]
```

-h Print the help

-L path Set the x86 elf interpreter prefix (default=/usr/local/qemu-i386)

-s size Set the x86 stack size in bytes (default=524288)

-cpu model Select CPU model (-cpu help for list and additional feature selection)

-E var=value Set environment var to value.

-U var Remove var from the environment.

-B offset Offset guest address by the specified number of bytes. This is useful when the address region required by guest applications is reserved on the host. This option is currently only supported on some hosts.

-R size Pre-allocate a guest virtual address space of the given size (in bytes). "G", "M", and "k" suffixes may be used when specifying the size.

Debug options:

-d item1,... Activate logging of the specified items (use '-d help' for a list of log items)

-p pagesize Act as if the host page size was 'pagesize' bytes

-g port Wait gdb connection to port

-singlestep Run the emulation in single step mode.

Environment variables:

QEMU_STRACE Print system calls and arguments similar to the 'strace' program (NOTE: the actual 'strace' program will not work because the user space emulator hasn't implemented ptrace). At the moment this is incomplete. All system calls that don't have a specific argument format are printed with information for six arguments. Many flag-style arguments don't have decoders and will show up as numbers.

Other binaries

user mode (Alpha) `qemu-alpha` TODO.

user mode (Arm) `qemu-armeb` TODO.

user mode (Arm) `qemu-arm` is also capable of running Arm "Angel" semihosted ELF binaries (as implemented by the arm-elf and arm-eabi Newlib/GDB configurations), and arm-uclinux bFLT format binaries.

user mode (ColdFire) user mode (M68K) `qemu-m68k` is capable of running semihosted binaries using the BDM (m5xxx-ram-hosted.ld) or m68k-sim (sim.ld) syscall interfaces, and coldfire uClinux bFLT format binaries.

The binary format is detected automatically.

user mode (Cris) `qemu-cris` TODO.

user mode (i386) `qemu-i386` TODO. `qemu-x86_64` TODO.

user mode (Microblaze) `qemu-microblaze` TODO.

user mode (MIPS) `qemu-mips` executes 32-bit big endian MIPS binaries (MIPS O32 ABI).

`qemu-mipsel` executes 32-bit little endian MIPS binaries (MIPS O32 ABI).

`qemu-mips64` executes 64-bit big endian MIPS binaries (MIPS N64 ABI).

`qemu-mips64el` executes 64-bit little endian MIPS binaries (MIPS N64 ABI).

`qemu-mipsn32` executes 32-bit big endian MIPS binaries (MIPS N32 ABI).

`qemu-mipsn32el` executes 32-bit little endian MIPS binaries (MIPS N32 ABI).

user mode (NiosII) `qemu-nios2` TODO.

user mode (PowerPC) `qemu-ppc64abi32` TODO. `qemu-ppc64` TODO. `qemu-ppc` TODO.

user mode (SH4) `qemu-sh4eb` TODO. `qemu-sh4` TODO.

user mode (SPARC) `qemu-sparc` can execute Sparc32 binaries (Sparc32 CPU, 32 bit ABI).

`qemu-sparc32plus` can execute Sparc32 and SPARC32PLUS binaries (Sparc64 CPU, 32 bit ABI).

`qemu-sparc64` can execute some Sparc64 (Sparc64 CPU, 64 bit ABI) and SPARC32PLUS binaries (Sparc64 CPU, 32 bit ABI).

2.1.4 BSD User space emulator

BSD Status

- target Sparc64 on Sparc64: Some trivial programs work.

Quick Start

In order to launch a BSD process, QEMU needs the process executable itself and all the target dynamic libraries used by it.

- On Sparc64, you can just try to launch any process by using the native libraries:

  ```
  qemu-sparc64 /bin/ls
  ```

Command line options

```
qemu-sparc64 [-h] [-d] [-L path] [-s size] [-bsd type] program [arguments...]
```

`-h` Print the help

`-L path` Set the library root path (default=/)

`-s size` Set the stack size in bytes (default=524288)

`-ignore-environment` Start with an empty environment. Without this option, the initial environment is a copy of the caller's environment.

`-E var=value` Set environment var to value.

`-U var` Remove var from the environment.

`-bsd type` Set the type of the emulated BSD Operating system. Valid values are FreeBSD, NetBSD and OpenBSD (default).

Debug options:

`-d item1,...` Activate logging of the specified items (use '-d help' for a list of log items)

`-p pagesize` Act as if the host page size was 'pagesize' bytes

`-singlestep` Run the emulation in single step mode.

CHAPTER 3

QEMU Tools Guide

Contents:

3.1 QEMU disk image utility

3.1.1 Synopsis

qemu-img [*standard options*] *command* [*command options*]

3.1.2 Description

qemu-img allows you to create, convert and modify images offline. It can handle all image formats supported by QEMU.

Warning: Never use qemu-img to modify images in use by a running virtual machine or any other process; this may destroy the image. Also, be aware that querying an image that is being modified by another process may encounter inconsistent state.

3.1.3 Options

Standard options:

`-h, --help`
Display this help and exit

`-V, --version`
Display version information and exit

`-T, --trace` `[[enable=]PATTERN][,events=FILE][,file=FILE]`
Specify tracing options.

[enable=]PATTERN
Immediately enable events matching *PATTERN* (either event name or a globbing pattern). This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend. To specify multiple events or patterns, specify the `-trace` option multiple times.

Use `-trace help` to print a list of names of trace points.

events=FILE
Immediately enable events listed in *FILE*. The file must contain one event name (as listed in the `trace-events-all` file) per line; globbing patterns are accepted too. This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend.

file=FILE
Log output traces to *FILE*. This option is only available if QEMU has been compiled with the `simple` tracing backend.

The following commands are supported:

amend `[--object OBJECTDEF] [--image-opts] [-p] [-q] [-f FMT] [-t CACHE] [--force] -o OPTIO`

bench `[-c COUNT] [-d DEPTH] [-f FMT] [--flush-interval=FLUSH_INTERVAL] [-i AIO] [-n] [--no-`

bitmap `(--merge SOURCE | --add | --remove | --clear | --enable | --disable)... [-b SOURCE_`

check `[--object OBJECTDEF] [--image-opts] [-q] [-f FMT] [--output=OFMT] [-r [leaks | all]]`

commit `[--object OBJECTDEF] [--image-opts] [-q] [-f FMT] [-t CACHE] [-b BASE] [-d] [-p] FI`

compare `[--object OBJECTDEF] [--image-opts] [-f FMT] [-F FMT] [-T SRC_CACHE] [-p] [-q] [-s`

convert `[--object OBJECTDEF] [--image-opts] [--target-image-opts] [--target-is-zero] [--bi`

create `[--object OBJECTDEF] [-q] [-f FMT] [-b BACKING_FILE] [-F BACKING_FMT] [-u] [-o OPTI`

dd `[--image-opts] [-U] [-f FMT] [-O OUTPUT_FMT] [bs=BLOCK_SIZE] [count=BLOCKS] [skip=BLOCK`

info `[--object OBJECTDEF] [--image-opts] [-f FMT] [--output=OFMT] [--backing-chain] [-U] F`

map `[--object OBJECTDEF] [--image-opts] [-f FMT] [--start-offset=OFFSET] [--max-length=LEN`

measure `[--output=OFMT] [-O OUTPUT_FMT] [-o OPTIONS] [--size N | [--object OBJECTDEF] [--i`

snapshot `[--object OBJECTDEF] [--image-opts] [-U] [-q] [-l | -a SNAPSHOT | -c SNAPSHOT | -`

rebase `[--object OBJECTDEF] [--image-opts] [-U] [-q] [-f FMT] [-t CACHE] [-T SRC_CACHE] [-`

resize `[--object OBJECTDEF] [--image-opts] [-f FMT] [--preallocation=PREALLOC] [-q] [--shr`

Command parameters:

FILENAME is a disk image filename.

FMT is the disk image format. It is guessed automatically in most cases. See below for a description of the supported disk formats.

SIZE is the disk image size in bytes. Optional suffixes `k` or `K` (kilobyte, 1024) `M` (megabyte, 1024k) and `G` (gigabyte, 1024M) and T (terabyte, 1024G) are supported. `b` is ignored.

OUTPUT_FILENAME is the destination disk image filename.

OUTPUT_FMT is the destination format.

OPTIONS is a comma separated list of format specific options in a name=value format. Use `-o ?` for an overview of the options supported by the used format or see the format descriptions below for details.

SNAPSHOT_PARAM is param used for internal snapshot, format is ‘snapshot.id=[ID],snapshot.name=[NAME]’ or ‘[ID_OR_NAME]’.

--object OBJECTDEF
: is a QEMU user creatable object definition. See the *qemu(1)* manual page for a description of the object properties. The most common object type is a `secret`, which is used to supply passwords and/or encryption keys.

--image-opts
: Indicates that the source *FILENAME* parameter is to be interpreted as a full option string, not a plain filename. This parameter is mutually exclusive with the *-f* parameter.

--target-image-opts
: Indicates that the OUTPUT_FILENAME parameter(s) are to be interpreted as a full option string, not a plain filename. This parameter is mutually exclusive with the *-O* parameters. It is currently required to also use the *-n* parameter to skip image creation. This restriction may be relaxed in a future release.

--force-share (-U)
: If specified, `qemu-img` will open the image in shared mode, allowing other QEMU processes to open it in write mode. For example, this can be used to get the image information (with 'info' subcommand) when the image is used by a running guest. Note that this could produce inconsistent results because of concurrent metadata changes, etc. This option is only allowed when opening images in read-only mode.

--backing-chain
: Will enumerate information about backing files in a disk image chain. Refer below for further description.

-c
: Indicates that target image must be compressed (qcow format only).

-h
: With or without a command, shows help and lists the supported formats.

-p
: Display progress bar (compare, convert and rebase commands only). If the *-p* option is not used for a command that supports it, the progress is reported when the process receives a `SIGUSR1` or `SIGINFO` signal.

-q
: Quiet mode - do not print any output (except errors). There's no progress bar in case both *-q* and *-p* options are used.

-S SIZE
: Indicates the consecutive number of bytes that must contain only zeros for qemu-img to create a sparse image during conversion. This value is rounded down to the nearest 512 bytes. You may use the common size suffixes like `k` for kilobytes.

-t CACHE
: Specifies the cache mode that should be used with the (destination) file. See the documentation of the emulator's `-drive cache=...` option for allowed values.

-T SRC_CACHE
: Specifies the cache mode that should be used with the source file(s). See the documentation of the emulator's `-drive cache=...` option for allowed values.

Parameters to compare subcommand:

-f
: First image format

-F
: Second image format

-s
: Strict mode - fail on different image size or sector allocation

Parameters to convert subcommand:

`--bitmaps`
Additionally copy all persistent bitmaps from the top layer of the source

`-n`
Skip the creation of the target volume

`-m`
Number of parallel coroutines for the convert process

`-W`
Allow out-of-order writes to the destination. This option improves performance, but is only recommended for preallocated devices like host devices or other raw block devices.

`-C`
Try to use copy offloading to move data from source image to target. This may improve performance if the data is remote, such as with NFS or iSCSI backends, but will not automatically sparsify zero sectors, and may result in a fully allocated target image depending on the host support for getting allocation information.

`--salvage`
Try to ignore I/O errors when reading. Unless in quiet mode (`-q`), errors will still be printed. Areas that cannot be read from the source will be treated as containing only zeroes.

`--target-is-zero`
Assume that reading the destination image will always return zeros. This parameter is mutually exclusive with a destination image that has a backing file. It is required to also use the `-n` parameter to skip image creation.

Parameters to dd subcommand:

`bs``=BLOCK_SIZE`
Defines the block size

`count``=BLOCKS`
Sets the number of input blocks to copy

`if``=INPUT`
Sets the input file

`of``=OUTPUT`
Sets the output file

`skip``=BLOCKS`
Sets the number of input blocks to skip

Parameters to snapshot subcommand:

`snapshot`
Is the name of the snapshot to create, apply or delete

`-a`
Applies a snapshot (revert disk to saved state)

`-c`
Creates a snapshot

`-d`
Deletes a snapshot

`-l`
Lists all snapshots in the given image

Command description:

amend `[--object OBJECTDEF] [--image-opts] [-p] [-q] [-f FMT] [-t CACHE] [--force] -o OPTION`

Amends the image format specific *OPTIONS* for the image file *FILENAME*. Not all file formats support this operation.

The set of options that can be amended are dependent on the image format, but note that amending the backing chain relationship should instead be performed with `qemu-img rebase`.

–force allows some unsafe operations. Currently for -f luks, it allows to erase the last encryption key, and to overwrite an active encryption key.

bench `[-c COUNT] [-d DEPTH] [-f FMT] [--flush-interval=FLUSH_INTERVAL] [-i AIO] [-n] [--no-`

Run a simple sequential I/O benchmark on the specified image. If `-w` is specified, a write test is performed, otherwise a read test is performed.

A total number of *COUNT* I/O requests is performed, each *BUFFER_SIZE* bytes in size, and with *DEPTH* requests in parallel. The first request starts at the position given by *OFFSET*, each following request increases the current position by *STEP_SIZE*. If *STEP_SIZE* is not given, *BUFFER_SIZE* is used for its value.

If *FLUSH_INTERVAL* is specified for a write test, the request queue is drained and a flush is issued before new writes are made whenever the number of remaining requests is a multiple of *FLUSH_INTERVAL*. If additionally `--no-drain` is specified, a flush is issued without draining the request queue first.

if `-i` is specified, *AIO* option can be used to specify different AIO backends: `threads`, `native` or `io_uring`.

If `-n` is specified, the native AIO backend is used if possible. On Linux, this option only works if `-t none` or `-t directsync` is specified as well.

For write tests, by default a buffer filled with zeros is written. This can be overridden with a pattern byte specified by *PATTERN*.

bitmap `(--merge SOURCE | --add | --remove | --clear | --enable | --disable)... [-b SOURCE_`

Perform one or more modifications of the persistent bitmap *BITMAP* in the disk image *FILENAME*. The various modifications are:

`--add` to create *BITMAP*, enabled to record future edits.

`--remove` to remove *BITMAP*.

`--clear` to clear *BITMAP*.

`--enable` to change *BITMAP* to start recording future edits.

`--disable` to change *BITMAP* to stop recording future edits.

`--merge` to merge the contents of the *SOURCE* bitmap into *BITMAP*.

Additional options include `-g` which sets a non-default *GRANULARITY* for `--add`, and `-b` and `-F` which select an alternative source file for all *SOURCE* bitmaps used by `--merge`.

To see what bitmaps are present in an image, use `qemu-img info`.

check `[--object OBJECTDEF] [--image-opts] [-q] [-f FMT] [--output=OFMT] [-r [leaks | all]]`

Perform a consistency check on the disk image *FILENAME*. The command can output in the format *OFMT* which is either `human` or `json`. The JSON output is an object of QAPI type `ImageCheck`.

If `-r` is specified, qemu-img tries to repair any inconsistencies found during the check. `-r leaks` repairs only cluster leaks, whereas `-r all` fixes all kinds of errors, with a higher risk of choosing the wrong fix or hiding corruption that has already occurred.

Only the formats `qcow2`, `qed` and `vdi` support consistency checks.

In case the image does not have any inconsistencies, check exits with `0`. Other exit codes indicate the kind of inconsistency found or if another error occurred. The following table summarizes all exit codes of the check subcommand:

0 Check completed, the image is (now) consistent

1 Check not completed because of internal errors

2 Check completed, image is corrupted

3 Check completed, image has leaked clusters, but is not corrupted

63 Checks are not supported by the image format

If `-r` is specified, exit codes representing the image state refer to the state after (the attempt at) repairing it. That is, a successful `-r all` will yield the exit code 0, independently of the image state before.

`commit` `[--object OBJECTDEF] [--image-opts] [-q] [-f FMT] [-t CACHE] [-b BASE] [-d] [-p] FI`

Commit the changes recorded in *FILENAME* in its base image or backing file. If the backing file is smaller than the snapshot, then the backing file will be resized to be the same size as the snapshot. If the snapshot is smaller than the backing file, the backing file will not be truncated. If you want the backing file to match the size of the smaller snapshot, you can safely truncate it yourself once the commit operation successfully completes.

The image *FILENAME* is emptied after the operation has succeeded. If you do not need *FILENAME* afterwards and intend to drop it, you may skip emptying *FILENAME* by specifying the `-d` flag.

If the backing chain of the given image file *FILENAME* has more than one layer, the backing file into which the changes will be committed may be specified as *BASE* (which has to be part of *FILENAME*'s backing chain). If *BASE* is not specified, the immediate backing file of the top image (which is *FILENAME*) will be used. Note that after a commit operation all images between *BASE* and the top image will be invalid and may return garbage data when read. For this reason, `-b` implies `-d` (so that the top image stays valid).

`compare` `[--object OBJECTDEF] [--image-opts] [-f FMT] [-F FMT] [-T SRC_CACHE] [-p] [-q] [-s]`

Check if two images have the same content. You can compare images with different format or settings.

The format is probed unless you specify it by `-f` (used for *FILENAME1*) and/or `-F` (used for *FILENAME2*) option.

By default, images with different size are considered identical if the larger image contains only unallocated and/or zeroed sectors in the area after the end of the other image. In addition, if any sector is not allocated in one image and contains only zero bytes in the second one, it is evaluated as equal. You can use Strict mode by specifying the `-s` option. When compare runs in Strict mode, it fails in case image size differs or a sector is allocated in one image and is not allocated in the second one.

By default, compare prints out a result message. This message displays information that both images are same or the position of the first different byte. In addition, result message can report different image size in case Strict mode is used.

Compare exits with `0` in case the images are equal and with `1` in case the images differ. Other exit codes mean an error occurred during execution and standard error output should contain an error message. The following table sumarizes all exit codes of the compare subcommand:

0 Images are identical

1 Images differ

2 Error on opening an image

3 Error on checking a sector allocation

4 Error on reading data

`convert` `[--object OBJECTDEF] [--image-opts] [--target-image-opts] [--target-is-zero] [--bi`

Convert the disk image *FILENAME* or a snapshot *SNAPSHOT_PARAM* to disk image *OUTPUT_FILENAME* using format *OUTPUT_FMT*. It can be optionally compressed (`-c` option) or use any format specific options like encryption (`-o` option).

Only the formats `qcow` and `qcow2` support compression. The compression is read-only. It means that if a compressed sector is rewritten, then it is rewritten as uncompressed data.

Image conversion is also useful to get smaller image when using a growable format such as `qcow`: the empty sectors are detected and suppressed from the destination image.

SPARSE_SIZE indicates the consecutive number of bytes (defaults to 4k) that must contain only zeros for qemu-img to create a sparse image during conversion. If *SPARSE_SIZE* is 0, the source will not be scanned for unallocated or zero sectors, and the destination image will always be fully allocated.

You can use the *BACKING_FILE* option to force the output image to be created as a copy on write image of the specified base image; the *BACKING_FILE* should have the same content as the input's base image, however the path, image format, etc may differ.

If a relative path name is given, the backing file is looked up relative to the directory containing *OUTPUT_FILENAME*.

If the `-n` option is specified, the target volume creation will be skipped. This is useful for formats such as `rbd` if the target volume has already been created with site specific options that cannot be supplied through qemu-img.

Out of order writes can be enabled with `-W` to improve performance. This is only recommended for preallocated devices like host devices or other raw block devices. Out of order write does not work in combination with creating compressed images.

NUM_COROUTINES specifies how many coroutines work in parallel during the convert process (defaults to 8).

`create` `[--object OBJECTDEF] [-q] [-f FMT] [-b BACKING_FILE] [-F BACKING_FMT] [-u] [-o OPTIO`

Create the new disk image *FILENAME* of size *SIZE* and format *FMT*. Depending on the file format, you can add one or more *OPTIONS* that enable additional features of this format.

If the option *BACKING_FILE* is specified, then the image will record only the differences from *BACKING_FILE*. No size needs to be specified in this case. *BACKING_FILE* will never be modified unless you use the `commit` monitor command (or qemu-img commit).

If a relative path name is given, the backing file is looked up relative to the directory containing *FILENAME*.

Note that a given backing file will be opened to check that it is valid. Use the `-u` option to enable unsafe backing file mode, which means that the image will be created even if the associated backing file cannot be opened. A matching backing file must be created or additional options be used to make the backing file specification valid when you want to use an image created this way.

The size can also be specified using the *SIZE* option with `-o`, it doesn't need to be specified separately in this case.

`dd` `[--image-opts] [-U] [-f FMT] [-O OUTPUT_FMT] [bs=BLOCK_SIZE] [count=BLOCKS] [skip=BLOCKS`

dd copies from *INPUT* file to *OUTPUT* file converting it from *FMT* format to *OUTPUT_FMT* format.

The data is by default read and written using blocks of 512 bytes but can be modified by specifying *BLOCK_SIZE*. If count=*BLOCKS* is specified dd will stop reading input after reading *BLOCKS* input blocks.

The size syntax is similar to *`dd(1)`*'s size syntax.

`info` `[--object OBJECTDEF] [--image-opts] [-f FMT] [--output=OFMT] [--backing-chain] [-U] FI`

Give information about the disk image *FILENAME*. Use it in particular to know the size reserved on disk which can be different from the displayed size. If VM snapshots are stored in the disk image, they are displayed too.

If a disk image has a backing file chain, information about each disk image in the chain can be recursively enumerated by using the option `--backing-chain`.

For instance, if you have an image chain like:

```
base.qcow2 <- snap1.qcow2 <- snap2.qcow2
```

To enumerate information about each disk image in the above chain, starting from top to base, do:

```
qemu-img info --backing-chain snap2.qcow2
```

The command can output in the format *OFMT* which is either `human` or `json`. The JSON output is an object of QAPI type `ImageInfo`; with `--backing-chain`, it is an array of `ImageInfo` objects.

`--output=human` reports the following information (for every image in the chain):

image The image file name

file format The image format

virtual size The size of the guest disk

disk size How much space the image file occupies on the host file system (may be shown as 0 if this information is unavailable, e.g. because there is no file system)

cluster_size Cluster size of the image format, if applicable

encrypted Whether the image is encrypted (only present if so)

cleanly shut down This is shown as `no` if the image is dirty and will have to be auto-repaired the next time it is opened in qemu.

backing file The backing file name, if present

backing file format The format of the backing file, if the image enforces it

Snapshot list A list of all internal snapshots

Format specific information Further information whose structure depends on the image format. This section is a textual representation of the respective `ImageInfoSpecific*` QAPI object (e.g. `ImageInfoSpecificQCow2` for qcow2 images).

`map` `[--object OBJECTDEF] [--image-opts] [-f FMT] [--start-offset=OFFSET] [--max-length=LEN`
Dump the metadata of image *FILENAME* and its backing file chain. In particular, this commands dumps the allocation state of every sector of *FILENAME*, together with the topmost file that allocates it in the backing file chain.

Two option formats are possible. The default format (`human`) only dumps known-nonzero areas of the file. Known-zero parts of the file are omitted altogether, and likewise for parts that are not allocated throughout the chain. `qemu-img` output will identify a file from where the data can be read, and the offset in the file. Each line will include four fields, the first three of which are hexadecimal numbers. For example the first line of:

```
Offset          Length          Mapped to       File
0               0x20000         0x50000         /tmp/overlay.qcow2
0x100000        0x10000         0x95380000      /tmp/backing.qcow2
```

means that 0x20000 (131072) bytes starting at offset 0 in the image are available in /tmp/overlay.qcow2 (opened in `raw` format) starting at offset 0x50000 (327680). Data that is compressed, encrypted, or otherwise not available in raw format will cause an error if `human` format is in use. Note that file names can include newlines, thus it is not safe to parse this output format in scripts.

The alternative format `json` will return an array of dictionaries in JSON format. It will include similar information in the `start`, `length`, `offset` fields; it will also include other more specific information:

- whether the sectors contain actual data or not (boolean field `data`; if false, the sectors are either unallocated or stored as optimized all-zero clusters);
- whether the data is known to read as zero (boolean field `zero`);

- in order to make the output shorter, the target file is expressed as a `depth`; for example, a depth of 2 refers to the backing file of the backing file of *FILENAME*.

In JSON format, the `offset` field is optional; it is absent in cases where `human` format would omit the entry or exit with an error. If `data` is false and the `offset` field is present, the corresponding sectors in the file are not yet in use, but they are preallocated.

For more information, consult `include/block/block.h` in QEMU's source code.

`measure` `[--output=OFMT] [-O OUTPUT_FMT] [-o OPTIONS] [--size N | [--object OBJECTDEF] [--i`
Calculate the file size required for a new image. This information can be used to size logical volumes or SAN LUNs appropriately for the image that will be placed in them. The values reported are guaranteed to be large enough to fit the image. The command can output in the format *OFMT* which is either `human` or `json`. The JSON output is an object of QAPI type `BlockMeasureInfo`.

If the size *N* is given then act as if creating a new empty image file using `qemu-img create`. If *FILENAME* is given then act as if converting an existing image file using `qemu-img convert`. The format of the new file is given by *OUTPUT_FMT* while the format of an existing file is given by *FMT*.

A snapshot in an existing image can be specified using *SNAPSHOT_PARAM*.

The following fields are reported:

```
required size: 524288
fully allocated size: 1074069504
bitmaps size: 0
```

The `required size` is the file size of the new image. It may be smaller than the virtual disk size if the image format supports compact representation.

The `fully allocated size` is the file size of the new image once data has been written to all sectors. This is the maximum size that the image file can occupy with the exception of internal snapshots, dirty bitmaps, vmstate data, and other advanced image format features.

The `bitmaps size` is the additional size required in order to copy bitmaps from a source image in addition to the guest-visible data; the line is omitted if either source or destination lacks bitmap support, or 0 if bitmaps are supported but there is nothing to copy.

`snapshot` `[--object OBJECTDEF] [--image-opts] [-U] [-q] [-l | -a SNAPSHOT | -c SNAPSHOT | -`
List, apply, create or delete snapshots in image *FILENAME*.

`rebase` `[--object OBJECTDEF] [--image-opts] [-U] [-q] [-f FMT] [-t CACHE] [-T SRC_CACHE] [-`
Changes the backing file of an image. Only the formats `qcow2` and `qed` support changing the backing file.

The backing file is changed to *BACKING_FILE* and (if the image format of *FILENAME* supports this) the backing file format is changed to *BACKING_FMT*. If *BACKING_FILE* is specified as "" (the empty string), then the image is rebased onto no backing file (i.e. it will exist independently of any backing file).

If a relative path name is given, the backing file is looked up relative to the directory containing *FILENAME*.

CACHE specifies the cache mode to be used for *FILENAME*, whereas *SRC_CACHE* specifies the cache mode for reading backing files.

There are two different modes in which `rebase` can operate:

Safe mode This is the default mode and performs a real rebase operation. The new backing file may differ from the old one and qemu-img rebase will take care of keeping the guest-visible content of *FILENAME* unchanged.

In order to achieve this, any clusters that differ between *BACKING_FILE* and the old backing file of *FILENAME* are merged into *FILENAME* before actually changing the backing file.

Note that the safe mode is an expensive operation, comparable to converting an image. It only works if the old backing file still exists.

Unsafe mode qemu-img uses the unsafe mode if `-u` is specified. In this mode, only the backing file name and format of *FILENAME* is changed without any checks on the file contents. The user must take care of specifying the correct new backing file, or the guest-visible content of the image will be corrupted.

This mode is useful for renaming or moving the backing file to somewhere else. It can be used without an accessible old backing file, i.e. you can use it to fix an image whose backing file has already been moved/renamed.

You can use `rebase` to perform a "diff" operation on two disk images. This can be useful when you have copied or cloned a guest, and you want to get back to a thin image on top of a template or base image.

Say that `base.img` has been cloned as `modified.img` by copying it, and that the `modified.img` guest has run so there are now some changes compared to `base.img`. To construct a thin image called `diff.qcow2` that contains just the differences, do:

```
qemu-img create -f qcow2 -b modified.img diff.qcow2
qemu-img rebase -b base.img diff.qcow2
```

At this point, `modified.img` can be discarded, since `base.img + diff.qcow2` contains the same information.

`resize` `[--object OBJECTDEF] [--image-opts] [-f FMT] [--preallocation=PREALLOC] [-q] [--shr`
Change the disk image as if it had been created with *SIZE*.

Before using this command to shrink a disk image, you MUST use file system and partitioning tools inside the VM to reduce allocated file systems and partition sizes accordingly. Failure to do so will result in data loss!

When shrinking images, the `--shrink` option must be given. This informs qemu-img that the user acknowledges all loss of data beyond the truncated image's end.

After using this command to grow a disk image, you must use file system and partitioning tools inside the VM to actually begin using the new space on the device.

When growing an image, the `--preallocation` option may be used to specify how the additional image area should be allocated on the host. See the format description in the *Notes* section which values are allowed. Using this option may result in slightly more data being allocated than necessary.

3.1.4 Notes

Supported image file formats:

`raw`

Raw disk image format (default). This format has the advantage of being simple and easily exportable to all other emulators. If your file system supports *holes* (for example in ext2 or ext3 on Linux or NTFS on Windows), then only the written sectors will reserve space. Use `qemu-img info` to know the real size used by the image or `ls -ls` on Unix/Linux.

Supported options:

`preallocation` Preallocation mode (allowed values: `off`, `falloc`, `full`). `falloc` mode preallocates space for image by calling `posix_fallocate()`. `full` mode preallocates space for image by writing data to underlying storage. This data may or may not be zero, depending on the storage location.

`qcow2`

QEMU image format, the most versatile format. Use it to have smaller images (useful if your filesystem does not supports holes, for example on Windows), optional AES encryption, zlib based compression and support of multiple VM snapshots.

Supported options:

`compat` Determines the qcow2 version to use. `compat=0.10` uses the traditional image format that can be read by any QEMU since 0.10. `compat=1.1` enables image format extensions that only QEMU 1.1 and newer understand (this is the default). Amongst others, this includes zero clusters, which allow efficient copy-on-read for sparse images.

`backing_file` File name of a base image (see `create` subcommand)

`backing_fmt` Image format of the base image

`encryption` If this option is set to `on`, the image is encrypted with 128-bit AES-CBC.

The use of encryption in qcow and qcow2 images is considered to be flawed by modern cryptography standards, suffering from a number of design problems:

- The AES-CBC cipher is used with predictable initialization vectors based on the sector number. This makes it vulnerable to chosen plaintext attacks which can reveal the existence of encrypted data.
- The user passphrase is directly used as the encryption key. A poorly chosen or short passphrase will compromise the security of the encryption.
- In the event of the passphrase being compromised there is no way to change the passphrase to protect data in any qcow images. The files must be cloned, using a different encryption passphrase in the new file. The original file must then be securely erased using a program like shred, though even this is ineffective with many modern storage technologies.
- Initialization vectors used to encrypt sectors are based on the guest virtual sector number, instead of the host physical sector. When a disk image has multiple internal snapshots this means that data in multiple physical sectors is encrypted with the same initialization vector. With the CBC mode, this opens the possibility of watermarking attacks if the attack can collect multiple sectors encrypted with the same IV and some predictable data. Having multiple qcow2 images with the same passphrase also exposes this weakness since the passphrase is directly used as the key.

Use of qcow / qcow2 encryption is thus strongly discouraged. Users are recommended to use an alternative encryption technology such as the Linux dm-crypt / LUKS system.

`cluster_size` Changes the qcow2 cluster size (must be between 512 and 2M). Smaller cluster sizes can improve the image file size whereas larger cluster sizes generally provide better performance.

`preallocation` Preallocation mode (allowed values: `off`, `metadata`, `falloc`, `full`). An image with preallocated metadata is initially larger but can improve performance when the image needs to grow. `falloc` and `full` preallocations are like the same options of `raw` format, but sets up metadata also.

`lazy_refcounts` If this option is set to `on`, reference count updates are postponed with the goal of avoiding metadata I/O and improving performance. This is particularly interesting with `cache=writethrough` which doesn't batch metadata updates. The tradeoff is that after a host crash, the reference count tables must be rebuilt, i.e. on the next open an (automatic) `qemu-img check -r all` is required, which may take some time.

This option can only be enabled if `compat=1.1` is specified.

`nocow` If this option is set to `on`, it will turn off COW of the file. It's only valid on btrfs, no effect on other file systems.

Btrfs has low performance when hosting a VM image file, even more when the guest on the VM also using btrfs as file system. Turning off COW is a way to mitigate this bad performance. Generally there are two ways to turn off COW on btrfs:

- Disable it by mounting with nodatacow, then all newly created files will be NOCOW
- For an empty file, add the NOCOW file attribute. That's what this option does.

Note: this option is only valid to new or empty files. If there is an existing file which is COW and has data blocks already, it couldn't be changed to NOCOW by setting `nocow=on`. One can issue `lsattr filename` to check if the NOCOW flag is set or not (Capital 'C' is NOCOW flag).

`Other`

QEMU also supports various other image file formats for compatibility with older QEMU versions or other hypervisors, including VMDK, VDI, VHD (vpc), VHDX, qcow1 and QED. For a full list of supported formats see `qemu-img --help`. For a more detailed description of these formats, see the QEMU block drivers reference documentation.

The main purpose of the block drivers for these formats is image conversion. For running VMs, it is recommended to convert the disk images to either raw or qcow2 in order to achieve good performance.

3.2 QEMU Disk Network Block Device Server

3.2.1 Synopsis

qemu-nbd [*OPTION*]... *filename*

qemu-nbd -L [*OPTION*]...

qemu-nbd -d *dev*

3.2.2 Description

Export a QEMU disk image using the NBD protocol.

Other uses:

- Bind a /dev/nbdX block device to a QEMU server (on Linux).
- As a client to query exports of a remote NBD server.

3.2.3 Options

filename is a disk image filename, or a set of block driver options if `--image-opts` is specified.

dev is an NBD device.

`--object` `type,id=ID,...props...`
Define a new instance of the *type* object class identified by *ID*. See the *`qemu(1)`* manual page for full details of the properties supported. The common object types that it makes sense to define are the `secret` object, which is used to supply passwords and/or encryption keys, and the `tls-creds` object, which is used to supply TLS credentials for the qemu-nbd server or client.

`-p, --port``=PORT`
TCP port to listen on as a server, or connect to as a client (default `10809`).

-o, --offset=OFFSET
: The offset into the image.

-b, --bind=IFACE
: The interface to bind to as a server, or connect to as a client (default `0.0.0.0`).

-k, --socket=PATH
: Use a unix socket with path *PATH*.

--image-opts
: Treat *filename* as a set of image options, instead of a plain filename. If this flag is specified, the `-f` flag should not be used, instead the `format=` option should be set.

-f, --format=FMT
: Force the use of the block driver for format *FMT* instead of auto-detecting.

-r, --read-only
: Export the disk as read-only.

-B, --bitmap=NAME
: If *filename* has a qcow2 persistent bitmap *NAME*, expose that bitmap via the `qemu:dirty-bitmap:NAME` context accessible through NBD_OPT_SET_META_CONTEXT.

-s, --snapshot
: Use *filename* as an external snapshot, create a temporary file with `backing_file=`*filename*, redirect the write to the temporary one.

-l, --load-snapshot=SNAPSHOT_PARAM
: Load an internal snapshot inside *filename* and export it as an read-only device, SNAPSHOT_PARAM format is `snapshot.id=[ID],snapshot.name=[NAME]` or `[ID_OR_NAME]`

--cache=CACHE
: The cache mode to be used with the file. See the documentation of the emulator's `-drive cache=...` option for allowed values.

-n, --nocache
: Equivalent to `--cache=none`.

--aio=AIO
: Set the asynchronous I/O mode between `threads` (the default), `native` (Linux only), and `io_uring` (Linux 5.1+).

--discard=DISCARD
: Control whether `discard` (also known as `trim` or `unmap`) requests are ignored or passed to the filesystem. *DISCARD* is one of `ignore` (or `off`), `unmap` (or `on`). The default is `ignore`.

--detect-zeroes=DETECT_ZEROES
: Control the automatic conversion of plain zero writes by the OS to driver-specific optimized zero write commands. *DETECT_ZEROES* is one of `off`, `on`, or `unmap`. `unmap` converts a zero write to an unmap operation and can only be used if *DISCARD* is set to `unmap`. The default is `off`.

-c, --connect=DEV
: Connect *filename* to NBD device *DEV* (Linux only).

-d, --disconnect
: Disconnect the device *DEV* (Linux only).

-e, --shared=NUM
: Allow up to *NUM* clients to share the device (default `1`). Safe for readers, but for now, consistency is not guaranteed between multiple writers.

-t, --persistent
Don't exit on the last connection.

-x, --export-name=NAME
Set the NBD volume export name (default of a zero-length string).

-D, --description=DESCRIPTION
Set the NBD volume export description, as a human-readable string.

-L, --list
Connect as a client and list all details about the exports exposed by a remote NBD server. This enables list mode, and is incompatible with options that change behavior related to a specific export (such as `--export-name`, `--offset`, ...).

--tls-creds=ID
Enable mandatory TLS encryption for the server by setting the ID of the TLS credentials object previously created with the –object option; or provide the credentials needed for connecting as a client in list mode.

--fork
Fork off the server process and exit the parent once the server is running.

--pid-file=PATH
Store the server's process ID in the given file.

--tls-authz=ID
Specify the ID of a qauthz object previously created with the `--object` option. This will be used to authorize connecting users against their x509 distinguished name.

-v, --verbose
Display extra debugging information.

-h, --help
Display this help and exit.

-V, --version
Display version information and exit.

-T, --trace [[enable=]PATTERN][,events=FILE][,file=FILE]
Specify tracing options.

[enable=]PATTERN
Immediately enable events matching *PATTERN* (either event name or a globbing pattern). This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend. To specify multiple events or patterns, specify the `-trace` option multiple times.

Use `-trace help` to print a list of names of trace points.

events=FILE
Immediately enable events listed in *FILE*. The file must contain one event name (as listed in the `trace-events-all` file) per line; globbing patterns are accepted too. This option is only available if QEMU has been compiled with the `simple`, `log` or `ftrace` tracing backend.

file=FILE
Log output traces to *FILE*. This option is only available if QEMU has been compiled with the `simple` tracing backend.

3.2.4 Examples

Start a server listening on port 10809 that exposes only the guest-visible contents of a qcow2 file, with no TLS encryption, and with the default export name (an empty string). The command is one-shot, and will block until the first successful client disconnects:

```
qemu-nbd -f qcow2 file.qcow2
```

Start a long-running server listening with encryption on port 10810, and whitelist clients with a specific X.509 certificate to connect to a 1 megabyte subset of a raw file, using the export name 'subset':

```
qemu-nbd \
  --object tls-creds-x509,id=tls0,endpoint=server,dir=/path/to/qemutls \
  --object 'authz-simple,id=auth0,identity=CN=laptop.example.com,,\
            O=Example Org,,L=London,,ST=London,,C=GB' \
  --tls-creds tls0 --tls-authz auth0 \
  -t -x subset -p 10810 \
  --image-opts driver=raw,offset=1M,size=1M,file.driver=file,file.filename=file.raw
```

Serve a read-only copy of a guest image over a Unix socket with as many as 5 simultaneous readers, with a persistent process forked as a daemon:

```
qemu-nbd --fork --persistent --shared=5 --socket=/path/to/sock \
  --read-only --format=qcow2 file.qcow2
```

Expose the guest-visible contents of a qcow2 file via a block device /dev/nbd0 (and possibly creating /dev/nbd0p1 and friends for partitions found within), then disconnect the device when done. Access to bind qemu-nbd to an /dev/nbd device generally requires root privileges, and may also require the execution of `modprobe nbd` to enable the kernel NBD client module. *CAUTION*: Do not use this method to mount filesystems from an untrusted guest image - a malicious guest may have prepared the image to attempt to trigger kernel bugs in partition probing or file system mounting.

```
qemu-nbd -c /dev/nbd0 -f qcow2 file.qcow2
qemu-nbd -d /dev/nbd0
```

Query a remote server to see details about what export(s) it is serving on port 10809, and authenticating via PSK:

```
qemu-nbd \
  --object tls-creds-psk,id=tls0,dir=/tmp/keys,username=eblake,endpoint=client \
  --tls-creds tls0 -L -b remote.example.com
```

3.2.5 See also

qemu(1), *qemu-img(1)*

3.3 QEMU SystemTap trace tool

3.3.1 Synopsis

qemu-trace-stap [*GLOBAL-OPTIONS*] *COMMAND* [*COMMAND-OPTIONS*] *ARGS…*

3.3.2 Description

The `qemu-trace-stap` program facilitates tracing of the execution of QEMU emulators using SystemTap.

It is required to have the SystemTap runtime environment installed to use this program, since it is a wrapper around execution of the `stap` program.

3.3.3 Options

The following global options may be used regardless of which command is executed:

--verbose, -v
Display verbose information about command execution.

The following commands are valid:

list `BINARY PATTERN...`
List all the probe names provided by *BINARY* that match *PATTERN*.

If *BINARY* is not an absolute path, it will be located by searching the directories listed in the `$PATH` environment variable.

PATTERN is a plain string that is used to filter the results of this command. It may optionally contain a `*` wildcard to facilitate matching multiple probes without listing each one explicitly. Multiple *PATTERN* arguments may be given, causing listing of probes that match any of the listed names. If no *PATTERN* is given, the all possible probes will be listed.

For example, to list all probes available in the `qemu-system-x86_64` binary:

```
$ qemu-trace-stap list qemu-system-x86_64
```

To filter the list to only cover probes related to QEMU's cryptographic subsystem, in a binary outside `$PATH`

```
$ qemu-trace-stap list /opt/qemu/4.0.0/bin/qemu-system-x86_64 'qcrypto*'
```

run `OPTIONS BINARY PATTERN...`
Run a trace session, printing formatted output any time a process that is executing *BINARY* triggers a probe matching *PATTERN*.

If *BINARY* is not an absolute path, it will be located by searching the directories listed in the `$PATH` environment variable.

PATTERN is a plain string that matches a probe name shown by the *LIST* command. It may optionally contain a `*` wildcard to facilitate matching multiple probes without listing each one explicitly. Multiple *PATTERN* arguments may be given, causing all matching probes to be monitored. At least one *PATTERN* is required, since stap is not capable of tracing all known QEMU probes concurrently without overflowing its trace buffer.

Invocation of this command does not need to be synchronized with invocation of the QEMU process(es). It will match probes on all existing running processes and all future launched processes, unless told to only monitor a specific process.

Valid command specific options are:

--pid=`PID`, **-p** `PID`
Restrict the tracing session so that it only triggers for the process identified by *PID*.

For example, to monitor all processes executing `qemu-system-x86_64` as found on `$PATH`, displaying all I/O related probes:

```
$ qemu-trace-stap run qemu-system-x86_64 'qio*'
```

To monitor only the QEMU process with PID 1732

```
$ qemu-trace-stap run --pid=1732 qemu-system-x86_64 'qio*'
```

To monitor QEMU processes running an alternative binary outside of `$PATH`, displaying verbose information about setup of the tracing environment:

```
$ qemu-trace-stap -v run /opt/qemu/4.0.0/qemu-system-x86_64 'qio*'
```

3.3.4 See also

qemu(1), *stap(1)*

3.4 QEMU 9p virtfs proxy filesystem helper

3.4.1 Synopsis

virtfs-proxy-helper [*OPTIONS*]

3.4.2 Description

Pass-through security model in QEMU 9p server needs root privilege to do few file operations (like chown, chmod to any mode/uid:gid). There are two issues in pass-through security model:

- TOCTTOU vulnerability: Following symbolic links in the server could provide access to files beyond 9p export path.
- Running QEMU with root privilege could be a security issue.

To overcome above issues, following approach is used: A new filesystem type 'proxy' is introduced. Proxy FS uses chroot + socket combination for securing the vulnerability known with following symbolic links. Intention of adding a new filesystem type is to allow qemu to run in non-root mode, but doing privileged operations using socket IO.

Proxy helper (a stand alone binary part of qemu) is invoked with root privileges. Proxy helper chroots into 9p export path and creates a socket pair or a named socket based on the command line parameter. QEMU and proxy helper communicate using this socket. QEMU proxy fs driver sends filesystem request to proxy helper and receives the response from it.

The proxy helper is designed so that it can drop root privileges except for the capabilities needed for doing filesystem operations.

3.4.3 Options

The following options are supported:

-h
Display help and exit

-p, --path `PATH`
Path to export for proxy filesystem driver

-f, --fd `SOCKET_ID`
Use given file descriptor as socket descriptor for communicating with qemu proxy fs drier. Usually a helper like libvirt will create socketpair and pass one of the fds as parameter to this option.

-s, --socket `SOCKET_FILE`
Creates named socket file for communicating with qemu proxy fs driver

-u, --uid `UID`
uid to give access to named socket file; used in combination with -g.

-g, --gid `GID`
gid to give access to named socket file; used in combination with -u.

-n, --nodaemon
Run as a normal program. By default program will run in daemon mode

3.5 QEMU virtio-fs shared file system daemon

3.5.1 Synopsis

virtiofsd [*OPTIONS*]

3.5.2 Description

Share a host directory tree with a guest through a virtio-fs device. This program is a vhost-user backend that implements the virtio-fs device. Each virtio-fs device instance requires its own virtiofsd process.

This program is designed to work with QEMU's `--device vhost-user-fs-pci` but should work with any virtual machine monitor (VMM) that supports vhost-user. See the Examples section below.

This program must be run as the root user. Upon startup the program will switch into a new file system namespace with the shared directory tree as its root. This prevents "file system escapes" due to symlinks and other file system objects that might lead to files outside the shared directory. The program also sandboxes itself using seccomp(2) to prevent ptrace(2) and other vectors that could allow an attacker to compromise the system after gaining control of the virtiofsd process.

3.5.3 Options

-h, --help
Print help.

-V, --version
Print version.

-d
Enable debug output.

--syslog
Print log messages to syslog instead of stderr.

-o `OPTION`

- debug - Enable debug output.
- flock|no_flock - Enable/disable flock. The default is `no_flock`.
- modcaps=CAPLIST Modify the list of capabilities allowed; CAPLIST is a colon separated list of capabilities, each preceded by either + or -, e.g. ''+sys_admin:-chown''.
- log_level=LEVEL - Print only log messages matching LEVEL or more severe. LEVEL is one of `err`, `warn`, `info`, or `debug`. The default is `info`.
- posix_lock|no_posix_lock - Enable/disable remote POSIX locks. The default is `no_posix_lock`.
- readdirplus|no_readdirplus - Enable/disable readdirplus. The default is `readdirplus`.
- source=PATH - Share host directory tree located at PATH. This option is required.

- timeout=TIMEOUT - I/O timeout in seconds. The default depends on cache= option.
- writeback|no_writeback - Enable/disable writeback cache. The cache alows the FUSE client to buffer and merge write requests. The default is `no_writeback`.
- xattr|no_xattr - Enable/disable extended attributes (xattr) on files and directories. The default is `no_xattr`.

`--socket-path``=PATH`
Listen on vhost-user UNIX domain socket at PATH.

`--fd``=FDNUM`
Accept connections from vhost-user UNIX domain socket file descriptor FDNUM. The file descriptor must already be listening for connections.

`--thread-pool-size``=NUM`
Restrict the number of worker threads per request queue to NUM. The default is 64.

`--cache``=none|auto|always`
Select the desired trade-off between coherency and performance. `none` forbids the FUSE client from caching to achieve best coherency at the cost of performance. `auto` acts similar to NFS with a 1 second metadata cache timeout. `always` sets a long cache lifetime at the expense of coherency.

3.5.4 Examples

Export `/var/lib/fs/vm001/` on vhost-user UNIX domain socket `/var/run/vm001-vhost-fs.sock`:

```
host# virtiofsd --socket-path=/var/run/vm001-vhost-fs.sock -o source=/var/lib/fs/vm001
host# qemu-system-x86_64 \
    -chardev socket,id=char0,path=/var/run/vm001-vhost-fs.sock \
    -device vhost-user-fs-pci,chardev=char0,tag=myfs \
    -object memory-backend-memfd,id=mem,size=4G,share=on \
    -numa node,memdev=mem \
    ...
guest# mount -t virtiofs myfs /mnt
```

CHAPTER 4

QEMU System Emulation Management and Interoperability Guide

This manual contains documents and specifications that are useful for making QEMU interoperate with other software.

Contents:

4.1 Dirty Bitmaps and Incremental Backup

Dirty Bitmaps are in-memory objects that track writes to block devices. They can be used in conjunction with various block job operations to perform incremental or differential backup regimens.

This document explains the conceptual mechanisms, as well as up-to-date, complete and comprehensive documentation on the API to manipulate them. (Hopefully, the "why", "what", and "how".)

The intended audience for this document is developers who are adding QEMU backup features to management applications, or power users who run and administer QEMU directly via QMP.

Contents

4.1.1 Overview

Bitmaps are bit vectors where each '1' bit in the vector indicates a modified ("dirty") segment of the corresponding block device. The size of the segment that is tracked is the granularity of the bitmap. If the granularity of a bitmap is 64K, each '1' bit means that a 64K region as a whole may have changed in some way, possibly by as little as one byte.

Smaller granularities mean more accurate tracking of modified disk data, but requires more computational overhead and larger bitmap sizes. Larger granularities mean smaller bitmap sizes, but less targeted backups.

The size of a bitmap (in bytes) can be computed as such: `size` = ceil(ceil(`image_size` / `granularity`) / 8)

e.g. the size of a 64KiB granularity bitmap on a 2TiB image is:

`size` = ((2147483648K / 64K) / 8) = 4194304B = 4MiB.

QEMU uses these bitmaps when making incremental backups to know which sections of the file to copy out. They are not enabled by default and must be explicitly added in order to begin tracking writes.

Bitmaps can be created at any time and can be attached to any arbitrary block node in the storage graph, but are most useful conceptually when attached to the root node attached to the guest's storage device model.

That is to say: It's likely most useful to track the guest's writes to disk, but you could theoretically track things like qcow2 metadata changes by attaching the bitmap elsewhere in the storage graph. This is beyond the scope of this document.

QEMU supports persisting these bitmaps to disk via the qcow2 image format. Bitmaps which are stored or loaded in this way are called "persistent", whereas bitmaps that are not are called "transient".

QEMU also supports the migration of both transient bitmaps (tracking any arbitrary image format) or persistent bitmaps (qcow2) via live migration.

4.1.2 Supported Image Formats

QEMU supports all documented features below on the qcow2 image format.

However, qcow2 is only strictly necessary for the persistence feature, which writes bitmap data to disk upon close. If persistence is not required for a specific use case, all bitmap features excepting persistence are available for any arbitrary image format.

For example, Dirty Bitmaps can be combined with the 'raw' image format, but any changes to the bitmap will be discarded upon exit.

Warning: Transient bitmaps will not be saved on QEMU exit! Persistent bitmaps are available only on qcow2 images.

4.1.3 Dirty Bitmap Names

Bitmap objects need a method to reference them in the API. All API-created and managed bitmaps have a human-readable name chosen by the user at creation time.

- A bitmap's name is unique to the node, but bitmaps attached to different nodes can share the same name. Therefore, all bitmaps are addressed via their (node, name) pair.
- The name of a user-created bitmap cannot be empty ("").
- Transient bitmaps can have JSON unicode names that are effectively not length limited. (QMP protocol may restrict messages to less than 64MiB.)
- Persistent storage formats may impose their own requirements on bitmap names and namespaces. Presently, only qcow2 supports persistent bitmaps. See docs/interop/qcow2.txt for more details on restrictions. Notably:
  - qcow2 bitmap names are limited to between 1 and 1023 bytes long.
  - No two bitmaps saved to the same qcow2 file may share the same name.
- QEMU occasionally uses bitmaps for internal use which have no name. They are hidden from API query calls, cannot be manipulated by the external API, are never persistent, nor ever migrated.

4.1.4 Bitmap Status

Dirty Bitmap objects can be queried with the QMP command query-block, and are visible via the BlockDirtyInfo QAPI structure.

This struct shows the name, granularity, and dirty byte count for each bitmap. Additionally, it shows several boolean status indicators:

- `recording`: This bitmap is recording writes.
- `busy`: This bitmap is in-use by an operation.
- `persistent`: This bitmap is a persistent type.

- `inconsistent`: This bitmap is corrupted and cannot be used.

The `+busy` status prohibits you from deleting, clearing, or otherwise modifying a bitmap, and happens when the bitmap is being used for a backup operation or is in the process of being loaded from a migration. Many of the commands documented below will refuse to work on such bitmaps.

The `+inconsistent` status similarly prohibits almost all operations, notably allowing only the `block-dirty-bitmap-remove` operation.

There is also a deprecated `status` field of type DirtyBitmapStatus. A bitmap historically had five visible states:

1. `Frozen`: This bitmap is currently in-use by an operation and is immutable. It can't be deleted, renamed, reset, etc.

   (This is now `+busy`.)

2. `Disabled`: This bitmap is not recording new writes.

   (This is now `-recording -busy`.)

3. `Active`: This bitmap is recording new writes.

   (This is now `+recording -busy`.)

4. `Locked`: This bitmap is in-use by an operation, and is immutable. The difference from "Frozen" was primarily implementation details.

   (This is now `+busy`.)

5. `Inconsistent`: This persistent bitmap was not saved to disk correctly, and can no longer be used. It remains in memory to serve as an indicator of failure.

   (This is now `+inconsistent`.)

These states are directly replaced by the status indicators and should not be used. The difference between `Frozen` and `Locked` is an implementation detail and should not be relevant to external users.

4.1.5 Basic QMP Usage

The primary interface to manipulating bitmap objects is via the QMP interface. If you are not familiar, see docs/interop/qmp-intro.txt for a broad overview, and qemu-qmp-ref for a full reference of all QMP commands.

Supported Commands

There are six primary bitmap-management API commands:

- `block-dirty-bitmap-add`
- `block-dirty-bitmap-remove`
- `block-dirty-bitmap-clear`
- `block-dirty-bitmap-disable`
- `block-dirty-bitmap-enable`
- `block-dirty-bitmap-merge`

And one related query command:

- `query-block`

Creation: block-dirty-bitmap-add

block-dirty-bitmap-add:

Creates a new bitmap that tracks writes to the specified node. granularity, persistence, and recording state can be adjusted at creation time.

Example

to create a new, actively recording persistent bitmap:

```
-> { "execute": "block-dirty-bitmap-add",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0",
       "persistent": true,
     }
   }

<- { "return": {} }
```

- This bitmap will have a default granularity that matches the cluster size of its associated drive, if available, clamped to between [4KiB, 64KiB]. The current default for qcow2 is 64KiB.

Example

To create a new, disabled (`-recording`), transient bitmap that tracks changes in 32KiB segments:

```
-> { "execute": "block-dirty-bitmap-add",
     "arguments": {
       "node": "drive0",
       "name": "bitmap1",
       "granularity": 32768,
       "disabled": true
     }
   }

<- { "return": {} }
```

Deletion: block-dirty-bitmap-remove

block-dirty-bitmap-remove:

Deletes a bitmap. Bitmaps that are `+busy` cannot be removed.

- Deleting a bitmap does not impact any other bitmaps attached to the same node, nor does it affect any backups already created from this bitmap or node.
- Because bitmaps are only unique to the node to which they are attached, you must specify the node/drive name here, too.
- Deleting a persistent bitmap will remove it from the qcow2 file.

Example

Remove a bitmap named `bitmap0` from node `drive0`:

```
-> { "execute": "block-dirty-bitmap-remove",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0"
     }
   }

<- { "return": {} }
```

Resetting: block-dirty-bitmap-clear

block-dirty-bitmap-clear:

Clears all dirty bits from a bitmap. `+busy` bitmaps cannot be cleared.

- An incremental backup created from an empty bitmap will copy no data, as if nothing has changed.

Example

Clear all dirty bits from bitmap `bitmap0` on node `drive0`:

```
-> { "execute": "block-dirty-bitmap-clear",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0"
     }
   }

<- { "return": {} }
```

Enabling: block-dirty-bitmap-enable

block-dirty-bitmap-enable:

"Enables" a bitmap, setting the `recording` bit to true, causing writes to begin being recorded. `+busy` bitmaps cannot be enabled.

- Bitmaps default to being enabled when created, unless configured otherwise.
- Persistent enabled bitmaps will remember their `+recording` status on load.

Example

To set `+recording` on bitmap `bitmap0` on node `drive0`:

```
-> { "execute": "block-dirty-bitmap-enable",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0"
     }
   }
```

(continues on next page)

(continued from previous page)

```
<- { "return": {} }
```

Enabling: block-dirty-bitmap-disable

block-dirty-bitmap-disable:

"Disables" a bitmap, setting the `recording` bit to false, causing further writes to begin being ignored. `+busy` bitmaps cannot be disabled.

> **Warning:** This is potentially dangerous: QEMU makes no effort to stop any writes if there are disabled bitmaps on a node, and will not mark any disabled bitmaps as `+inconsistent` if any such writes do happen. Backups made from such bitmaps will not be able to be used to reconstruct a coherent image.

- Disabling a bitmap may be useful for examining which sectors of a disk changed during a specific time period, or for explicit management of differential backup windows.
- Persistent disabled bitmaps will remember their `-recording` status on load.

Example

To set `-recording` on bitmap `bitmap0` on node `drive0`:

```
-> { "execute": "block-dirty-bitmap-disable",
     "arguments": {
       "node": "drive0",
       "name": "bitmap0"
     }
   }

<- { "return": {} }
```

Merging, Copying: block-dirty-bitmap-merge

block-dirty-bitmap-merge:

Merges one or more bitmaps into a target bitmap. For any segment that is dirty in any one source bitmap, the target bitmap will mark that segment dirty.

- Merge takes one or more bitmaps as a source and merges them together into a single destination, such that any segment marked as dirty in any source bitmap(s) will be marked dirty in the destination bitmap.
- Merge does not create the destination bitmap if it does not exist. A blank bitmap can be created beforehand to achieve the same effect.
- The destination is not cleared prior to merge, so subsequent merge operations will continue to cumulatively mark more segments as dirty.
- If the merge operation should fail, the destination bitmap is guaranteed to be unmodified. The operation may fail if the source or destination bitmaps are busy, or have different granularities.

- Bitmaps can only be merged on the same node. There is only one "node" argument, so all bitmaps must be attached to that same node.
- Copy can be achieved by merging from a single source to an empty destination.

Example

Merge the data from `bitmap0` into the bitmap `new_bitmap` on node `drive0`. If `new_bitmap` was empty prior to this command, this achieves a copy.

```
-> { "execute": "block-dirty-bitmap-merge",
     "arguments": {
       "node": "drive0",
       "target": "new_bitmap",
       "bitmaps": [ "bitmap0" ]
     }
   }

<- { "return": {} }
```

Querying: query-block

query-block:

Not strictly a bitmaps command, but will return information about any bitmaps attached to nodes serving as the root for guest devices.

- The "inconsistent" bit will not appear when it is false, appearing only when the value is true to indicate there is a problem.

Example

Query the block sub-system of QEMU. The following json has trimmed irrelevant keys from the response to highlight only the bitmap-relevant portions of the API. This result highlights a bitmap `bitmap0` attached to the root node of device `drive0`.

```
-> {
     "execute": "query-block",
     "arguments": {}
   }

<- {
     "return": [ {
       "dirty-bitmaps": [ {
         "status": "active",
         "count": 0,
         "busy": false,
         "name": "bitmap0",
         "persistent": false,
         "recording": true,
         "granularity": 65536
       } ],
       "device": "drive0",
     } ]
   }
```

4.1.6 Bitmap Persistence

As outlined in *Supported Image Formats*, QEMU can persist bitmaps to qcow2 files. Demonstrated in *Creation: block-dirty-bitmap-add*, passing `persistent: true` to `block-dirty-bitmap-add` will persist that bitmap to disk.

Persistent bitmaps will be automatically loaded into memory upon load, and will be written back to disk upon close. Their usage should be mostly transparent.

However, if QEMU does not get a chance to close the file cleanly, the bitmap will be marked as `+inconsistent` at next load and considered unsafe to use for any operation. At this point, the only valid operation on such bitmaps is `block-dirty-bitmap-remove`.

Losing a bitmap in this way does not invalidate any existing backups that have been made from this bitmap, but no further backups will be able to be issued for this chain.

4.1.7 Transactions

Transactions are a QMP feature that allows you to submit multiple QMP commands at once, being guaranteed that they will all succeed or fail atomically, together. The interaction of bitmaps and transactions are demonstrated below.

See transaction in the QMP reference for more details.

Justification

Bitmaps can generally be modified at any time, but certain operations often only make sense when paired directly with other commands. When a VM is paused, it's easy to ensure that no guest writes occur between individual QMP commands. When a VM is running, this is difficult to accomplish with individual QMP commands that may allow guest writes to occur inbetween each command.

For example, using only individual QMP commands, we could:

1. Boot the VM in a paused state.
2. Create a full drive backup of drive0.
3. Create a new bitmap attached to drive0, confident that nothing has been written to drive0 in the meantime.
4. Resume execution of the VM.
5. At a later point, issue incremental backups from `bitmap0`.

At this point, the bitmap and drive backup would be correctly in sync, and incremental backups made from this point forward would be correctly aligned to the full drive backup.

This is not particularly useful if we decide we want to start incremental backups after the VM has been running for a while, for which we would want to perform actions such as the following:

1. Boot the VM and begin execution.
2. Using a single transaction, perform the following operations:
   - Create `bitmap0`.
   - Create a full drive backup of `drive0`.
3. At a later point, issue incremental backups from `bitmap0`.

Note: As a consideration, if `bitmap0` is created prior to the full drive backup, incremental backups can still be authored from this bitmap, but they will copy extra segments reflecting writes that occurred prior to the backup operation. Transactions allow us to narrow critical points in time to reduce waste, or, in the other direction, to ensure that no segments are omitted.

Supported Bitmap Transactions

- `block-dirty-bitmap-add`
- `block-dirty-bitmap-clear`
- `block-dirty-bitmap-enable`
- `block-dirty-bitmap-disable`
- `block-dirty-bitmap-merge`

The usages for these commands are identical to their respective QMP commands, but see the sections below for concrete examples.

4.1.8 Incremental Backups - Push Model

Incremental backups are simply partial disk images that can be combined with other partial disk images on top of a base image to reconstruct a full backup from the point in time at which the incremental backup was issued.

The "Push Model" here references the fact that QEMU is "pushing" the modified blocks out to a destination. We will be using the drive-backup and blockdev-backup QMP commands to create both full and incremental backups.

Both of these commands are jobs, which have their own QMP API for querying and management documented in Background jobs.

Example: New Incremental Backup Anchor Point

As outlined in the Transactions - *Justification* section, perhaps we want to create a new incremental backup chain attached to a drive.

This example creates a new, full backup of "drive0" and accompanies it with a new, empty bitmap that records writes from this point in time forward.

Note: Any new writes that happen after this command is issued, even while the backup job runs, will be written locally and not to the backup destination. These writes will be recorded in the bitmap accordingly.

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
         {
           "type": "block-dirty-bitmap-add",
           "data": {
             "node": "drive0",
             "name": "bitmap0"
           }
```

(continues on next page)

(continued from previous page)

```
          },
          {
            "type": "drive-backup",
            "data": {
              "device": "drive0",
              "target": "/path/to/drive0.full.qcow2",
              "sync": "full",
              "format": "qcow2"
            }
          }
        ]
      }
    }

<- { "return": {} }

<- {
     "timestamp": {
       "seconds": 1555436945,
       "microseconds": 179620
     },
     "data": {
       "status": "created",
       "id": "drive0"
     },
     "event": "JOB_STATUS_CHANGE"
   }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

<- {
     "timestamp": {...},
     "data": {
       "status": "concluded",
       "id": "drive0"
     },
     "event": "JOB_STATUS_CHANGE"
   }

<- {
     "timestamp": {...},
     "data": {
       "status": "null",
       "id": "drive0"
     },
```

(continues on next page)

(continued from previous page)

```
    "event": "JOB_STATUS_CHANGE"
  }
```

A full explanation of the job transition semantics and the JOB_STATUS_CHANGE event are beyond the scope of this document and will be omitted in all subsequent examples; above, several more events have been omitted for brevity.

Note: Subsequent examples will omit all events except BLOCK_JOB_COMPLETED except where necessary to illustrate workflow differences.

Omitted events and json objects will be represented by ellipses: `...`

Example: Resetting an Incremental Backup Anchor Point

If we want to start a new backup chain with an existing bitmap, we can also use a transaction to reset the bitmap while making a new full backup:

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
       {
         "type": "block-dirty-bitmap-clear",
         "data": {
           "node": "drive0",
           "name": "bitmap0"
         }
       },
       {
         "type": "drive-backup",
         "data": {
           "device": "drive0",
           "target": "/path/to/drive0.new_full.qcow2",
           "sync": "full",
           "format": "qcow2"
         }
       }
     ]
   }
 }

<- { "return": {} }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
```

(continues on next page)

(continued from previous page)

```
    }

...
```

The result of this example is identical to the first, but we clear an existing bitmap instead of adding a new one.

Tip: In both of these examples, "bitmap0" is tied conceptually to the creation of new, full backups. This relationship is not saved or remembered by QEMU; it is up to the operator or management layer to remember which bitmaps are associated with which backups.

Example: First Incremental Backup

1. Create a full backup and sync it to a dirty bitmap using any method:
   - Either of the two live backup method demonstrated above,
   - Using QMP commands with the VM paused as in the *Justification* section, or
   - With the VM offline, manually copy the image and start the VM in a paused state, careful to add a new bitmap before the VM begins execution.

   Whichever method is chosen, let's assume that at the end of this step:
   - The full backup is named `drive0.full.qcow2`.
   - The bitmap we created is named `bitmap0`, attached to `drive0`.
2. Create a destination image for the incremental backup that utilizes the full backup as a backing image.
   - Let's assume the new incremental image is named `drive0.inc0.qcow2`:

   ```
   $ qemu-img create -f qcow2 drive0.inc0.qcow2 \
     -b drive0.full.qcow2 -F qcow2
   ```
3. Issue an incremental backup command:

   ```
   -> {
        "execute": "drive-backup",
        "arguments": {
          "device": "drive0",
          "bitmap": "bitmap0",
          "target": "drive0.inc0.qcow2",
          "format": "qcow2",
          "sync": "incremental",
          "mode": "existing"
        }
      }

   <- { "return": {} }

   ...

   <- {
        "timestamp": {...},
        "data": {
          "device": "drive0",
   ```

(continues on next page)

(continued from previous page)

```
        "type": "backup",
        "speed": 0,
        "len": 68719476736,
        "offset": 68719476736
      },
      "event": "BLOCK_JOB_COMPLETED"
    }

...
```

This copies any blocks modified since the full backup was created into the `drive0.inc0.qcow2` file. During the operation, `bitmap0` is marked `+busy`. If the operation is successful, `bitmap0` will be cleared to reflect the "incremental" backup regimen, which only copies out new changes from each incremental backup.

Note: Any new writes that occur after the backup operation starts do not get copied to the destination. The backup's "point in time" is when the backup starts, not when it ends. These writes are recorded in a special bitmap that gets re-added to bitmap0 when the backup ends so that the next incremental backup can copy them out.

Example: Second Incremental Backup

1. Create a new destination image for the incremental backup that points to the previous one, e.g.: `drive0.inc1.qcow2`

```
$ qemu-img create -f qcow2 drive0.inc1.qcow2 \
  -b drive0.inc0.qcow2 -F qcow2
```

2. Issue a new incremental backup command. The only difference here is that we have changed the target image below.

```
-> {
     "execute": "drive-backup",
     "arguments": {
       "device": "drive0",
       "bitmap": "bitmap0",
       "target": "drive0.inc1.qcow2",
       "format": "qcow2",
       "sync": "incremental",
       "mode": "existing"
     }
   }

<- { "return": {} }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
```

(continues on next page)

(continued from previous page)

```
    },
    "event": "BLOCK_JOB_COMPLETED"
  }

...
```

Because the first incremental backup from the previous example completed successfully, `bitmap0` was synchronized with `drive0.inc0.qcow2`. Here, we use `bitmap0` again to create a new incremental backup that targets the previous one, creating a chain of three images:

Diagram

```
+-------------------+   +-------------------+   +-------------------+
| drive0.full.qcow2 |<--| drive0.inc0.qcow2 |<--| drive0.inc1.qcow2 |
+-------------------+   +-------------------+   +-------------------+
```

Each new incremental backup re-synchronizes the bitmap to the latest backup authored, allowing a user to continue to "consume" it to create new backups on top of an existing chain.

In the above diagram, neither drive0.inc1.qcow2 nor drive0.inc0.qcow2 are complete images by themselves, but rely on their backing chain to reconstruct a full image. The dependency terminates with each full backup.

Each backup in this chain remains independent, and is unchanged by new entries made later in the chain. For instance, drive0.inc0.qcow2 remains a perfectly valid backup of the disk as it was when that backup was issued.

Example: Incremental Push Backups without Backing Files

Backup images are best kept off-site, so we often will not have the preceding backups in a chain available to link against. This is not a problem at backup time; we simply do not set the backing image when creating the destination image:

1. Create a new destination image with no backing file set. We will need to specify the size of the base image, because the backing file isn't available for QEMU to use to determine it.

```
$ qemu-img create -f qcow2 drive0.inc2.qcow2 64G
```

Note: Alternatively, you can omit `mode: "existing"` from the push backup commands to have QEMU create an image without a backing file for you, but you lose control over format options like compatibility and preallocation presets.

2. Issue a new incremental backup command. Apart from the new destination image, there is no difference from the last two examples.

```
-> {
     "execute": "drive-backup",
     "arguments": {
       "device": "drive0",
       "bitmap": "bitmap0",
       "target": "drive0.inc2.qcow2",
       "format": "qcow2",
       "sync": "incremental",
       "mode": "existing"
```

(continues on next page)

(continued from previous page)

```
      }
    }

<- { "return": {} }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

...
```

The only difference from the perspective of the user is that you will need to set the backing image when attempting to restore the backup:

```
$ qemu-img rebase drive0.inc2.qcow2 \
  -u -b drive0.inc1.qcow2
```

This uses the "unsafe" rebase mode to simply set the backing file to a file that isn't present.

It is also possible to use `--image-opts` to specify the entire backing chain by hand as an ephemeral property at runtime, but that is beyond the scope of this document.

Example: Multi-drive Incremental Backup

Assume we have a VM with two drives, "drive0" and "drive1" and we wish to back both of them up such that the two backups represent the same crash-consistent point in time.

1. For each drive, create an empty image:

```
$ qemu-img create -f qcow2 drive0.full.qcow2 64G
$ qemu-img create -f qcow2 drive1.full.qcow2 64G
```

2. Create a full (anchor) backup for each drive, with accompanying bitmaps:

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
         {
           "type": "block-dirty-bitmap-add",
           "data": {
             "node": "drive0",
             "name": "bitmap0"
           }
         },
         {
```

(continues on next page)

(continued from previous page)

```
            "type": "block-dirty-bitmap-add",
            "data": {
              "node": "drive1",
              "name": "bitmap0"
            }
          },
          {
            "type": "drive-backup",
            "data": {
              "device": "drive0",
              "target": "/path/to/drive0.full.qcow2",
              "sync": "full",
              "format": "qcow2"
            }
          },
          {
            "type": "drive-backup",
            "data": {
              "device": "drive1",
              "target": "/path/to/drive1.full.qcow2",
              "sync": "full",
              "format": "qcow2"
            }
          }
        ]
      }
    }

<- { "return": {} }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive1",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
     "event": "BLOCK_JOB_COMPLETED"
   }
```

(continues on next page)

(continued from previous page)

```
...
```

3. Later, create new destination images for each of the incremental backups that point to their respective full backups:

```
$ qemu-img create -f qcow2 drive0.inc0.qcow2 \
  -b drive0.full.qcow2 -F qcow2
$ qemu-img create -f qcow2 drive1.inc0.qcow2 \
  -b drive1.full.qcow2 -F qcow2
```

4. Issue a multi-drive incremental push backup transaction:

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
         {
           "type": "drive-backup",
           "data": {
             "device": "drive0",
             "bitmap": "bitmap0",
             "format": "qcow2",
             "mode": "existing",
             "sync": "incremental",
             "target": "drive0.inc0.qcow2"
           }
         },
         {
           "type": "drive-backup",
           "data": {
             "device": "drive1",
             "bitmap": "bitmap0",
             "format": "qcow2",
             "mode": "existing",
             "sync": "incremental",
             "target": "drive1.inc0.qcow2"
           }
         },
       ]
     }
   }

<- { "return": {} }

...

<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 68719476736,
       "offset": 68719476736
     },
```

(continues on next page)

(continued from previous page)

```
      "event": "BLOCK_JOB_COMPLETED"
    }

...

<- {
      "timestamp": {...},
      "data": {
        "device": "drive1",
        "type": "backup",
        "speed": 0,
        "len": 68719476736,
        "offset": 68719476736
      },
      "event": "BLOCK_JOB_COMPLETED"
    }

...
```

4.1.9 Push Backup Errors & Recovery

In the event of an error that occurs after a push backup job is successfully launched, either by an individual QMP command or a QMP transaction, the user will receive a `BLOCK_JOB_COMPLETE` event with a failure message, accompanied by a `BLOCK_JOB_ERROR` event.

In the case of a job being cancelled, the user will receive a `BLOCK_JOB_CANCELLED` event instead of a pair of COMPLETE and ERROR events.

In either failure case, the bitmap used for the failed operation is not cleared. It will contain all of the dirty bits it did at the start of the operation, plus any new bits that got marked during the operation.

Effectively, the "point in time" that a bitmap is recording differences against is kept at the issuance of the last successful incremental backup, instead of being moved forward to the start of this now-failed backup.

Once the underlying problem is addressed (e.g. more storage space is allocated on the destination), the incremental backup command can be retried with the same bitmap.

Example: Individual Failures

Incremental Push Backup jobs that fail individually behave simply as described above. This example demonstrates the single-job failure case:

1. Create a target image:

```
$ qemu-img create -f qcow2 drive0.inc0.qcow2 \
  -b drive0.full.qcow2 -F qcow2
```

2. Attempt to create an incremental backup via QMP:

```
-> {
     "execute": "drive-backup",
     "arguments": {
       "device": "drive0",
       "bitmap": "bitmap0",
       "target": "drive0.inc0.qcow2",
```

(continues on next page)

(continued from previous page)

```
        "format": "qcow2",
        "sync": "incremental",
        "mode": "existing"
      }
    }

<- { "return": {} }
```

3. Receive a pair of events indicating failure:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "action": "report",
       "operation": "write"
     },
     "event": "BLOCK_JOB_ERROR"
   }

<- {
     "timestamp": {...},
     "data": {
       "speed": 0,
       "offset": 0,
       "len": 67108864,
       "error": "No space left on device",
       "device": "drive0",
       "type": "backup"
     },
     "event": "BLOCK_JOB_COMPLETED"
   }
```

4. Delete the failed image, and re-create it.

```
$ rm drive0.inc0.qcow2
$ qemu-img create -f qcow2 drive0.inc0.qcow2 \
  -b drive0.full.qcow2 -F qcow2
```

5. Retry the command after fixing the underlying problem, such as freeing up space on the backup volume:

```
-> {
     "execute": "drive-backup",
     "arguments": {
       "device": "drive0",
       "bitmap": "bitmap0",
       "target": "drive0.inc0.qcow2",
       "format": "qcow2",
       "sync": "incremental",
       "mode": "existing"
     }
   }

<- { "return": {} }
```

6. Receive confirmation that the job completed successfully:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 67108864,
       "offset": 67108864
     },
     "event": "BLOCK_JOB_COMPLETED"
   }
```

Example: Partial Transactional Failures

QMP commands like drive-backup conceptually only start a job, and so transactions containing these commands may succeed even if the job it created later fails. This might have surprising interactions with notions of how a "transaction" ought to behave.

This distinction means that on occasion, a transaction containing such job launching commands may appear to succeed and return success, but later individual jobs associated with the transaction may fail. It is possible that a management application may have to deal with a partial backup failure after a "successful" transaction.

If multiple backup jobs are specified in a single transaction, if one of those jobs fails, it will not interact with the other backup jobs in any way by default. The job(s) that succeeded will clear the dirty bitmap associated with the operation, but the job(s) that failed will not. It is therefore not safe to delete any incremental backups that were created successfully in this scenario, even though others failed.

This example illustrates a transaction with two backup jobs, where one fails and one succeeds:

1. Issue the transaction to start a backup of both drives.

```
-> {
     "execute": "transaction",
     "arguments": {
       "actions": [
       {
         "type": "drive-backup",
         "data": {
           "device": "drive0",
           "bitmap": "bitmap0",
           "format": "qcow2",
           "mode": "existing",
           "sync": "incremental",
           "target": "drive0.inc0.qcow2"
         }
       },
       {
         "type": "drive-backup",
         "data": {
           "device": "drive1",
           "bitmap": "bitmap0",
           "format": "qcow2",
           "mode": "existing",
           "sync": "incremental",
           "target": "drive1.inc0.qcow2"
         }
       }]
```

(continues on next page)

(continued from previous page)

```
      }
    }
```

2. Receive notice that the Transaction was accepted, and jobs were launched:

```
<- { "return": {} }
```

3. Receive notice that the first job has completed:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive0",
       "type": "backup",
       "speed": 0,
       "len": 67108864,
       "offset": 67108864
     },
     "event": "BLOCK_JOB_COMPLETED"
   }
```

4. Receive notice that the second job has failed:

```
<- {
     "timestamp": {...},
     "data": {
       "device": "drive1",
       "action": "report",
       "operation": "read"
     },
     "event": "BLOCK_JOB_ERROR"
   }

...

<- {
     "timestamp": {...},
     "data": {
       "speed": 0,
       "offset": 0,
       "len": 67108864,
       "error": "Input/output error",
       "device": "drive1",
       "type": "backup"
     },
     "event": "BLOCK_JOB_COMPLETED"
   }
```

At the conclusion of the above example, `drive0.inc0.qcow2` is valid and must be kept, but `drive1.inc0.qcow2` is incomplete and should be deleted. If a VM-wide incremental backup of all drives at a point-in-time is to be made, new backups for both drives will need to be made, taking into account that a new incremental backup for drive0 needs to be based on top of `drive0.inc0.qcow2`.

For this example, an incremental backup for `drive0` was created, but not for `drive1`. The last VM-wide crash-consistent backup that is available in this case is the full backup:

```
[drive0.full.qcow2] <-- [drive0.inc0.qcow2]
[drive1.full.qcow2]
```

To repair this, issue a new incremental backup across both drives. The result will be backup chains that resemble the following:

```
[drive0.full.qcow2] <-- [drive0.inc0.qcow2] <-- [drive0.inc1.qcow2]
[drive1.full.qcow2] <-------------------------- [drive1.inc1.qcow2]
```

Example: Grouped Completion Mode

While jobs launched by transactions normally complete or fail individually, it's possible to instruct them to complete or fail together as a group. QMP transactions take an optional properties structure that can affect the behavior of the transaction.

The `completion-mode` transaction property can be either `individual` which is the default legacy behavior described above, or `grouped`, detailed below.

In `grouped` completion mode, no jobs will report success until all jobs are ready to report success. If any job fails, all other jobs will be cancelled.

Regardless of if a participating incremental backup job failed or was cancelled, their associated bitmaps will all be held at their existing points-in-time, as in individual failure cases.

Here's the same multi-drive backup scenario from *Example: Partial Transactional Failures*, but with the `grouped` completion-mode property applied:

1. Issue the multi-drive incremental backup transaction:

```
-> {
     "execute": "transaction",
     "arguments": {
       "properties": {
         "completion-mode": "grouped"
       },
       "actions": [
       {
         "type": "drive-backup",
         "data": {
           "device": "drive0",
           "bitmap": "bitmap0",
           "format": "qcow2",
           "mode": "existing",
           "sync": "incremental",
           "target": "drive0.inc0.qcow2"
         }
       },
       {
         "type": "drive-backup",
         "data": {
           "device": "drive1",
           "bitmap": "bitmap0",
           "format": "qcow2",
           "mode": "existing",
           "sync": "incremental",
           "target": "drive1.inc0.qcow2"
         }
```

(continues on next page)

(continued from previous page)

```
        }]
      }
    }
```

2. Receive notice that the Transaction was accepted, and jobs were launched:

```
<- { "return": {} }
```

3. Receive notification that the backup job for `drive1` has failed:

```
<- {
      "timestamp": {...},
      "data": {
        "device": "drive1",
        "action": "report",
        "operation": "read"
      },
      "event": "BLOCK_JOB_ERROR"
    }

<- {
      "timestamp": {...},
      "data": {
        "speed": 0,
        "offset": 0,
        "len": 67108864,
        "error": "Input/output error",
        "device": "drive1",
        "type": "backup"
      },
      "event": "BLOCK_JOB_COMPLETED"
    }
```

4. Receive notification that the job for `drive0` has been cancelled:

```
<- {
      "timestamp": {...},
      "data": {
        "device": "drive0",
        "type": "backup",
        "speed": 0,
        "len": 67108864,
        "offset": 16777216
      },
      "event": "BLOCK_JOB_CANCELLED"
    }
```

At the conclusion of *this* example, both jobs have been aborted due to a failure. Both destination images should be deleted and are no longer of use.

The transaction as a whole can simply be re-issued at a later time.

4.2 D-Bus

4.2.1 Introduction

QEMU may be running with various helper processes involved:

- vhost-user* processes (gpu, virtfs, input, etc…)
- TPM emulation (or other devices)
- user networking (slirp)
- network services (DHCP/DNS, samba/ftp etc)
- background tasks (compression, streaming etc)
- client UI
- admin & cli

Having several processes allows stricter security rules, as well as greater modularity.

While QEMU itself uses QMP as primary IPC (and Spice/VNC for remote display), D-Bus is the de facto IPC of choice on Unix systems. The wire format is machine friendly, good bindings exist for various languages, and there are various tools available.

Using a bus, helper processes can discover and communicate with each other easily, without going through QEMU. The bus topology is also easier to apprehend and debug than a mesh. However, it is wise to consider the security aspects of it.

4.2.2 Security

A QEMU D-Bus bus should be private to a single VM. Thus, only cooperative tasks are running on the same bus to serve the VM.

D-Bus, the protocol and standard, doesn't have mechanisms to enforce security between peers once the connection is established. Peers may have additional mechanisms to enforce security rules, based for example on UNIX credentials.

The daemon can control which peers can send/recv messages using various metadata attributes, however, this is alone is not generally sufficient to make the deployment secure. The semantics of the actual methods implemented using D-Bus are just as critical. Peers need to carefully validate any information they received from a peer with a different trust level.

dbus-daemon policy

dbus-daemon can enforce various policies based on the UID/GID of the processes that are connected to it. It is thus a good idea to run helpers as different UID from QEMU and set appropriate policies.

Depending on the use case, you may choose different scenarios:

- Everything the same UID
  - Convenient for developers
  - Improved reliability - crash of one part doens't take out entire VM
  - No security benefit over traditional QEMU, unless additional unless additional controls such as SELinux or AppArmor are applied
- Two UIDs, one for QEMU, one for dbus & helpers

    - Moderately improved user based security isolation

- Many UIDs, one for QEMU one for dbus and one for each helpers
    - Best user based security isolation
    - Complex to manager distinct UIDs needed for each VM

For example, to allow only `qemu` user to talk to `qemu-helper` `org.qemu.Helper1` service, a dbus-daemon policy may contain:

```
<policy user="qemu">
   <allow send_destination="org.qemu.Helper1"/>
   <allow receive_sender="org.qemu.Helper1"/>
</policy>

<policy user="qemu-helper">
   <allow own="org.qemu.Helper1"/>
</policy>
```

dbus-daemon can also perfom SELinux checks based on the security context of the source and the target. For example, `virtiofs_t` could be allowed to send a message to `svirt_t`, but `virtiofs_t` wouldn't be allowed to send a message to `virtiofs_t`.

See dbus-daemon man page for details.

4.2.3 Guidelines

When implementing new D-Bus interfaces, it is recommended to follow the "D-Bus API Design Guidelines": https://dbus.freedesktop.org/doc/dbus-api-design.html

The "org.qemu.*" prefix is reserved for services implemented & distributed by the QEMU project.

4.2.4 QEMU Interfaces

D-Bus VMState

4.3 D-Bus VMState

4.3.1 Introduction

The QEMU dbus-vmstate object's aim is to migrate helpers' data running on a QEMU D-Bus bus. (refer to the *D-Bus* document for some recommendations on D-Bus usage)

Upon migration, QEMU will go through the queue of `org.qemu.VMState1` D-Bus name owners and query their `Id`. It must be unique among the helpers.

It will then save arbitrary data of each Id to be transferred in the migration stream and restored/loaded at the corresponding destination helper.

For now, the data amount to be transferred is arbitrarily limited to 1Mb. The state must be saved quickly (a fraction of a second). (D-Bus imposes a time limit on reply anyway, and migration would fail if data isn't given quickly enough.)

dbus-vmstate object can be configured with the expected list of helpers by setting its `id-list` property, with a comma-separated `Id` list.

4.3.2 Interface

On object path `/org/qemu/VMState1`, the following `org.qemu.VMState1` interface should be implemented:

```
<interface name="org.qemu.VMState1">
  <property name="Id" type="s" access="read"/>
  <method name="Load">
    <arg type="ay" name="data" direction="in"/>
  </method>
  <method name="Save">
    <arg type="ay" name="data" direction="out"/>
  </method>
</interface>
```

"Id" property

A string that identifies the helper uniquely. (maximum 256 bytes including terminating NUL byte)

Note: The helper ID namespace is a separate namespace. In particular, it is not related to QEMU "id" used in -object/-device objects.

Load(in u8[] bytes) method

The method called on destination with the state to restore.

The helper may be initially started in a waiting state (with an –incoming argument for example), and it may resume on success.

An error may be returned to the caller.

Save(out u8[] bytes) method

The method called on the source to get the current state to be migrated. The helper should continue to run normally.

An error may be returned to the caller.

4.4 Live Block Device Operations

QEMU Block Layer currently (as of QEMU 2.9) supports four major kinds of live block device jobs – stream, commit, mirror, and backup. These can be used to manipulate disk image chains to accomplish certain tasks, namely: live copy data from backing files into overlays; shorten long disk image chains by merging data from overlays into backing files; live synchronize data from a disk image chain (including current active disk) to another target image; and point-in-time (and incremental) backups of a block device. Below is a description of the said block (QMP) primitives, and some (non-exhaustive list of) examples to illustrate their use.

Note: The file `qapi/block-core.json` in the QEMU source tree has the canonical QEMU API (QAPI) schema documentation for the QMP primitives discussed here.

Contents

4.4.1 Disk image backing chain notation

A simple disk image chain. (This can be created live using QMP `blockdev-snapshot-sync`, or offline via `qemu-img`):

```
                 (Live QEMU)
                      |
                      .
                      V

        [A] <----- [B]

(backing file)    (overlay)
```

The arrow can be read as: Image [A] is the backing file of disk image [B]. And live QEMU is currently writing to image [B], consequently, it is also referred to as the "active layer".

There are two kinds of terminology that are common when referring to files in a disk image backing chain:

(1) Directional: 'base' and 'top'. Given the simple disk image chain above, image [A] can be referred to as 'base', and image [B] as 'top'. (This terminology can be seen in in QAPI schema file, block-core.json.)

(2) Relational: 'backing file' and 'overlay'. Again, taking the same simple disk image chain from the above, disk image [A] is referred to as the backing file, and image [B] as overlay.

Throughout this document, we will use the relational terminology.

Important: The overlay files can generally be any format that supports a backing file, although QCOW2 is the preferred format and the one used in this document.

4.4.2 Brief overview of live block QMP primitives

The following are the four different kinds of live block operations that QEMU block layer supports.

(1) `block-stream`: Live copy of data from backing files into overlay files.

> **Note:** Once the 'stream' operation has finished, three things to note:
>
> (a) QEMU rewrites the backing chain to remove reference to the now-streamed and redundant backing file;
>
> (b) the streamed file *itself* won't be removed by QEMU, and must be explicitly discarded by the user;
>
> (c) the streamed file remains valid – i.e. further overlays can be created based on it. Refer the `block-stream` section further below for more details.

(2) `block-commit`: Live merge of data from overlay files into backing files (with the optional goal of removing the overlay file from the chain). Since QEMU 2.0, this includes "active `block-commit`" (i.e. merge the current active layer into the base image).

> **Note:** Once the 'commit' operation has finished, there are three things to note here as well:
>
> (a) QEMU rewrites the backing chain to remove reference to now-redundant overlay images that have been committed into a backing file;
>
> (b) the committed file *itself* won't be removed by QEMU – it ought to be manually removed;
>
> (c) however, unlike in the case of `block-stream`, the intermediate images will be rendered invalid – i.e. no more further overlays can be created based on them. Refer the `block-commit` section further below for more details.

(3) `drive-mirror` (and `blockdev-mirror`): Synchronize a running disk to another image.

(4) `drive-backup` (and `blockdev-backup`): Point-in-time (live) copy of a block device to a destination.

4.4.3 Interacting with a QEMU instance

To show some example invocations of command-line, we will use the following invocation of QEMU, with a QMP server running over UNIX socket:

```
$ ./qemu-system-x86_64 -display none -no-user-config \
    -M q35 -nodefaults -m 512 \
    -blockdev node-name=node-A,driver=qcow2,file.driver=file,file.node-name=file,file.
↪filename=./a.qcow2 \
    -device virtio-blk,drive=node-A,id=virtio0 \
    -monitor stdio -qmp unix:/tmp/qmp-sock,server,nowait
```

The -blockdev command-line option, used above, is available from QEMU 2.9 onwards. In the above invocation, notice the node-name parameter that is used to refer to the disk image a.qcow2 ('node-A') – this is a cleaner way to refer to a disk image (as opposed to referring to it by spelling out file paths). So, we will continue to designate a node-name to each further disk image created (either via blockdev-snapshot-sync, or blockdev-add) as part of the disk image chain, and continue to refer to the disks using their node-name (where possible, because block-commit does not yet, as of QEMU 2.9, accept node-name parameter) when performing various block operations.

To interact with the QEMU instance launched above, we will use the qmp-shell utility (located at: qemu/scripts/qmp, as part of the QEMU source directory), which takes key-value pairs for QMP commands. Invoke it as below (which will also print out the complete raw JSON syntax for reference – examples in the following sections):

```
$ ./qmp-shell -v -p /tmp/qmp-sock
(QEMU)
```

Note: In the event we have to repeat a certain QMP command, we will: for the first occurrence of it, show the qmp-shell invocation, *and* the corresponding raw JSON QMP syntax; but for subsequent invocations, present just the qmp-shell syntax, and omit the equivalent JSON output.

4.4.4 Example disk image chain

We will use the below disk image chain (and occasionally spelling it out where appropriate) when discussing various primitives:

```
[A] <-- [B] <-- [C] <-- [D]
```

Where [A] is the original base image; [B] and [C] are intermediate overlay images; image [D] is the active layer – i.e. live QEMU is writing to it. (The rule of thumb is: live QEMU will always be pointing to the rightmost image in a disk image chain.)

The above image chain can be created by invoking blockdev-snapshot-sync commands as following (which shows the creation of overlay image [B]) using the qmp-shell (our invocation also prints the raw JSON invocation of it):

```
(QEMU) blockdev-snapshot-sync node-name=node-A snapshot-file=b.qcow2 snapshot-node-
↪name=node-B format=qcow2
{
    "execute": "blockdev-snapshot-sync",
    "arguments": {
        "node-name": "node-A",
        "snapshot-file": "b.qcow2",
        "format": "qcow2",
        "snapshot-node-name": "node-B"
    }
}
```

Here, "node-A" is the name QEMU internally uses to refer to the base image [A] – it is the backing file, based on which the overlay image, [B], is created.

To create the rest of the overlay images, [C], and [D] (omitting the raw JSON output for brevity):

```
(QEMU) blockdev-snapshot-sync node-name=node-B snapshot-file=c.qcow2 snapshot-node-
↪name=node-C format=qcow2
(QEMU) blockdev-snapshot-sync node-name=node-C snapshot-file=d.qcow2 snapshot-node-
↪name=node-D format=qcow2
```

4.4.5 A note on points-in-time vs file names

In our disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

We have *three* points in time and an active layer:

- Point 1: Guest state when [B] was created is contained in file [A]
- Point 2: Guest state when [C] was created is contained in [A] + [B]
- Point 3: Guest state when [D] was created is contained in [A] + [B] + [C]
- Active layer: Current guest state is contained in [A] + [B] + [C] + [D]

Therefore, be aware with naming choices:

- Naming a file after the time it is created is misleading – the guest data for that point in time is *not* contained in that file (as explained earlier)
- Rather, think of files as a *delta* from the backing file

4.4.6 Live block streaming — `block-stream`

The `block-stream` command allows you to do live copy data from backing files into overlay images.

Given our original example disk image chain from earlier:

```
[A] <-- [B] <-- [C] <-- [D]
```

The disk image chain can be shortened in one of the following different ways (not an exhaustive list).

(1) Merge everything into the active layer: I.e. copy all contents from the base image, [A], and overlay images, [B] and [C], into [D], *while* the guest is running. The resulting chain will be a standalone image, [D] – with contents from [A], [B] and [C] merged into it (where live QEMU writes go to):

```
[D]
```

(2) Taking the same example disk image chain mentioned earlier, merge only images [B] and [C] into [D], the active layer. The result will be contents of images [B] and [C] will be copied into [D], and the backing file pointer of image [D] will be adjusted to point to image [A]. The resulting chain will be:

```
[A] <-- [D]
```

(3) Intermediate streaming (available since QEMU 2.8): Starting afresh with the original example disk image chain, with a total of four images, it is possible to copy contents from image [B] into image [C]. Once the copy is finished, image [B] can now be (optionally) discarded; and the backing file pointer of image [C] will be adjusted to point to [A]. I.e. after performing "intermediate streaming" of [B] into [C], the resulting image chain will be (where live QEMU is writing to [D]):

```
[A] <-- [C] <-- [D]
```

QMP invocation for `block-stream`

For *Case-1*, to merge contents of all the backing files into the active layer, where 'node-D' is the current active image (by default `block-stream` will flatten the entire chain); `qmp-shell` (and its corresponding JSON output):

```
(QEMU) block-stream device=node-D job-id=job0
{
    "execute": "block-stream",
    "arguments": {
        "device": "node-D",
        "job-id": "job0"
    }
}
```

For *Case-2*, merge contents of the images [B] and [C] into [D], where image [D] ends up referring to image [A] as its backing file:

```
(QEMU) block-stream device=node-D base-node=node-A job-id=job0
```

And for *Case-3*, of "intermediate" streaming", merge contents of images [B] into [C], where [C] ends up referring to [A] as its backing image:

```
(QEMU) block-stream device=node-C base-node=node-A job-id=job0
```

Progress of a `block-stream` operation can be monitored via the QMP command:

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
```

Once the `block-stream` operation has completed, QEMU will emit an event, `BLOCK_JOB_COMPLETED`. The intermediate overlays remain valid, and can now be (optionally) discarded, or retained to create further overlays based on them. Finally, the `block-stream` jobs can be restarted at anytime.

4.4.7 Live block commit — `block-commit`

The `block-commit` command lets you merge live data from overlay images into backing file(s). Since QEMU 2.0, this includes "live active commit" (i.e. it is possible to merge the "active layer", the right-most image in a disk image chain where live QEMU will be writing to, into the base image). This is analogous to `block-stream`, but in the opposite direction.

Again, starting afresh with our example disk image chain, where live QEMU is writing to the right-most image in the chain, [D]:

```
[A] <-- [B] <-- [C] <-- [D]
```

The disk image chain can be shortened in one of the following ways:

(1) Commit content from only image [B] into image [A]. The resulting chain is the following, where image [C] is adjusted to point at [A] as its new backing file:

```
[A] <-- [C] <-- [D]
```

(2) Commit content from images [B] and [C] into image [A]. The resulting chain, where image [D] is adjusted to point to image [A] as its new backing file:

```
[A] <-- [D]
```

(3) Commit content from images [B], [C], and the active layer [D] into image [A]. The resulting chain (in this case, a consolidated single image):

```
[A]
```

(4) Commit content from image only image [C] into image [B]. The resulting chain:

```
[A] <-- [B] <-- [D]
```

(5) Commit content from image [C] and the active layer [D] into image [B]. The resulting chain:

```
[A] <-- [B]
```

QMP invocation for `block-commit`

For *Case-1*, to merge contents only from image [B] into image [A], the invocation is as follows:

```
(QEMU) block-commit device=node-D base=a.qcow2 top=b.qcow2 job-id=job0
{
    "execute": "block-commit",
    "arguments": {
        "device": "node-D",
        "job-id": "job0",
        "top": "b.qcow2",
        "base": "a.qcow2"
    }
}
```

Once the above `block-commit` operation has completed, a `BLOCK_JOB_COMPLETED` event will be issued, and no further action is required. As the end result, the backing file of image [C] is adjusted to point to image [A], and the original 4-image chain will end up being transformed to:

```
[A] <-- [C] <-- [D]
```

Note: The intermediate image [B] is invalid (as in: no more further overlays based on it can be created).

Reasoning: An intermediate image after a 'stream' operation still represents that old point-in-time, and may be valid in that context. However, an intermediate image after a 'commit' operation no longer represents any point-in-time, and is invalid in any context.

However, *Case-3* (also called: "active `block-commit`") is a *two-phase* operation: In the first phase, the content from the active overlay, along with the intermediate overlays, is copied into the backing file (also called the base image). In the second phase, adjust the said backing file as the current active image – possible via issuing the command `block-job-complete`. Optionally, the `block-commit` operation can be cancelled by issuing the command `block-job-cancel`, but be careful when doing this.

Once the `block-commit` operation has completed, the event `BLOCK_JOB_READY` will be emitted, signalling that the synchronization has finished. Now the job can be gracefully completed by issuing the command `block-job-complete` – until such a command is issued, the 'commit' operation remains active.

The following is the flow for *Case-3* to convert a disk image chain such as this:

```
[A] <-- [B] <-- [C] <-- [D]
```

Into:

```
[A]
```

Where content from all the subsequent overlays, [B], and [C], including the active layer, [D], is committed back to [A] – which is where live QEMU is performing all its current writes).

Start the "active `block-commit`" operation:

```
(QEMU) block-commit device=node-D base=a.qcow2 top=d.qcow2 job-id=job0
{
    "execute": "block-commit",
    "arguments": {
        "device": "node-D",
        "job-id": "job0",
        "top": "d.qcow2",
        "base": "a.qcow2"
    }
}
```

Once the synchronization has completed, the event `BLOCK_JOB_READY` will be emitted.

Then, optionally query for the status of the active block operations. We can see the 'commit' job is now ready to be completed, as indicated by the line *"ready": true*:

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
{
    "return": [
        {
            "busy": false,
            "type": "commit",
            "len": 1376256,
            "paused": false,
            "ready": true,
            "io-status": "ok",
            "offset": 1376256,
            "device": "job0",
            "speed": 0
        }
    ]
}
```

Gracefully complete the 'commit' block device job:

```
(QEMU) block-job-complete device=job0
{
    "execute": "block-job-complete",
    "arguments": {
        "device": "job0"
    }
}
{
    "return": {}
}
```

Finally, once the above job is completed, an event `BLOCK_JOB_COMPLETED` will be emitted.

Note: The invocation for rest of the cases (2, 4, and 5), discussed in the previous section, is omitted for brevity.

4.4.8 Live disk synchronization — `drive-mirror` and `blockdev-mirror`

Synchronize a running disk image chain (all or part of it) to a target image.

Again, given our familiar disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

The `drive-mirror` (and its newer equivalent `blockdev-mirror`) allows you to copy data from the entire chain into a single target image (which can be located on a different host), [E].

Note: When you cancel an in-progress 'mirror' job *before* the source and target are synchronized, `block-job-cancel` will emit the event `BLOCK_JOB_CANCELLED`. However, note that if you cancel a 'mirror' job *after* it has indicated (via the event `BLOCK_JOB_READY`) that the source and target have reached synchronization, then the event emitted by `block-job-cancel` changes to `BLOCK_JOB_COMPLETED`.

Besides the 'mirror' job, the "active `block-commit`" is the only other block device job that emits the event `BLOCK_JOB_READY`. The rest of the block device jobs ('stream', "non-active `block-commit`", and 'backup') end automatically.

So there are two possible actions to take, after a 'mirror' job has emitted the event `BLOCK_JOB_READY`, indicating that the source and target have reached synchronization:

(1) Issuing the command `block-job-cancel` (after it emits the event `BLOCK_JOB_COMPLETED`) will create a point-in-time (which is at the time of *triggering* the cancel command) copy of the entire disk image chain (or only the top-most image, depending on the `sync` mode), contained in the target image [E]. One use case for this is live VM migration with non-shared storage.

(2) Issuing the command `block-job-complete` (after it emits the event `BLOCK_JOB_COMPLETED`) will adjust the guest device (i.e. live QEMU) to point to the target image, [E], causing all the new writes from this point on to happen there.

About synchronization modes: The synchronization mode determines *which* part of the disk image chain will be copied to the target. Currently, there are four different kinds:

(1) `full` – Synchronize the content of entire disk image chain to the target

(2) `top` – Synchronize only the contents of the top-most disk image in the chain to the target

(3) `none` – Synchronize only the new writes from this point on.

> **Note:** In the case of `drive-backup` (or `blockdev-backup`), the behavior of `none` synchronization mode is different. Normally, a `backup` job consists of two parts: Anything that is overwritten by the guest is first copied out to the backup, and in the background the whole image is copied from start to end. With `sync=none`, it's only the first part.

(4) `incremental` – Synchronize content that is described by the dirty bitmap

Note: Refer to the *Dirty Bitmaps and Incremental Backup* document in the QEMU source tree to learn about the detailed workings of the `incremental` synchronization mode.

QMP invocation for `drive-mirror`

To copy the contents of the entire disk image chain, from [A] all the way to [D], to a new target (`drive-mirror` will create the destination file, if it doesn't already exist), call it [E]:

```
(QEMU) drive-mirror device=node-D target=e.qcow2 sync=full job-id=job0
{
    "execute": "drive-mirror",
    "arguments": {
        "device": "node-D",
        "job-id": "job0",
        "target": "e.qcow2",
        "sync": "full"
    }
}
```

The `"sync": "full"`, from the above, means: copy the *entire* chain to the destination.

Following the above, querying for active block jobs will show that a 'mirror' job is "ready" to be completed (and QEMU will also emit an event, `BLOCK_JOB_READY`):

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
{
    "return": [
        {
            "busy": false,
            "type": "mirror",
            "len": 21757952,
            "paused": false,
            "ready": true,
            "io-status": "ok",
            "offset": 21757952,
            "device": "job0",
            "speed": 0
        }
    ]
}
```

And, as noted in the previous section, there are two possible actions at this point:

(a) Create a point-in-time snapshot by ending the synchronization. The point-in-time is at the time of *ending* the sync. (The result of the following being: the target image, [E], will be populated with content from the entire chain, [A] to [D]):

```
(QEMU) block-job-cancel device=job0
{
    "execute": "block-job-cancel",
    "arguments": {
        "device": "job0"
    }
}
```

(b) Or, complete the operation and pivot the live QEMU to the target copy:

```
(QEMU) block-job-complete device=job0
```

In either of the above cases, if you once again run the *query-block-jobs* command, there should not be any active block operation.

Comparing 'commit' and 'mirror': In both then cases, the overlay images can be discarded. However, with 'commit', the *existing* base image will be modified (by updating it with contents from overlays); while in the case of 'mirror', a *new* target image is populated with the data from the disk image chain.

QMP invocation for live storage migration with `drive-mirror` + NBD

Live storage migration (without shared storage setup) is one of the most common use-cases that takes advantage of the `drive-mirror` primitive and QEMU's built-in Network Block Device (NBD) server. Here's a quick walk-through of this setup.

Given the disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

Instead of copying content from the entire chain, synchronize *only* the contents of the *top*-most disk image (i.e. the active layer), [D], to a target, say, [TargetDisk].

Important: The destination host must already have the contents of the backing chain, involving images [A], [B], and [C], visible via other means – whether by `cp`, `rsync`, or by some storage array-specific command.)

Sometimes, this is also referred to as "shallow copy" – because only the "active layer", and not the rest of the image chain, is copied to the destination.

Note: In this example, for the sake of simplicity, we'll be using the same `localhost` as both source and destination.

As noted earlier, on the destination host the contents of the backing chain – from images [A] to [C] – are already expected to exist in some form (e.g. in a file called, `Contents-of-A-B-C.qcow2`). Now, on the destination host, let's create a target overlay image (with the image `Contents-of-A-B-C.qcow2` as its backing file), to which the contents of image [D] (from the source QEMU) will be mirrored to:

```
$ qemu-img create -f qcow2 -b ./Contents-of-A-B-C.qcow2 \
    -F qcow2 ./target-disk.qcow2
```

And start the destination QEMU (we already have the source QEMU running – discussed in the section: *Interacting with a QEMU instance*) instance, with the following invocation. (As noted earlier, for simplicity's sake, the destination QEMU is started on the same host, but it could be located elsewhere):

```
$ ./qemu-system-x86_64 -display none -no-user-config \
    -M q35 -nodefaults -m 512 \
    -blockdev node-name=node-TargetDisk,driver=qcow2,file.driver=file,file.node-
↪name=file,file.filename=./target-disk.qcow2 \
    -device virtio-blk,drive=node-TargetDisk,id=virtio0 \
    -S -monitor stdio -qmp unix:./qmp-sock2,server,nowait \
    -incoming tcp:localhost:6666
```

Given the disk image chain on source QEMU:

```
[A] <-- [B] <-- [C] <-- [D]
```

On the destination host, it is expected that the contents of the chain `[A] <-- [B] <-- [C]` are *already* present, and therefore copy *only* the content of image [D].

(1) [On *destination* QEMU] As part of the first step, start the built-in NBD server on a given host (local host, represented by `::`)and port:

```
(QEMU) nbd-server-start addr={"type":"inet","data":{"host":"::","port":"49153"}}
{
    "execute": "nbd-server-start",
    "arguments": {
        "addr": {
            "data": {
                "host": "::",
                "port": "49153"
            },
            "type": "inet"
        }
    }
}
```

(2) [On *destination* QEMU] And export the destination disk image using QEMU's built-in NBD server:

```
(QEMU) nbd-server-add device=node-TargetDisk writable=true
{
    "execute": "nbd-server-add",
    "arguments": {
        "device": "node-TargetDisk"
    }
}
```

(3) [On *source* QEMU] Then, invoke `drive-mirror` (NB: since we're running `drive-mirror` with `mode=existing` (meaning: synchronize to a pre-created file, therefore 'existing', file on the target host), with the synchronization mode as 'top' (`"sync: "top"`):

```
(QEMU) drive-mirror device=node-D target=nbd:localhost:49153:exportname=node-
↪TargetDisk sync=top mode=existing job-id=job0
{
    "execute": "drive-mirror",
    "arguments": {
        "device": "node-D",
        "mode": "existing",
        "job-id": "job0",
        "target": "nbd:localhost:49153:exportname=node-TargetDisk",
        "sync": "top"
    }
}
```

(4) [On *source* QEMU] Once `drive-mirror` copies the entire data, and the event `BLOCK_JOB_READY` is emitted, issue `block-job-cancel` to gracefully end the synchronization, from source QEMU:

```
(QEMU) block-job-cancel device=job0
{
    "execute": "block-job-cancel",
    "arguments": {
        "device": "job0"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

(5) [On *destination* QEMU] Then, stop the NBD server:

```
(QEMU) nbd-server-stop
{
    "execute": "nbd-server-stop",
    "arguments": {}
}
```

(6) [On *destination* QEMU] Finally, resume the guest vCPUs by issuing the QMP command *cont*:

```
(QEMU) cont
{
    "execute": "cont",
    "arguments": {}
}
```

Note: Higher-level libraries (e.g. libvirt) automate the entire above process (although note that libvirt does not allow same-host migrations to localhost for other reasons).

Notes on `blockdev-mirror`

The `blockdev-mirror` command is equivalent in core functionality to `drive-mirror`, except that it operates at node-level in a BDS graph.

Also: for `blockdev-mirror`, the 'target' image needs to be explicitly created (using `qemu-img`) and attach it to live QEMU via `blockdev-add`, which assigns a name to the to-be created target node.

E.g. the sequence of actions to create a point-in-time backup of an entire disk image chain, to a target, using `blockdev-mirror` would be:

(0) Create the QCOW2 overlays, to arrive at a backing chain of desired depth

(1) Create the target image (using `qemu-img`), say, `e.qcow2`

(2) Attach the above created file (`e.qcow2`), run-time, using `blockdev-add` to QEMU

(3) Perform `blockdev-mirror` (use `"sync": "full"` to copy the entire chain to the target). And notice the event `BLOCK_JOB_READY`

(4) Optionally, query for active block jobs, there should be a 'mirror' job ready to be completed

(5) Gracefully complete the 'mirror' block device job, and notice the the event `BLOCK_JOB_COMPLETED`

(6) Shutdown the guest by issuing the QMP `quit` command so that caches are flushed

(7) Then, finally, compare the contents of the disk image chain, and the target copy with `qemu-img compare`. You should notice: "Images are identical"

QMP invocation for `blockdev-mirror`

Given the disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

To copy the contents of the entire disk image chain, from [A] all the way to [D], to a new target, call it [E]. The following is the flow.

Create the overlay images, [B], [C], and [D]:

```
(QEMU) blockdev-snapshot-sync node-name=node-A snapshot-file=b.qcow2 snapshot-node-
↪name=node-B format=qcow2
(QEMU) blockdev-snapshot-sync node-name=node-B snapshot-file=c.qcow2 snapshot-node-
↪name=node-C format=qcow2
(QEMU) blockdev-snapshot-sync node-name=node-C snapshot-file=d.qcow2 snapshot-node-
↪name=node-D format=qcow2
```

Create the target image, [E]:

```
$ qemu-img create -f qcow2 e.qcow2 39M
```

Add the above created target image to QEMU, via `blockdev-add`:

```
(QEMU) blockdev-add driver=qcow2 node-name=node-E file={"driver":"file","filename":"e.
↪qcow2"}
{
    "execute": "blockdev-add",
    "arguments": {
        "node-name": "node-E",
        "driver": "qcow2",
        "file": {
            "driver": "file",
            "filename": "e.qcow2"
        }
    }
}
```

Perform `blockdev-mirror`, and notice the event `BLOCK_JOB_READY`:

```
(QEMU) blockdev-mirror device=node-B target=node-E sync=full job-id=job0
{
    "execute": "blockdev-mirror",
    "arguments": {
        "device": "node-D",
        "job-id": "job0",
        "target": "node-E",
        "sync": "full"
    }
}
```

Query for active block jobs, there should be a 'mirror' job ready:

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
{
    "return": [
        {
```

(continues on next page)

(continued from previous page)

```
            "busy": false,
            "type": "mirror",
            "len": 21561344,
            "paused": false,
            "ready": true,
            "io-status": "ok",
            "offset": 21561344,
            "device": "job0",
            "speed": 0
        }
    ]
}
```

Gracefully complete the block device job operation, and notice the event `BLOCK_JOB_COMPLETED`:

```
(QEMU) block-job-complete device=job0
{
    "execute": "block-job-complete",
    "arguments": {
        "device": "job0"
    }
}
{
    "return": {}
}
```

Shutdown the guest, by issuing the `quit` QMP command:

```
(QEMU) quit
{
    "execute": "quit",
    "arguments": {}
}
```

4.4.9 Live disk backup — `drive-backup` and `blockdev-backup`

The `drive-backup` (and its newer equivalent `blockdev-backup`) allows you to create a point-in-time snapshot.

In this case, the point-in-time is when you *start* the `drive-backup` (or its newer equivalent `blockdev-backup`) command.

QMP invocation for `drive-backup`

Yet again, starting afresh with our example disk image chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

To create a target image [E], with content populated from image [A] to [D], from the above chain, the following is the syntax. (If the target image does not exist, `drive-backup` will create it):

```
(QEMU) drive-backup device=node-D sync=full target=e.qcow2 job-id=job0
{
    "execute": "drive-backup",
    "arguments": {
```

(continues on next page)

(continued from previous page)

```
        "device": "node-D",
        "job-id": "job0",
        "sync": "full",
        "target": "e.qcow2"
    }
}
```

Once the above `drive-backup` has completed, a `BLOCK_JOB_COMPLETED` event will be issued, indicating the live block device job operation has completed, and no further action is required.

Notes on `blockdev-backup`

The `blockdev-backup` command is equivalent in functionality to `drive-backup`, except that it operates at node-level in a Block Driver State (BDS) graph.

E.g. the sequence of actions to create a point-in-time backup of an entire disk image chain, to a target, using `blockdev-backup` would be:

(0) Create the QCOW2 overlays, to arrive at a backing chain of desired depth

(1) Create the target image (using `qemu-img`), say, `e.qcow2`

(2) Attach the above created file (`e.qcow2`), run-time, using `blockdev-add` to QEMU

(3) Perform `blockdev-backup` (use `"sync": "full"` to copy the entire chain to the target). And notice the event `BLOCK_JOB_COMPLETED`

(4) Shutdown the guest, by issuing the QMP `quit` command, so that caches are flushed

(5) Then, finally, compare the contents of the disk image chain, and the target copy with `qemu-img compare`. You should notice: “Images are identical”

The following section shows an example QMP invocation for `blockdev-backup`.

QMP invocation for `blockdev-backup`

Given a disk image chain of depth 1 where image [B] is the active overlay (live QEMU is writing to it):

```
[A] <-- [B]
```

The following is the procedure to copy the content from the entire chain to a target image (say, [E]), which has the full content from [A] and [B].

Create the overlay [B]:

```
(QEMU) blockdev-snapshot-sync node-name=node-A snapshot-file=b.qcow2 snapshot-node-
↪name=node-B format=qcow2
{
    "execute": "blockdev-snapshot-sync",
    "arguments": {
        "node-name": "node-A",
        "snapshot-file": "b.qcow2",
        "format": "qcow2",
        "snapshot-node-name": "node-B"
    }
}
```

Create a target image that will contain the copy:

```
$ qemu-img create -f qcow2 e.qcow2 39M
```

Then add it to QEMU via `blockdev-add`:

```
(QEMU) blockdev-add driver=qcow2 node-name=node-E file={"driver":"file","filename":"e.
↪qcow2"}
{
    "execute": "blockdev-add",
    "arguments": {
        "node-name": "node-E",
        "driver": "qcow2",
        "file": {
            "driver": "file",
            "filename": "e.qcow2"
        }
    }
}
```

Then invoke `blockdev-backup` to copy the contents from the entire image chain, consisting of images [A] and [B] to the target image 'e.qcow2':

```
(QEMU) blockdev-backup device=node-B target=node-E sync=full job-id=job0
{
    "execute": "blockdev-backup",
    "arguments": {
        "device": "node-B",
        "job-id": "job0",
        "target": "node-E",
        "sync": "full"
    }
}
```

Once the above 'backup' operation has completed, the event, `BLOCK_JOB_COMPLETED` will be emitted, signalling successful completion.

Next, query for any active block device jobs (there should be none):

```
(QEMU) query-block-jobs
{
    "execute": "query-block-jobs",
    "arguments": {}
}
```

Shutdown the guest:

```
(QEMU) quit
{
        "execute": "quit",
            "arguments": {}
}
        "return": {}
}
```

Note: The above step is really important; if forgotten, an error, "Failed to get shared "write" lock on e.qcow2", will be thrown when you do `qemu-img compare` to verify the integrity of the disk image with the backup content.

The end result will be the image 'e.qcow2' containing a point-in-time backup of the disk image chain – i.e. contents from images [A] and [B] at the time the `blockdev-backup` command was initiated.

One way to confirm the backup disk image contains the identical content with the disk image chain is to compare the backup and the contents of the chain, you should see "Images are identical". (NB: this is assuming QEMU was launched with `-S` option, which will not start the CPUs at guest boot up):

```
$ qemu-img compare b.qcow2 e.qcow2
Warning: Image size mismatch!
Images are identical.
```

NOTE: The "Warning: Image size mismatch!" is expected, as we created the target image (e.qcow2) with 39M size.

4.5 Persistent reservation helper protocol

QEMU's SCSI passthrough devices, `scsi-block` and `scsi-generic`, can delegate implementation of persistent reservations to an external (and typically privileged) program. Persistent Reservations allow restricting access to block devices to specific initiators in a shared storage setup.

For a more detailed reference please refer to the SCSI Primary Commands standard, specifically the section on Reservations and the "PERSISTENT RESERVE IN" and "PERSISTENT RESERVE OUT" commands.

This document describes the socket protocol used between QEMU's `pr-manager-helper` object and the external program.

Contents

4.5.1 Connection and initialization

All data transmitted on the socket is big-endian.

After connecting to the helper program's socket, the helper starts a simple feature negotiation process by writing four bytes corresponding to the features it exposes (`supported_features`). QEMU reads it, then writes four bytes corresponding to the desired features of the helper program (`requested_features`).

If a bit is 1 in `requested_features` and 0 in `supported_features`, the corresponding feature is not supported by the helper and the connection is closed. On the other hand, it is acceptable for a bit to be 0 in `requested_features` and 1 in `supported_features`; in this case, the helper will not enable the feature.

Right now no feature is defined, so the two parties always write four zero bytes.

4.5.2 Command format

It is invalid to send multiple commands concurrently on the same socket. It is however possible to connect multiple sockets to the helper and send multiple commands to the helper for one or more file descriptors.

A command consists of a request and a response. A request consists of a 16-byte SCSI CDB. A file descriptor must be passed to the helper together with the SCSI CDB using ancillary data.

The CDB has the following limitations:

- the command (stored in the first byte) must be one of 0x5E (PERSISTENT RESERVE IN) or 0x5F (PERSISTENT RESERVE OUT).
- the allocation length (stored in bytes 7-8 of the CDB for PERSISTENT RESERVE IN) or parameter list length (stored in bytes 5-8 of the CDB for PERSISTENT RESERVE OUT) is limited to 8 KiB.

For PERSISTENT RESERVE OUT, the parameter list is sent right after the CDB. The length of the parameter list is taken from the CDB itself.

The helper's reply has the following structure:

- 4 bytes for the SCSI status
- 4 bytes for the payload size (nonzero only for PERSISTENT RESERVE IN and only if the SCSI status is 0x00, i.e. GOOD)
- 96 bytes for the SCSI sense data
- if the size is nonzero, the payload follows

The sense data is always sent to keep the protocol simple, even though it is only valid if the SCSI status is CHECK CONDITION (0x02).

The payload size is always less than or equal to the allocation length specified in the CDB for the PERSISTENT RESERVE IN command.

If the protocol is violated, the helper closes the socket.

4.6 QEMU Guest Agent

4.6.1 Synopsis

qemu-ga [*OPTIONS*]

4.6.2 Description

The QEMU Guest Agent is a daemon intended to be run within virtual machines. It allows the hypervisor host to perform various operations in the guest, such as:

- get information from the guest
- set the guest's system time
- read/write a file
- sync and freeze the filesystems
- suspend the guest
- reconfigure guest local processors
- set user's password
- . . .

qemu-ga will read a system configuration file on startup (located at `/etc/qemu/qemu-ga.conf` by default), then parse remaining configuration options on the command line. For the same key, the last option wins, but the lists accumulate (see below for configuration file format).

4.6.3 Options

-m, --method=METHOD
Transport method: one of `unix-listen`, `virtio-serial`, or `isa-serial`, or `vsock-listen` (`virtio-serial` is the default).

-p, --path=PATH
Device/socket path (the default for virtio-serial is `/dev/virtio-ports/org.qemu.guest_agent.0`, the default for isa-serial is `/dev/ttyS0`). Socket addresses for vsock-listen are written as `<cid>:<port>`.

-l, --logfile=PATH
Set log file path (default is stderr).

-f, --pidfile=PATH
Specify pid file (default is `/var/run/qemu-ga.pid`).

-F, --fsfreeze-hook=PATH
Enable fsfreeze hook. Accepts an optional argument that specifies script to run on freeze/thaw. Script will be called with 'freeze'/'thaw' arguments accordingly (default is `/etc/qemu/fsfreeze-hook`). If using -F with an argument, do not follow -F with a space (for example: `-F/var/run/fsfreezehook.sh`).

-t, --statedir=PATH
Specify the directory to store state information (absolute paths only, default is `/var/run`).

-v, --verbose
Log extra debugging information.

-V, --version
Print version information and exit.

-d, --daemon
Daemonize after startup (detach from terminal).

-b, --blacklist=LIST
Comma-separated list of RPCs to disable (no spaces, `?` to list available RPCs).

-D, --dump-conf
Dump the configuration in a format compatible with `qemu-ga.conf` and exit.

-h, --help
Display this help and exit.

4.6.4 Files

The syntax of the `qemu-ga.conf` configuration file follows the Desktop Entry Specification, here is a quick summary: it consists of groups of key-value pairs, interspersed with comments.

```
# qemu-ga configuration sample
[general]
daemonize = 0
pidfile = /var/run/qemu-ga.pid
verbose = 0
method = virtio-serial
path = /dev/virtio-ports/org.qemu.guest_agent.0
statedir = /var/run
```

The list of keys follows the command line options:

Key	Key type
daemon	boolean
method	string
path	string
logfile	string
pidfile	string
fsfreeze-hook	string
statedir	string
verbose	boolean
blacklist	string list

4.6.5 See also

`qemu(1)`

4.7 Vhost-user Protocol

Table of Contents

4.7.1 Introduction

This protocol is aiming to complement the `ioctl` interface used to control the vhost implementation in the Linux kernel. It implements the control plane needed to establish virtqueue sharing with a user space process on the same host. It uses communication over a Unix domain socket to share file descriptors in the ancillary data of the message.

The protocol defines 2 sides of the communication, *master* and *slave*. *Master* is the application that shares its virtqueues, in our case QEMU. *Slave* is the consumer of the virtqueues.

In the current implementation QEMU is the *master*, and the *slave* is the external process consuming the virtio queues, for example a software Ethernet switch running in user space, such as Snabbswitch, or a block device backend processing read & write to a virtual disk. In order to facilitate interoperability between various backend implementations, it is recommended to follow the *Backend program conventions*.

Master and *slave* can be either a client (i.e. connecting) or server (listening) in the socket communication.

4.7.2 Message Specification

Note: All numbers are in the machine native byte order.

A vhost-user message consists of 3 header fields and a payload.

request	flags	size	payload

Header

request 32-bit type of the request

flags 32-bit bit field

- Lower 2 bits are the version (currently 0x01)
- Bit 2 is the reply flag - needs to be sent on each reply from the slave
- Bit 3 is the need_reply flag - see *REPLY_ACK* for details.

size 32-bit size of the payload

Payload

Depending on the request type, **payload** can be:

A single 64-bit integer

u64

u64 a 64-bit unsigned integer

A vring state description

index	num

index a 32-bit index

num a 32-bit number

A vring address description

index	flags	size	descriptor	used	available	log

index a 32-bit vring index

flags a 32-bit vring flags

descriptor a 64-bit ring address of the vring descriptor table

used a 64-bit ring address of the vring used ring

available a 64-bit ring address of the vring available ring

log a 64-bit guest address for logging

Note that a ring address is an IOVA if `VIRTIO_F_IOMMU_PLATFORM` has been negotiated. Otherwise it is a user address.

Memory regions description

num regions	padding	region0	…	region7

num regions a 32-bit number of regions

padding 32-bit

A region is:

guest address	size	user address	mmap offset

guest address a 64-bit guest address of the region

size a 64-bit size

user address a 64-bit user address

mmap offset 64-bit offset where region starts in the mapped memory

Log description

log size	log offset

log size size of area used for logging

log offset offset from start of supplied file descriptor where logging starts (i.e. where guest address 0 would be logged)

An IOTLB message

iova	size	user address	permissions flags	type

iova a 64-bit I/O virtual address programmed by the guest

size a 64-bit size

user address a 64-bit user address

permissions flags an 8-bit value: - 0: No access - 1: Read access - 2: Write access - 3: Read/Write access

type an 8-bit IOTLB message type: - 1: IOTLB miss - 2: IOTLB update - 3: IOTLB invalidate - 4: IOTLB access fail

Virtio device config space

offset	size	flags	payload

offset a 32-bit offset of virtio device's configuration space

size a 32-bit configuration space access size in bytes

flags a 32-bit value: - 0: Vhost master messages used for writeable fields - 1: Vhost master messages used for live migration

payload Size bytes array holding the contents of the virtio device's configuration space

Vring area description

u64	size	offset

u64 a 64-bit integer contains vring index and flags

size a 64-bit size of this area

offset a 64-bit offset of this area from the start of the supplied file descriptor

Inflight description

mmap size	mmap offset	num queues	queue size

mmap size a 64-bit size of area to track inflight I/O

mmap offset a 64-bit offset of this area from the start of the supplied file descriptor

num queues a 16-bit number of virtqueues

queue size a 16-bit size of virtqueues

C structure

In QEMU the vhost-user message is implemented with the following struct:

```
typedef struct VhostUserMsg {
    VhostUserRequest request;
    uint32_t flags;
    uint32_t size;
    union {
        uint64_t u64;
        struct vhost_vring_state state;
        struct vhost_vring_addr addr;
        VhostUserMemory memory;
        VhostUserLog log;
        struct vhost_iotlb_msg iotlb;
        VhostUserConfig config;
        VhostUserVringArea area;
        VhostUserInflight inflight;
    };
} QEMU_PACKED VhostUserMsg;
```

4.7.3 Communication

The protocol for vhost-user is based on the existing implementation of vhost for the Linux Kernel. Most messages that can be sent via the Unix domain socket implementing vhost-user have an equivalent ioctl to the kernel implementation.

The communication consists of *master* sending message requests and *slave* sending message replies. Most of the requests don't require replies. Here is a list of the ones that do:

- `VHOST_USER_GET_FEATURES`

- `VHOST_USER_GET_PROTOCOL_FEATURES`
- `VHOST_USER_GET_VRING_BASE`
- `VHOST_USER_SET_LOG_BASE` (if `VHOST_USER_PROTOCOL_F_LOG_SHMFD`)
- `VHOST_USER_GET_INFLIGHT_FD` (if `VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD`)

See also:

REPLY_ACK The section on `REPLY_ACK` protocol extension.

There are several messages that the master sends with file descriptors passed in the ancillary data:

- `VHOST_USER_SET_MEM_TABLE`
- `VHOST_USER_SET_LOG_BASE` (if `VHOST_USER_PROTOCOL_F_LOG_SHMFD`)
- `VHOST_USER_SET_LOG_FD`
- `VHOST_USER_SET_VRING_KICK`
- `VHOST_USER_SET_VRING_CALL`
- `VHOST_USER_SET_VRING_ERR`
- `VHOST_USER_SET_SLAVE_REQ_FD`
- `VHOST_USER_SET_INFLIGHT_FD` (if `VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD`)

If *master* is unable to send the full message or receives a wrong reply it will close the connection. An optional reconnection mechanism can be implemented.

If *slave* detects some error such as incompatible features, it may also close the connection. This should only happen in exceptional circumstances.

Any protocol extensions are gated by protocol feature bits, which allows full backwards compatibility on both master and slave. As older slaves don't support negotiating protocol features, a feature bit was dedicated for this purpose:

```
#define VHOST_USER_F_PROTOCOL_FEATURES 30
```

Starting and stopping rings

Client must only process each ring when it is started.

Client must only pass data between the ring and the backend, when the ring is enabled.

If ring is started but disabled, client must process the ring without talking to the backend.

For example, for a networking device, in the disabled state client must not supply any new RX packets, but must process and discard any TX packets.

If `VHOST_USER_F_PROTOCOL_FEATURES` has not been negotiated, the ring is initialized in an enabled state.

If `VHOST_USER_F_PROTOCOL_FEATURES` has been negotiated, the ring is initialized in a disabled state. Client must not pass data to/from the backend until ring is enabled by `VHOST_USER_SET_VRING_ENABLE` with parameter 1, or after it has been disabled by `VHOST_USER_SET_VRING_ENABLE` with parameter 0.

Each ring is initialized in a stopped state, client must not process it until ring is started, or after it has been stopped.

Client must start ring upon receiving a kick (that is, detecting that file descriptor is readable) on the descriptor specified by `VHOST_USER_SET_VRING_KICK` or receiving the in-band message `VHOST_USER_VRING_KICK` if negotiated, and stop ring upon receiving `VHOST_USER_GET_VRING_BASE`.

While processing the rings (whether they are enabled or not), client must support changing some configuration aspects on the fly.

Multiple queue support

Many devices have a fixed number of virtqueues. In this case the master already knows the number of available virtqueues without communicating with the slave.

Some devices do not have a fixed number of virtqueues. Instead the maximum number of virtqueues is chosen by the slave. The number can depend on host resource availability or slave implementation details. Such devices are called multiple queue devices.

Multiple queue support allows the slave to advertise the maximum number of queues. This is treated as a protocol extension, hence the slave has to implement protocol features first. The multiple queues feature is supported only when the protocol feature `VHOST_USER_PROTOCOL_F_MQ` (bit 0) is set.

The max number of queues the slave supports can be queried with message `VHOST_USER_GET_QUEUE_NUM`. Master should stop when the number of requested queues is bigger than that.

As all queues share one connection, the master uses a unique index for each queue in the sent message to identify a specified queue.

The master enables queues by sending message `VHOST_USER_SET_VRING_ENABLE`. vhost-user-net has historically automatically enabled the first queue pair.

Slaves should always implement the `VHOST_USER_PROTOCOL_F_MQ` protocol feature, even for devices with a fixed number of virtqueues, since it is simple to implement and offers a degree of introspection.

Masters must not rely on the `VHOST_USER_PROTOCOL_F_MQ` protocol feature for devices with a fixed number of virtqueues. Only true multiqueue devices require this protocol feature.

Migration

During live migration, the master may need to track the modifications the slave makes to the memory mapped regions. The client should mark the dirty pages in a log. Once it complies to this logging, it may declare the `VHOST_F_LOG_ALL` vhost feature.

To start/stop logging of data/used ring writes, server may send messages `VHOST_USER_SET_FEATURES` with `VHOST_F_LOG_ALL` and `VHOST_USER_SET_VRING_ADDR` with `VHOST_VRING_F_LOG` in ring's flags set to 1/0, respectively.

All the modifications to memory pointed by vring "descriptor" should be marked. Modifications to "used" vring should be marked if `VHOST_VRING_F_LOG` is part of ring's flags.

Dirty pages are of size:

```
#define VHOST_LOG_PAGE 0x1000
```

The log memory fd is provided in the ancillary data of `VHOST_USER_SET_LOG_BASE` message when the slave has `VHOST_USER_PROTOCOL_F_LOG_SHMFD` protocol feature.

The size of the log is supplied as part of `VhostUserMsg` which should be large enough to cover all known guest addresses. Log starts at the supplied offset in the supplied file descriptor. The log covers from address 0 to the maximum of guest regions. In pseudo-code, to mark page at `addr` as dirty:

```
page = addr / VHOST_LOG_PAGE
log[page / 8] |= 1 << page % 8
```

Where `addr` is the guest physical address.

Use atomic operations, as the log may be concurrently manipulated.

Note that when logging modifications to the used ring (when VHOST_VRING_F_LOG is set for this ring), log_guest_addr should be used to calculate the log offset: the write to first byte of the used ring is logged at this offset from log start. Also note that this value might be outside the legal guest physical address range (i.e. does not have to be covered by the VhostUserMemory table), but the bit offset of the last byte of the ring must fall within the size supplied by VhostUserLog.

VHOST_USER_SET_LOG_FD is an optional message with an eventfd in ancillary data, it may be used to inform the master that the log has been modified.

Once the source has finished migration, rings will be stopped by the source. No further update must be done before rings are restarted.

In postcopy migration the slave is started before all the memory has been received from the source host, and care must be taken to avoid accessing pages that have yet to be received. The slave opens a 'userfault'-fd and registers the memory with it; this fd is then passed back over to the master. The master services requests on the userfaultfd for pages that are accessed and when the page is available it performs WAKE ioctl's on the userfaultfd to wake the stalled slave. The client indicates support for this via the VHOST_USER_PROTOCOL_F_PAGEFAULT feature.

Memory access

The master sends a list of vhost memory regions to the slave using the VHOST_USER_SET_MEM_TABLE message. Each region has two base addresses: a guest address and a user address.

Messages contain guest addresses and/or user addresses to reference locations within the shared memory. The mapping of these addresses works as follows.

User addresses map to the vhost memory region containing that user address.

When the VIRTIO_F_IOMMU_PLATFORM feature has not been negotiated:

- Guest addresses map to the vhost memory region containing that guest address.

When the VIRTIO_F_IOMMU_PLATFORM feature has been negotiated:

- Guest addresses are also called I/O virtual addresses (IOVAs). They are translated to user addresses via the IOTLB.
- The vhost memory region guest address is not used.

IOMMU support

When the VIRTIO_F_IOMMU_PLATFORM feature has been negotiated, the master sends IOTLB entries update & invalidation by sending VHOST_USER_IOTLB_MSG requests to the slave with a struct vhost_iotlb_msg as payload. For update events, the iotlb payload has to be filled with the update message type (2), the I/O virtual address, the size, the user virtual address, and the permissions flags. Addresses and size must be within vhost memory regions set via the VHOST_USER_SET_MEM_TABLE request. For invalidation events, the iotlb payload has to be filled with the invalidation message type (3), the I/O virtual address and the size. On success, the slave is expected to reply with a zero payload, non-zero otherwise.

The slave relies on the slave communcation channel (see *Slave communication* section below) to send IOTLB miss and access failure events, by sending VHOST_USER_SLAVE_IOTLB_MSG requests to the master with a struct vhost_iotlb_msg as payload. For miss events, the iotlb payload has to be filled with the miss message type (1), the I/O virtual address and the permissions flags. For access failure event, the iotlb payload has to be filled with the access failure message type (4), the I/O virtual address and the permissions flags. For synchronization purpose, the slave may rely on the reply-ack feature, so the master may send a reply when operation is completed if the reply-ack feature is negotiated and slaves requests a reply. For miss events, completed operation means either master sent an update message containing the IOTLB entry containing requested address and permission, or master sent nothing if the IOTLB miss message is invalid (invalid IOVA or permission).

The master isn't expected to take the initiative to send IOTLB update messages, as the slave sends IOTLB miss messages for the guest virtual memory areas it needs to access.

Slave communication

An optional communication channel is provided if the slave declares `VHOST_USER_PROTOCOL_F_SLAVE_REQ` protocol feature, to allow the slave to make requests to the master.

The fd is provided via `VHOST_USER_SET_SLAVE_REQ_FD` ancillary data.

A slave may then send `VHOST_USER_SLAVE_*` messages to the master using this fd communication channel.

If `VHOST_USER_PROTOCOL_F_SLAVE_SEND_FD` protocol feature is negotiated, slave can send file descriptors (at most 8 descriptors in each message) to master via ancillary data using this fd communication channel.

Inflight I/O tracking

To support reconnecting after restart or crash, slave may need to resubmit inflight I/Os. If virtqueue is processed in order, we can easily achieve that by getting the inflight descriptors from descriptor table (split virtqueue) or descriptor ring (packed virtqueue). However, it can't work when we process descriptors out-of-order because some entries which store the information of inflight descriptors in available ring (split virtqueue) or descriptor ring (packed virtqueue) might be overrided by new entries. To solve this problem, slave need to allocate an extra buffer to store this information of inflight descriptors and share it with master for persistent. `VHOST_USER_GET_INFLIGHT_FD` and `VHOST_USER_SET_INFLIGHT_FD` are used to transfer this buffer between master and slave. And the format of this buffer is described below:

queue0 region	queue1 region	...	queueN region

N is the number of available virtqueues. Slave could get it from num queues field of `VhostUserInflight`.

For split virtqueue, queue region can be implemented as:

```
typedef struct DescStateSplit {
    /* Indicate whether this descriptor is inflight or not.
     * Only available for head-descriptor. */
    uint8_t inflight;

    /* Padding */
    uint8_t padding[5];

    /* Maintain a list for the last batch of used descriptors.
     * Only available when batching is used for submitting */
    uint16_t next;

    /* Used to preserve the order of fetching available descriptors.
     * Only available for head-descriptor. */
    uint64_t counter;
} DescStateSplit;

typedef struct QueueRegionSplit {
    /* The feature flags of this region. Now it's initialized to 0. */
    uint64_t features;

    /* The version of this region. It's 1 currently.
     * Zero value indicates an uninitialized buffer */
```

(continues on next page)

(continued from previous page)

```
    uint16_t version;

    /* The size of DescStateSplit array. It's equal to the virtqueue
     * size. Slave could get it from queue size field of VhostUserInflight. */
    uint16_t desc_num;

    /* The head of list that track the last batch of used descriptors. */
    uint16_t last_batch_head;

    /* Store the idx value of used ring */
    uint16_t used_idx;

    /* Used to track the state of each descriptor in descriptor table */
    DescStateSplit desc[];
} QueueRegionSplit;
```

To track inflight I/O, the queue region should be processed as follows:

When receiving available buffers from the driver:

1. Get the next available head-descriptor index from available ring, `i`
2. Set `desc[i].counter` to the value of global counter
3. Increase global counter by 1
4. Set `desc[i].inflight` to 1

When supplying used buffers to the driver:

1. Get corresponding used head-descriptor index, i
2. Set `desc[i].next` to `last_batch_head`
3. Set `last_batch_head` to `i`
4. Steps 1,2,3 may be performed repeatedly if batching is possible
5. Increase the `idx` value of used ring by the size of the batch
6. Set the `inflight` field of each `DescStateSplit` entry in the batch to 0
7. Set `used_idx` to the `idx` value of used ring

When reconnecting:

1. If the value of `used_idx` does not match the `idx` value of used ring (means the inflight field of `DescStateSplit` entries in last batch may be incorrect),
    a. Subtract the value of `used_idx` from the `idx` value of used ring to get last batch size of `DescStateSplit` entries
    b. Set the `inflight` field of each `DescStateSplit` entry to 0 in last batch list which starts from `last_batch_head`
    c. Set `used_idx` to the `idx` value of used ring
2. Resubmit inflight `DescStateSplit` entries in order of their counter value

For packed virtqueue, queue region can be implemented as:

```
typedef struct DescStatePacked {
    /* Indicate whether this descriptor is inflight or not.
     * Only available for head-descriptor. */
```

(continues on next page)

(continued from previous page)

```
    uint8_t inflight;

    /* Padding */
    uint8_t padding;

    /* Link to the next free entry */
    uint16_t next;

    /* Link to the last entry of descriptor list.
     * Only available for head-descriptor. */
    uint16_t last;

    /* The length of descriptor list.
     * Only available for head-descriptor. */
    uint16_t num;

    /* Used to preserve the order of fetching available descriptors.
     * Only available for head-descriptor. */
    uint64_t counter;

    /* The buffer id */
    uint16_t id;

    /* The descriptor flags */
    uint16_t flags;

    /* The buffer length */
    uint32_t len;

    /* The buffer address */
    uint64_t addr;
} DescStatePacked;

typedef struct QueueRegionPacked {
    /* The feature flags of this region. Now it's initialized to 0. */
    uint64_t features;

    /* The version of this region. It's 1 currently.
     * Zero value indicates an uninitialized buffer */
    uint16_t version;

    /* The size of DescStatePacked array. It's equal to the virtqueue
     * size. Slave could get it from queue size field of VhostUserInflight. */
    uint16_t desc_num;

    /* The head of free DescStatePacked entry list */
    uint16_t free_head;

    /* The old head of free DescStatePacked entry list */
    uint16_t old_free_head;

    /* The used index of descriptor ring */
    uint16_t used_idx;

    /* The old used index of descriptor ring */
    uint16_t old_used_idx;
```

(continues on next page)

(continued from previous page)

```
    /* Device ring wrap counter */
    uint8_t used_wrap_counter;

    /* The old device ring wrap counter */
    uint8_t old_used_wrap_counter;

    /* Padding */
    uint8_t padding[7];

    /* Used to track the state of each descriptor fetched from descriptor ring */
    DescStatePacked desc[];
} QueueRegionPacked;
```

To track inflight I/O, the queue region should be processed as follows:

When receiving available buffers from the driver:

1. Get the next available descriptor entry from descriptor ring, `d`
2. If `d` is head descriptor,
   a. Set `desc[old_free_head].num` to 0
   b. Set `desc[old_free_head].counter` to the value of global counter
   c. Increase global counter by 1
   d. Set `desc[old_free_head].inflight` to 1
3. If `d` is last descriptor, set `desc[old_free_head].last` to `free_head`
4. Increase `desc[old_free_head].num` by 1
5. Set `desc[free_head].addr`, `desc[free_head].len`, `desc[free_head].flags`, `desc[free_head].id` to `d.addr`, `d.len`, `d.flags`, `d.id`
6. Set `free_head` to `desc[free_head].next`
7. If `d` is last descriptor, set `old_free_head` to `free_head`

When supplying used buffers to the driver:

1. Get corresponding used head-descriptor entry from descriptor ring, `d`
2. Get corresponding `DescStatePacked` entry, `e`
3. Set `desc[e.last].next` to `free_head`
4. Set `free_head` to the index of `e`
5. Steps 1,2,3,4 may be performed repeatedly if batching is possible
6. Increase `used_idx` by the size of the batch and update `used_wrap_counter` if needed
7. Update `d.flags`
8. Set the `inflight` field of each head `DescStatePacked` entry in the batch to 0
9. Set `old_free_head`, `old_used_idx`, `old_used_wrap_counter` to `free_head`, `used_idx`, `used_wrap_counter`

When reconnecting:

1. If `used_idx` does not match `old_used_idx` (means the `inflight` field of `DescStatePacked` entries in last batch may be incorrect),

   a. Get the next descriptor ring entry through `old_used_idx`, `d`
   b. Use `old_used_wrap_counter` to calculate the available flags
   c. If `d.flags` is not equal to the calculated flags value (means slave has submitted the buffer to guest driver before crash, so it has to commit the in-progres update), set `old_free_head`, `old_used_idx`, `old_used_wrap_counter` to `free_head`, `used_idx`, `used_wrap_counter`

2. Set `free_head`, `used_idx`, `used_wrap_counter` to `old_free_head`, `old_used_idx`, `old_used_wrap_counter` (roll back any in-progress update)
3. Set the `inflight` field of each `DescStatePacked` entry in free list to 0
4. Resubmit inflight `DescStatePacked` entries in order of their counter value

In-band notifications

In some limited situations (e.g. for simulation) it is desirable to have the kick, call and error (if used) signals done via in-band messages instead of asynchronous eventfd notifications. This can be done by negotiating the `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` protocol feature.

Note that due to the fact that too many messages on the sockets can cause the sending application(s) to block, it is not advised to use this feature unless absolutely necessary. It is also considered an error to negotiate this feature without also negotiating `VHOST_USER_PROTOCOL_F_SLAVE_REQ` and `VHOST_USER_PROTOCOL_F_REPLY_ACK`, the former is necessary for getting a message channel from the slave to the master, while the latter needs to be used with the in-band notification messages to block until they are processed, both to avoid blocking later and for proper processing (at least in the simulation use case.) As it has no other way of signalling this error, the slave should close the connection as a response to a `VHOST_USER_SET_PROTOCOL_FEATURES` message that sets the in-band notifications feature flag without the other two.

Protocol features

```
#define VHOST_USER_PROTOCOL_F_MQ                    0
#define VHOST_USER_PROTOCOL_F_LOG_SHMFD             1
#define VHOST_USER_PROTOCOL_F_RARP                  2
#define VHOST_USER_PROTOCOL_F_REPLY_ACK             3
#define VHOST_USER_PROTOCOL_F_MTU                   4
#define VHOST_USER_PROTOCOL_F_SLAVE_REQ             5
#define VHOST_USER_PROTOCOL_F_CROSS_ENDIAN          6
#define VHOST_USER_PROTOCOL_F_CRYPTO_SESSION        7
#define VHOST_USER_PROTOCOL_F_PAGEFAULT             8
#define VHOST_USER_PROTOCOL_F_CONFIG                9
#define VHOST_USER_PROTOCOL_F_SLAVE_SEND_FD        10
#define VHOST_USER_PROTOCOL_F_HOST_NOTIFIER        11
#define VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD       12
#define VHOST_USER_PROTOCOL_F_RESET_DEVICE         13
#define VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS 14
#define VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS  15
#define VHOST_USER_PROTOCOL_F_STATUS               16
```

Master message types

`VHOST_USER_GET_FEATURES`

id 1

equivalent ioctl `VHOST_GET_FEATURES`

master payload N/A

slave payload `u64`

Get from the underlying vhost implementation the features bitmask. Feature bit `VHOST_USER_F_PROTOCOL_FEATURES` signals slave support for `VHOST_USER_GET_PROTOCOL_FEATURES` and `VHOST_USER_SET_PROTOCOL_FEATURES`.

`VHOST_USER_SET_FEATURES`

id 2

equivalent ioctl `VHOST_SET_FEATURES`

master payload `u64`

Enable features in the underlying vhost implementation using a bitmask. Feature bit `VHOST_USER_F_PROTOCOL_FEATURES` signals slave support for `VHOST_USER_GET_PROTOCOL_FEATURES` and `VHOST_USER_SET_PROTOCOL_FEATURES`.

`VHOST_USER_GET_PROTOCOL_FEATURES`

id 15

equivalent ioctl `VHOST_GET_FEATURES`

master payload N/A

slave payload `u64`

Get the protocol feature bitmask from the underlying vhost implementation. Only legal if feature bit `VHOST_USER_F_PROTOCOL_FEATURES` is present in `VHOST_USER_GET_FEATURES`.

Note: Slave that reported `VHOST_USER_F_PROTOCOL_FEATURES` must support this message even before `VHOST_USER_SET_FEATURES` was called.

`VHOST_USER_SET_PROTOCOL_FEATURES`

id 16

equivalent ioctl `VHOST_SET_FEATURES`

master payload `u64`

Enable protocol features in the underlying vhost implementation.

Only legal if feature bit `VHOST_USER_F_PROTOCOL_FEATURES` is present in `VHOST_USER_GET_FEATURES`.

Note: Slave that reported `VHOST_USER_F_PROTOCOL_FEATURES` must support this message even before `VHOST_USER_SET_FEATURES` was called.

`VHOST_USER_SET_OWNER`

id 3

equivalent ioctl `VHOST_SET_OWNER`

master payload N/A

Issued when a new connection is established. It sets the current *master* as an owner of the session. This can be used on the *slave* as a "session start" flag.

VHOST_USER_RESET_OWNER

id 4

master payload N/A

Deprecated

This is no longer used. Used to be sent to request disabling all rings, but some clients interpreted it to also discard connection state (this interpretation would lead to bugs). It is recommended that clients either ignore this message, or use it to disable all rings.

VHOST_USER_SET_MEM_TABLE

id 5

equivalent ioctl `VHOST_SET_MEM_TABLE`

master payload memory regions description

slave payload (postcopy only) memory regions description

Sets the memory map regions on the slave so it can translate the vring addresses. In the ancillary data there is an array of file descriptors for each memory mapped region. The size and ordering of the fds matches the number and ordering of memory regions.

When `VHOST_USER_POSTCOPY_LISTEN` has been received, `SET_MEM_TABLE` replies with the bases of the memory mapped regions to the master. The slave must have mmap'd the regions but not yet accessed them and should not yet generate a userfault event.

Note: `NEED_REPLY_MASK` is not set in this case. QEMU will then reply back to the list of mappings with an empty `VHOST_USER_SET_MEM_TABLE` as an acknowledgement; only upon reception of this message may the guest start accessing the memory and generating faults.

VHOST_USER_SET_LOG_BASE

id 6

equivalent ioctl `VHOST_SET_LOG_BASE`

master payload u64

slave payload N/A

Sets logging shared memory space.

When slave has `VHOST_USER_PROTOCOL_F_LOG_SHMFD` protocol feature, the log memory fd is provided in the ancillary data of `VHOST_USER_SET_LOG_BASE` message, the size and offset of shared memory area provided in the message.

VHOST_USER_SET_LOG_FD

id 7

equivalent ioctl `VHOST_SET_LOG_FD`

master payload N/A

Sets the logging file descriptor, which is passed as ancillary data.

VHOST_USER_SET_VRING_NUM

id 8

equivalent ioctl `VHOST_SET_VRING_NUM`

master payload vring state description

Set the size of the queue.

VHOST_USER_SET_VRING_ADDR

id 9

equivalent ioctl `VHOST_SET_VRING_ADDR`

master payload vring address description

slave payload N/A

Sets the addresses of the different aspects of the vring.

VHOST_USER_SET_VRING_BASE

id 10

equivalent ioctl `VHOST_SET_VRING_BASE`

master payload vring state description

Sets the base offset in the available vring.

VHOST_USER_GET_VRING_BASE

id 11

equivalent ioctl `VHOST_USER_GET_VRING_BASE`

master payload vring state description

slave payload vring state description

Get the available vring base offset.

VHOST_USER_SET_VRING_KICK

id 12

equivalent ioctl `VHOST_SET_VRING_KICK`

master payload `u64`

Set the event file descriptor for adding buffers to the vring. It is passed in the ancillary data.

Bits (0-7) of the payload contain the vring index. Bit 8 is the invalid FD flag. This flag is set when there is no file descriptor in the ancillary data. This signals that polling should be used instead of waiting for the kick. Note that if the protocol feature `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` has been negotiated this message isn't necessary as the ring is also started on the `VHOST_USER_VRING_KICK` message, it may however still be used to set an event file descriptor (which will be preferred over the message) or to enable polling.

VHOST_USER_SET_VRING_CALL

id 13

equivalent ioctl `VHOST_SET_VRING_CALL`

master payload `u64`

Set the event file descriptor to signal when buffers are used. It is passed in the ancillary data.

Bits (0-7) of the payload contain the vring index. Bit 8 is the invalid FD flag. This flag is set when there is no file descriptor in the ancillary data. This signals that polling will be used instead of waiting for the call. Note that if the protocol features `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` and `VHOST_USER_PROTOCOL_F_SLAVE_REQ` have been negotiated this message isn't necessary as the `VHOST_USER_SLAVE_VRING_CALL` message can be used, it may however still be used to set an event file descriptor or to enable polling.

VHOST_USER_SET_VRING_ERR

id 14

equivalent ioctl `VHOST_SET_VRING_ERR`

master payload `u64`

Set the event file descriptor to signal when error occurs. It is passed in the ancillary data.

Bits (0-7) of the payload contain the vring index. Bit 8 is the invalid FD flag. This flag is set when there is no file descriptor in the ancillary data. Note that if the protocol features `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` and `VHOST_USER_PROTOCOL_F_SLAVE_REQ` have been negotiated this message isn't necessary as the `VHOST_USER_SLAVE_VRING_ERR` message can be used, it may however still be used to set an event file descriptor (which will be preferred over the message).

VHOST_USER_GET_QUEUE_NUM

id 17

equivalent ioctl N/A

master payload N/A

slave payload u64

Query how many queues the backend supports.

This request should be sent only when `VHOST_USER_PROTOCOL_F_MQ` is set in queried protocol features by `VHOST_USER_GET_PROTOCOL_FEATURES`.

VHOST_USER_SET_VRING_ENABLE

id 18

equivalent ioctl N/A

master payload vring state description

Signal slave to enable or disable corresponding vring.

This request should be sent only when `VHOST_USER_F_PROTOCOL_FEATURES` has been negotiated.

VHOST_USER_SEND_RARP

id 19

equivalent ioctl N/A

master payload `u64`

Ask vhost user backend to broadcast a fake RARP to notify the migration is terminated for guest that does not support GUEST_ANNOUNCE.

Only legal if feature bit `VHOST_USER_F_PROTOCOL_FEATURES` is present in `VHOST_USER_GET_FEATURES` and protocol feature bit `VHOST_USER_PROTOCOL_F_RARP` is present in

`VHOST_USER_GET_PROTOCOL_FEATURES`. The first 6 bytes of the payload contain the mac address of the guest to allow the vhost user backend to construct and broadcast the fake RARP.

`VHOST_USER_NET_SET_MTU`

id 20

equivalent ioctl N/A

master payload `u64`

Set host MTU value exposed to the guest.

This request should be sent only when `VIRTIO_NET_F_MTU` feature has been successfully negotiated, `VHOST_USER_F_PROTOCOL_FEATURES` is present in `VHOST_USER_GET_FEATURES` and protocol feature bit `VHOST_USER_PROTOCOL_F_NET_MTU` is present in `VHOST_USER_GET_PROTOCOL_FEATURES`.

If `VHOST_USER_PROTOCOL_F_REPLY_ACK` is negotiated, slave must respond with zero in case the specified MTU is valid, or non-zero otherwise.

`VHOST_USER_SET_SLAVE_REQ_FD`

id 21

equivalent ioctl N/A

master payload N/A

Set the socket file descriptor for slave initiated requests. It is passed in the ancillary data.

This request should be sent only when `VHOST_USER_F_PROTOCOL_FEATURES` has been negotiated, and protocol feature bit `VHOST_USER_PROTOCOL_F_SLAVE_REQ` bit is present in `VHOST_USER_GET_PROTOCOL_FEATURES`. If `VHOST_USER_PROTOCOL_F_REPLY_ACK` is negotiated, slave must respond with zero for success, non-zero otherwise.

`VHOST_USER_IOTLB_MSG`

id 22

equivalent ioctl N/A (equivalent to `VHOST_IOTLB_MSG` message type)

master payload `struct vhost_iotlb_msg`

slave payload `u64`

Send IOTLB messages with `struct vhost_iotlb_msg` as payload.

Master sends such requests to update and invalidate entries in the device IOTLB. The slave has to acknowledge the request with sending zero as `u64` payload for success, non-zero otherwise.

This request should be send only when `VIRTIO_F_IOMMU_PLATFORM` feature has been successfully negotiated.

`VHOST_USER_SET_VRING_ENDIAN`

id 23

equivalent ioctl `VHOST_SET_VRING_ENDIAN`

master payload vring state description

Set the endianness of a VQ for legacy devices. Little-endian is indicated with state.num set to 0 and big-endian is indicated with state.num set to 1. Other values are invalid.

This request should be sent only when `VHOST_USER_PROTOCOL_F_CROSS_ENDIAN` has been negotiated. Backends that negotiated this feature should handle both endiannesses and expect this message once (per VQ) during device configuration (ie. before the master starts the VQ).

`VHOST_USER_GET_CONFIG`

id 24

equivalent ioctl N/A

master payload virtio device config space

slave payload virtio device config space

When `VHOST_USER_PROTOCOL_F_CONFIG` is negotiated, this message is submitted by the vhost-user master to fetch the contents of the virtio device configuration space, vhost-user slave's payload size MUST match master's request, vhost-user slave uses zero length of payload to indicate an error to vhost-user master. The vhost-user master may cache the contents to avoid repeated `VHOST_USER_GET_CONFIG` calls.

`VHOST_USER_SET_CONFIG`

id 25

equivalent ioctl N/A

master payload virtio device config space

slave payload N/A

When `VHOST_USER_PROTOCOL_F_CONFIG` is negotiated, this message is submitted by the vhost-user master when the Guest changes the virtio device configuration space and also can be used for live migration on the destination host. The vhost-user slave must check the flags field, and slaves MUST NOT accept SET_CONFIG for read-only configuration space fields unless the live migration bit is set.

`VHOST_USER_CREATE_CRYPTO_SESSION`

id 26

equivalent ioctl N/A

master payload crypto session description

slave payload crypto session description

Create a session for crypto operation. The server side must return the session id, 0 or positive for success, negative for failure. This request should be sent only when `VHOST_USER_PROTOCOL_F_CRYPTO_SESSION` feature has been successfully negotiated. It's a required feature for crypto devices.

`VHOST_USER_CLOSE_CRYPTO_SESSION`

id 27

equivalent ioctl N/A

master payload `u64`

Close a session for crypto operation which was previously created by `VHOST_USER_CREATE_CRYPTO_SESSION`.

This request should be sent only when `VHOST_USER_PROTOCOL_F_CRYPTO_SESSION` feature has been successfully negotiated. It's a required feature for crypto devices.

`VHOST_USER_POSTCOPY_ADVISE`

id 28

master payload N/A

slave payload userfault fd

When `VHOST_USER_PROTOCOL_F_PAGEFAULT` is supported, the master advises slave that a migration with postcopy enabled is underway, the slave must open a userfaultfd for later use. Note that at this stage the migration is still in precopy mode.

VHOST_USER_POSTCOPY_LISTEN

id 29

master payload N/A

Master advises slave that a transition to postcopy mode has happened. The slave must ensure that shared memory is registered with userfaultfd to cause faulting of non-present pages.

This is always sent sometime after a `VHOST_USER_POSTCOPY_ADVISE`, and thus only when `VHOST_USER_PROTOCOL_F_PAGEFAULT` is supported.

VHOST_USER_POSTCOPY_END

id 30

slave payload `u64`

Master advises that postcopy migration has now completed. The slave must disable the userfaultfd. The response is an acknowledgement only.

When `VHOST_USER_PROTOCOL_F_PAGEFAULT` is supported, this message is sent at the end of the migration, after `VHOST_USER_POSTCOPY_LISTEN` was previously sent.

The value returned is an error indication; 0 is success.

VHOST_USER_GET_INFLIGHT_FD

id 31

equivalent ioctl N/A

master payload inflight description

When `VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD` protocol feature has been successfully negotiated, this message is submitted by master to get a shared buffer from slave. The shared buffer will be used to track inflight I/O by slave. QEMU should retrieve a new one when vm reset.

VHOST_USER_SET_INFLIGHT_FD

id 32

equivalent ioctl N/A

master payload inflight description

When `VHOST_USER_PROTOCOL_F_INFLIGHT_SHMFD` protocol feature has been successfully negotiated, this message is submitted by master to send the shared inflight buffer back to slave so that slave could get inflight I/O after a crash or restart.

VHOST_USER_GPU_SET_SOCKET

id 33

equivalent ioctl N/A

master payload N/A

Sets the GPU protocol socket file descriptor, which is passed as ancillary data. The GPU protocol is used to inform the master of rendering state and updates. See vhost-user-gpu.rst for details.

VHOST_USER_RESET_DEVICE

id 34

equivalent ioctl N/A

master payload N/A

slave payload N/A

Ask the vhost user backend to disable all rings and reset all internal device state to the initial state, ready to be reinitialized. The backend retains ownership of the device throughout the reset operation.

Only valid if the `VHOST_USER_PROTOCOL_F_RESET_DEVICE` protocol feature is set by the backend.

`VHOST_USER_VRING_KICK`

id 35

equivalent ioctl N/A

slave payload vring state description

master payload N/A

When the `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` protocol feature has been successfully negotiated, this message may be submitted by the master to indicate that a buffer was added to the vring instead of signalling it using the vring's kick file descriptor or having the slave rely on polling.

The state.num field is currently reserved and must be set to 0.

`VHOST_USER_GET_MAX_MEM_SLOTS`

id 36

equivalent ioctl N/A

slave payload u64

When the `VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS` protocol feature has been successfully negotiated, this message is submitted by master to the slave. The slave should return the message with a u64 payload containing the maximum number of memory slots for QEMU to expose to the guest. The value returned by the backend will be capped at the maximum number of ram slots which can be supported by the target platform.

`VHOST_USER_ADD_MEM_REG`

id 37

equivalent ioctl N/A

slave payload memory region

When the `VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS` protocol feature has been successfully negotiated, this message is submitted by the master to the slave. The message payload contains a memory region descriptor struct, describing a region of guest memory which the slave device must map in. When the `VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS` protocol feature has been successfully negotiated, along with the `VHOST_USER_REM_MEM_REG` message, this message is used to set and update the memory tables of the slave device.

`VHOST_USER_REM_MEM_REG`

id 38

equivalent ioctl N/A

slave payload memory region

When the `VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS` protocol feature has been successfully negotiated, this message is submitted by the master to the slave. The message payload contains a memory region descriptor struct, describing a region of guest memory which the slave device must unmap. When the `VHOST_USER_PROTOCOL_F_CONFIGURE_MEM_SLOTS` protocol feature has been successfully negotiated, along with the `VHOST_USER_ADD_MEM_REG` message, this message is used to set and update the memory tables of the slave device.

VHOST_USER_SET_STATUS

id 39

equivalent ioctl VHOST_VDPA_SET_STATUS

slave payload N/A

master payload `u64`

When the `VHOST_USER_PROTOCOL_F_STATUS` protocol feature has been successfully negotiated, this message is submitted by the master to notify the backend with updated device status as defined in the Virtio specification.

VHOST_USER_GET_STATUS

id 40

equivalent ioctl VHOST_VDPA_GET_STATUS

slave payload `u64`

master payload N/A

When the `VHOST_USER_PROTOCOL_F_STATUS` protocol feature has been successfully negotiated, this message is submitted by the master to query the backend for its device status as defined in the Virtio specification.

Slave message types

VHOST_USER_SLAVE_IOTLB_MSG

id 1

equivalent ioctl N/A (equivalent to `VHOST_IOTLB_MSG` message type)

slave payload `struct vhost_iotlb_msg`

master payload N/A

Send IOTLB messages with `struct vhost_iotlb_msg` as payload. Slave sends such requests to notify of an IOTLB miss, or an IOTLB access failure. If `VHOST_USER_PROTOCOL_F_REPLY_ACK` is negotiated, and slave set the `VHOST_USER_NEED_REPLY` flag, master must respond with zero when operation is successfully completed, or non-zero otherwise. This request should be send only when `VIRTIO_F_IOMMU_PLATFORM` feature has been successfully negotiated.

VHOST_USER_SLAVE_CONFIG_CHANGE_MSG

id 2

equivalent ioctl N/A

slave payload N/A

master payload N/A

When `VHOST_USER_PROTOCOL_F_CONFIG` is negotiated, vhost-user slave sends such messages to notify that the virtio device's configuration space has changed, for those host devices which can support such feature, host driver can send `VHOST_USER_GET_CONFIG` message to slave to get the latest content. If

`VHOST_USER_PROTOCOL_F_REPLY_ACK` is negotiated, and slave set the `VHOST_USER_NEED_REPLY` flag, master must respond with zero when operation is successfully completed, or non-zero otherwise.

VHOST_USER_SLAVE_VRING_HOST_NOTIFIER_MSG

id 3

equivalent ioctl N/A

slave payload vring area description

master payload N/A

Sets host notifier for a specified queue. The queue index is contained in the `u64` field of the vring area description. The host notifier is described by the file descriptor (typically it's a VFIO device fd) which is passed as ancillary data and the size (which is mmap size and should be the same as host page size) and offset (which is mmap offset) carried in the vring area description. QEMU can mmap the file descriptor based on the size and offset to get a memory range. Registering a host notifier means mapping this memory range to the VM as the specified queue's notify MMIO region. Slave sends this request to tell QEMU to de-register the existing notifier if any and register the new notifier if the request is sent with a file descriptor.

This request should be sent only when `VHOST_USER_PROTOCOL_F_HOST_NOTIFIER` protocol feature has been successfully negotiated.

VHOST_USER_SLAVE_VRING_CALL

id 4

equivalent ioctl N/A

slave payload vring state description

master payload N/A

When the `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` protocol feature has been successfully negotiated, this message may be submitted by the slave to indicate that a buffer was used from the vring instead of signalling this using the vring's call file descriptor or having the master relying on polling.

The state.num field is currently reserved and must be set to 0.

VHOST_USER_SLAVE_VRING_ERR

id 5

equivalent ioctl N/A

slave payload vring state description

master payload N/A

When the `VHOST_USER_PROTOCOL_F_INBAND_NOTIFICATIONS` protocol feature has been successfully negotiated, this message may be submitted by the slave to indicate that an error occurred on the specific vring, instead of signalling the error file descriptor set by the master via `VHOST_USER_SET_VRING_ERR`.

The state.num field is currently reserved and must be set to 0.

VHOST_USER_PROTOCOL_F_REPLY_ACK

The original vhost-user specification only demands replies for certain commands. This differs from the vhost protocol implementation where commands are sent over an `ioctl()` call and block until the client has completed.

With this protocol extension negotiated, the sender (QEMU) can set the `need_reply` [Bit 3] flag to any command. This indicates that the client MUST respond with a Payload `VhostUserMsg` indicating success or failure. The payload should be set to zero on success or non-zero on failure, unless the message already has an explicit reply body.

The response payload gives QEMU a deterministic indication of the result of the command. Today, QEMU is expected to terminate the main vhost-user loop upon receiving such errors. In future, qemu could be taught to be more resilient for selective requests.

For the message types that already solicit a reply from the client, the presence of `VHOST_USER_PROTOCOL_F_REPLY_ACK` or need_reply bit being set brings no behavioural change. (See the *Communication* section for details.)

4.7.4 Backend program conventions

vhost-user backends can provide various devices & services and may need to be configured manually depending on the use case. However, it is a good idea to follow the conventions listed here when possible. Users, QEMU or libvirt, can then rely on some common behaviour to avoid heterogenous configuration and management of the backend programs and facilitate interoperability.

Each backend installed on a host system should come with at least one JSON file that conforms to the vhost-user.json schema. Each file informs the management applications about the backend type, and binary location. In addition, it defines rules for management apps for picking the highest priority backend when multiple match the search criteria (see `@VhostUserBackend` documentation in the schema file).

If the backend is not capable of enabling a requested feature on the host (such as 3D acceleration with virgl), or the initialization failed, the backend should fail to start early and exit with a status != 0. It may also print a message to stderr for further details.

The backend program must not daemonize itself, but it may be daemonized by the management layer. It may also have a restricted access to the system.

File descriptors 0, 1 and 2 will exist, and have regular stdin/stdout/stderr usage (they may have been redirected to /dev/null by the management layer, or to a log handler).

The backend program must end (as quickly and cleanly as possible) when the SIGTERM signal is received. Eventually, it may receive SIGKILL by the management layer after a few seconds.

The following command line options have an expected behaviour. They are mandatory, unless explicitly said differently:

--socket-path=PATH	This option specify the location of the vhost-user Unix domain socket. It is incompatible with –fd.
--fd=FDNUM	When this argument is given, the backend program is started with the vhost-user socket as file descriptor FDNUM. It is incompatible with –socket-path.
--print-capabilities	Output to stdout the backend capabilities in JSON format, and then exit successfully. Other options and arguments should be ignored, and the backend program should not perform its normal function. The capabilities can be reported dynamically depending on the host capabilities.

The JSON output is described in the `vhost-user.json` schema, by `` ` ``@VHostUserBackendCapabilities``. Example:

```
{
  "type": "foo",
  "features": [
    "feature-a",
    "feature-b"
  ]
}
```

vhost-user-input

Command line options:

--evdev-path=PATH	Specify the linux input device. (optional)
--no-grab	Do no request exclusive access to the input device. (optional)

vhost-user-gpu

Command line options:

--render-node=PATH	Specify the GPU DRM render node. (optional)
--virgl	Enable virgl rendering support. (optional)

vhost-user-blk

Command line options:

--blk-file=PATH	Specify block device or file path. (optional)
--read-only	Enable read-only. (optional)

4.8 Vhost-user-gpu Protocol

Table of Contents

4.8.1 Introduction

The vhost-user-gpu protocol is aiming at sharing the rendering result of a virtio-gpu, done from a vhost-user slave process to a vhost-user master process (such as QEMU). It bears a resemblance to a display server protocol, if you consider QEMU as the display server and the slave as the client, but in a very limited way. Typically, it will work by setting a scanout/display configuration, before sending flush events for the display updates. It will also update the cursor shape and position.

The protocol is sent over a UNIX domain stream socket, since it uses socket ancillary data to share opened file descriptors (DMABUF fds or shared memory). The socket is usually obtained via `VHOST_USER_GPU_SET_SOCKET`.

Requests are sent by the *slave*, and the optional replies by the *master*.

4.8.2 Wire format

Unless specified differently, numbers are in the machine native byte order.

A vhost-user-gpu message (request and reply) consists of 3 header fields and a payload.

request	flags	size	payload

Header

request `u32`, type of the request

flags `u32`, 32-bit bit field:

- Bit 2 is the reply flag - needs to be set on each reply

size `u32`, size of the payload

Payload types

Depending on the request type, **payload** can be:

VhostUserGpuCursorPos

scanout-id	x	y

scanout-id `u32`, the scanout where the cursor is located

x/y `u32`, the cursor postion

VhostUserGpuCursorUpdate

pos	hot_x	hot_y	cursor

pos a `VhostUserGpuCursorPos`, the cursor location

hot_x/hot_y `u32`, the cursor hot location

cursor `[u32; 64 * 64]`, 64x64 RGBA cursor data (PIXMAN_a8r8g8b8 format)

VhostUserGpuScanout

scanout-id	w	h

scanout-id `u32`, the scanout configuration to set

w/h `u32`, the scanout width/height size

VhostUserGpuUpdate

scanout-id	x	y	w	h	data

scanout-id `u32`, the scanout content to update

x/y/w/h `u32`, region of the update

data RGB data (PIXMAN_x8r8g8b8 format)

VhostUserGpuDMABUFScanout

scanout-id	x	y	w	h	fdw	fwh	stride	flags	fourcc

scanout-id `u32`, the scanout configuration to set

x/y `u32`, the location of the scanout within the DMABUF

w/h `u32`, the scanout width/height size

fdw/fdh/stride/flags `u32`, the DMABUF width/height/stride/flags

fourcc `i32`, the DMABUF fourcc

C structure

In QEMU the vhost-user-gpu message is implemented with the following struct:

```
typedef struct VhostUserGpuMsg {
    uint32_t request; /* VhostUserGpuRequest */
    uint32_t flags;
    uint32_t size; /* the following payload size */
    union {
        VhostUserGpuCursorPos cursor_pos;
```

(continues on next page)

(continued from previous page)

```
        VhostUserGpuCursorUpdate cursor_update;
        VhostUserGpuScanout scanout;
        VhostUserGpuUpdate update;
        VhostUserGpuDMABUFScanout dmabuf_scanout;
        struct virtio_gpu_resp_display_info display_info;
        uint64_t u64;
    } payload;
} QEMU_PACKED VhostUserGpuMsg;
```

Protocol features

None yet.

As the protocol may need to evolve, new messages and communication changes are negotiated thanks to preliminary `VHOST_USER_GPU_GET_PROTOCOL_FEATURES` and `VHOST_USER_GPU_SET_PROTOCOL_FEATURES` requests.

4.8.3 Communication

Message types

`VHOST_USER_GPU_GET_PROTOCOL_FEATURES`

id 1

request payload N/A

reply payload `u64`

Get the supported protocol features bitmask.

`VHOST_USER_GPU_SET_PROTOCOL_FEATURES`

id 2

request payload `u64`

reply payload N/A

Enable protocol features using a bitmask.

`VHOST_USER_GPU_GET_DISPLAY_INFO`

id 3

request payload N/A

reply payload `struct virtio_gpu_resp_display_info` (from virtio specification)

Get the preferred display configuration.

`VHOST_USER_GPU_CURSOR_POS`

id 4

request payload `VhostUserGpuCursorPos`

reply payload N/A

Set/show the cursor position.

`VHOST_USER_GPU_CURSOR_POS_HIDE`

id 5

request payload `VhostUserGpuCursorPos`

reply payload N/A

Set/hide the cursor.

`VHOST_USER_GPU_CURSOR_UPDATE`

id 6

request payload `VhostUserGpuCursorUpdate`

reply payload N/A

Update the cursor shape and location.

`VHOST_USER_GPU_SCANOUT`

id 7

request payload `VhostUserGpuScanout`

reply payload N/A

Set the scanout resolution. To disable a scanout, the dimensions width/height are set to 0.

`VHOST_USER_GPU_UPDATE`

id 8

request payload `VhostUserGpuUpdate`

reply payload N/A

Update the scanout content. The data payload contains the graphical bits. The display should be flushed and presented.

`VHOST_USER_GPU_DMABUF_SCANOUT`

id 9

request payload `VhostUserGpuDMABUFScanout`

reply payload N/A

Set the scanout resolution/configuration, and share a DMABUF file descriptor for the scanout content, which is passed as ancillary data. To disable a scanout, the dimensions width/height are set to 0, there is no file descriptor passed.

`VHOST_USER_GPU_DMABUF_UPDATE`

id 10

request payload `VhostUserGpuUpdate`

reply payload empty payload

The display should be flushed and presented according to updated region from `VhostUserGpuUpdate`.

Note: there is no data payload, since the scanout is shared thanks to DMABUF, that must have been set previously with `VHOST_USER_GPU_DMABUF_SCANOUT`.

4.9 Vhost-vdpa Protocol

4.9.1 Introduction

vDPA(Virtual data path acceleration) device is a device that uses a datapath which complies with the virtio specifications with vendor specific control path. vDPA devices can be both physically located on the hardware or emulated by software.

This document describes the vDPA support in qemu

Here is the kernel commit here https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4c8cf31885f69e86be0b5b9e6677a26797365e1d

TODO : More information will add later

CHAPTER 5

QEMU System Emulation Guest Hardware Specifications

Contents:

5.1 POWER9 XIVE interrupt controller

The POWER9 processor comes with a new interrupt controller architecture, called XIVE as "eXternal Interrupt Virtualization Engine".

Compared to the previous architecture, the main characteristics of XIVE are to support a larger number of interrupt sources and to deliver interrupts directly to virtual processors without hypervisor assistance. This removes the context switches required for the delivery process.

5.1.1 XIVE architecture

The XIVE IC is composed of three sub-engines, each taking care of a processing layer of external interrupts:

- Interrupt Virtualization Source Engine (IVSE), or Source Controller (SC). These are found in PCI PHBs, in the Processor Service Interface (PSI) host bridge Controller, but also inside the main controller for the core IPIs and other sub-chips (NX, CAP, NPU) of the chip/processor. They are configured to feed the IVRE with events.
- Interrupt Virtualization Routing Engine (IVRE) or Virtualization Controller (VC). It handles event coalescing and perform interrupt routing by matching an event source number with an Event Notification Descriptor (END).
- Interrupt Virtualization Presentation Engine (IVPE) or Presentation Controller (PC). It maintains the interrupt context state of each thread and handles the delivery of the external interrupt to the thread.

```
            XIVE Interrupt Controller
            +-------------------------------------+      IPIs
            | +---------+ +----------+ +---------+ |     +-------+
            | |IVRE     | |Common Q  | |IVPE     |----> | CORES |
            | |     esb | |          | |         |----> |       |
            | |     eas | |  Bridge  | |   tctx  |----> |       |
            | |SC   end | |          | |    nvt  | |    |       |
```

(continues on next page)

(continued from previous page)

```
+------+    | +---------+ +----+----+ +--------+ |    +-+-+-+-+
| RAM  |    +------------------|-----------------+      | | |
|      |                       |                        | | |
|      |                       |                        | | |
|      |  +--------------------v------------------------v-v-v--+    other
|      <--+                     Power Bus                      +--> chips
|  esb |  +---------+-----------------------+------------------+
|  eas |            |                       |
|  end |         +--|------+                |
|  nvt |       +----+----+ |           +----+----+
+------+       |IVSE     | |           |IVSE     |
               |         | |           |         |
               | PQ-bits | |           | PQ-bits |
               | local   |-+           |  in VC  |
               +---------+             +---------+
                  PCIe                 NX,NPU,CAPI


PQ-bits: 2 bits source state machine (P:pending Q:queued)
esb: Event State Buffer (Array of PQ bits in an IVSE)
eas: Event Assignment Structure
end: Event Notification Descriptor
nvt: Notification Virtual Target
tctx: Thread interrupt Context registers
```

XIVE internal tables

Each of the sub-engines uses a set of tables to redirect interrupts from event sources to CPU threads.

```
                                        +-------+
User or O/S                             |  EQ   |
    or                          +------>|entries|
Hypervisor                      |       |  ..   |
  Memory                        |       +-------+
                                |           ^
                                |           |
           +--------------------------------------------------+
                                |           |
Hypervisor      +------+    +---+--+    +---+--+   +------+
  Memory        | ESB  |    | EAT  |    | ENDT |   | NVTT |
 (skiboot)      +----+-+    +----+-+    +----+-+   +------+
                  ^  |        ^  |        ^  |       ^
                  |  |        |  |        |  |       |
           +--------------------------------------------------+
                  |  |        |  |        |  |       |
                  |  |        |  |        |  |       |
             +----|--|--------|--|--------|--|-+   +-|-----+    +------+
             |    |  |        |  |        |  | |   | | tctx|    |Thread|
 IPI or   ---+    +  v        +  v        +  v |---| +  .. |----->     |
HW events    |                                 |   |       |    |      |
             |             IVRE                |   | IVPE  |    +------+
             +---------------------------------+   +-------+
```

The IVSE have a 2-bits state machine, P for pending and Q for queued, for each source that allows events to be triggered. They are stored in an Event State Buffer (ESB) array and can be controlled by MMIOs.

If the event is let through, the IVRE looks up in the Event Assignment Structure (EAS) table for an Event Notification Descriptor (END) configured for the source. Each Event Notification Descriptor defines a notification path to a CPU and an in-memory Event Queue, in which will be enqueued an EQ data for the O/S to pull.

The IVPE determines if a Notification Virtual Target (NVT) can handle the event by scanning the thread contexts of the VCPUs dispatched on the processor HW threads. It maintains the interrupt context state of each thread in a NVT table.

XIVE thread interrupt context

The XIVE presenter can generate four different exceptions to its HW threads:

- hypervisor exception
- O/S exception
- Event-Based Branch (user level)
- msgsnd (doorbell)

Each exception has a state independent from the others called a Thread Interrupt Management context. This context is a set of registers which lets the thread handle priority management and interrupt acknowledgment among other things. The most important ones being :

- Interrupt Priority Register (PIPR)
- Interrupt Pending Buffer (IPB)
- Current Processor Priority (CPPR)
- Notification Source Register (NSR)

TIMA

The Thread Interrupt Management registers are accessible through a specific MMIO region, called the Thread Interrupt Management Area (TIMA), four aligned pages, each exposing a different view of the registers. First page (page address ending in `0b00`) gives access to the entire context and is reserved for the ring 0 view for the physical thread context. The second (page address ending in `0b01`) is for the hypervisor, ring 1 view. The third (page address ending in `0b10`) is for the operating system, ring 2 view. The fourth (page address ending in `0b11`) is for user level, ring 3 view.

Interrupt flow from an O/S perspective

After an event data has been enqueued in the O/S Event Queue, the IVPE raises the bit corresponding to the priority of the pending interrupt in the register IBP (Interrupt Pending Buffer) to indicate that an event is pending in one of the 8 priority queues. The Pending Interrupt Priority Register (PIPR) is also updated using the IPB. This register represent the priority of the most favored pending notification.

The PIPR is then compared to the Current Processor Priority Register (CPPR). If it is more favored (numerically less than), the CPU interrupt line is raised and the EO bit of the Notification Source Register (NSR) is updated to notify the presence of an exception for the O/S. The O/S acknowledges the interrupt with a special load in the Thread Interrupt Management Area.

The O/S handles the interrupt and when done, performs an EOI using a MMIO operation on the ESB management page of the associate source.

5.1.2 Overview of the QEMU models for XIVE

The XiveSource models the IVSE in general, internal and external. It handles the source ESBs and the MMIO interface to control them.

The XiveNotifier is a small helper interface interconnecting the XiveSource to the XiveRouter.

The XiveRouter is an abstract model acting as a combined IVRE and IVPE. It routes event notifications using the EAS and END tables to the IVPE sub-engine which does a CAM scan to find a CPU to deliver the exception. Storage should be provided by the inheriting classes.

XiveEnDSource is a special source object. It exposes the END ESB MMIOs of the Event Queues which are used for coalescing event notifications and for escalation. Not used on the field, only to sync the EQ cache in OPAL.

Finally, the XiveTCTX contains the interrupt state context of a thread, four sets of registers, one for each exception that can be delivered to a CPU. These contexts are scanned by the IVPE to find a matching VP when a notification is triggered. It also models the Thread Interrupt Management Area (TIMA), which exposes the thread context registers to the CPU for interrupt management.

5.2 XIVE for sPAPR (pseries machines)

The POWER9 processor comes with a new interrupt controller architecture, called XIVE as "eXternal Interrupt Virtualization Engine". It supports a larger number of interrupt sources and offers virtualization features which enables the HW to deliver interrupts directly to virtual processors without hypervisor assistance.

A QEMU `pseries` machine (which is PAPR compliant) using POWER9 processors can run under two interrupt modes:

- *Legacy Compatibility Mode*

  the hypervisor provides identical interfaces and similar functionality to PAPR+ Version 2.7. This is the default mode

  It is also referred as *XICS* in QEMU.

- *XIVE native exploitation mode*

  the hypervisor provides new interfaces to manage the XIVE control structures, and provides direct control for interrupt management through MMIO pages.

Which interrupt modes can be used by the machine is negotiated with the guest O/S during the Client Architecture Support negotiation sequence. The two modes are mutually exclusive.

Both interrupt mode share the same IRQ number space. See below for the layout.

5.2.1 CAS Negotiation

QEMU advertises the supported interrupt modes in the device tree property `ibm,arch-vec-5-platform-support` in byte 23 and the OS Selection for XIVE is indicated in the `ibm,architecture-vec-5` property byte 23.

The interrupt modes supported by the machine depend on the CPU type (POWER9 is required for XIVE) but also on the machine property `ic-mode` which can be set on the command line. It can take the following values: `xics`, `xive`, and `dual` which is the default mode. `dual` means that both modes XICS **and** XIVE are supported and if the guest OS supports XIVE, this mode will be selected.

The choosen interrupt mode is activated after a reconfiguration done in a machine reset.

5.2.2 KVM negotiation

When the guest starts under KVM, the capabilities of the host kernel and QEMU are also negotiated. Depending on the version of the host kernel, KVM will advertise the XIVE capability to QEMU or not.

Nevertheless, the available interrupt modes in the machine should not depend on the XIVE KVM capability of the host. On older kernels without XIVE KVM support, QEMU will use the emulated XIVE device as a fallback and on newer kernels (>=5.2), the KVM XIVE device.

XIVE native exploitation mode is not supported for KVM nested guests, VMs running under a L1 hypervisor (KVM on pSeries). In that case, the hypervisor will not advertise the KVM capability and QEMU will use the emulated XIVE device, same as for older versions of KVM.

As a final refinement, the user can also switch the use of the KVM device with the machine option `kernel_irqchip`.

XIVE support in KVM

For guest OSes supporting XIVE, the resulting interrupt modes on host kernels with XIVE KVM support are the following:

ic-mode	kernel_irqchip		
/	allowed (default)	off	on
dual (default)	XIVE KVM	XIVE emul.	XIVE KVM
xive	XIVE KVM	XIVE emul.	XIVE KVM
xics	XICS KVM	XICS emul.	XICS KVM

For legacy guest OSes without XIVE support, the resulting interrupt modes are the following:

ic-mode	kernel_irqchip		
/	allowed (default)	off	on
dual (default)	XICS KVM	XICS emul.	XICS KVM
xive	QEMU error(3)	QEMU error(3)	QEMU error(3)
xics	XICS KVM	XICS emul.	XICS KVM

(3) QEMU fails at CAS with `Guest requested unavailable interrupt mode (XICS), either don't set the ic-mode machine property or try ic-mode=xics or ic-mode=dual`

No XIVE support in KVM

For guest OSes supporting XIVE, the resulting interrupt modes on host kernels without XIVE KVM support are the following:

ic-mode	kernel_irqchip		
/	allowed (default)	off	on
dual (default)	XIVE emul.(1)	XIVE emul.	QEMU error (2)
xive	XIVE emul.(1)	XIVE emul.	QEMU error (2)
xics	XICS KVM	XICS emul.	XICS KVM

(1) QEMU warns with `warning: kernel_irqchip requested but unavailable: IRQ_XIVE capability must be present for KVM` In some cases (old host kernels or KVM nested guests), one may hit a QEMU/KVM incompatibility due to device destruction in reset. QEMU fails with `KVM is incompatible with ic-mode=dual,kernel-irqchip=on`

(2) QEMU fails with `kernel_irqchip requested but unavailable: IRQ_XIVE capability must be present for KVM`

For legacy guest OSes without XIVE support, the resulting interrupt modes are the following:

ic-mode	kernel_irqchip		
/	allowed (default)	off	on
dual (default)	QEMU error(4)	XICS emul.	QEMU error(4)
xive	QEMU error(3)	QEMU error(3)	QEMU error(3)
xics	XICS KVM	XICS emul.	XICS KVM

(3) QEMU fails at CAS with `Guest requested unavailable interrupt mode (XICS), either don't set the ic-mode machine property or try ic-mode=xics or ic-mode=dual`

(4) QEMU/KVM incompatibility due to device destruction in reset. QEMU fails with `KVM is incompatible with ic-mode=dual,kernel-irqchip=on`

5.2.3 XIVE Device tree properties

The properties for the PAPR interrupt controller node when the *XIVE native exploitation mode* is selected shoud contain:

- `device_type`

  value should be "power-ivpe".

- `compatible`

  value should be "ibm,power-ivpe".

- `reg`

  contains the base address and size of the thread interrupt managnement areas (TIMA), for the User level and for the Guest OS level. Only the Guest OS level is taken into account today.

- `ibm,xive-eq-sizes`

  the size of the event queues. One cell per size supported, contains log2 of size, in ascending order.

- `ibm,xive-lisn-ranges`

  the IRQ interrupt number ranges assigned to the guest for the IPIs.

The root node also exports :

- `ibm,plat-res-int-priorities`

  contains a list of priorities that the hypervisor has reserved for its own use.

5.2.4 IRQ number space

IRQ Number space of the `pseries` machine is 8K wide and is the same for both interrupt mode. The different ranges are defined as follow :

- `0x0000 .. 0x0FFF` 4K CPU IPIs (only used under XIVE)
- `0x1000 .. 0x1000` 1 EPOW
- `0x1001 .. 0x1001` 1 HOTPLUG
- `0x1002 .. 0x10FF` unused

- `0x1100 .. 0x11FF` 256 VIO devices
- `0x1200 .. 0x127F` 32x4 LSIs for PHB devices
- `0x1280 .. 0x12FF` unused
- `0x1300 .. 0x1FFF` PHB MSIs (dynamically allocated)

5.2.5 Monitoring XIVE

The state of the XIVE interrupt controller can be queried through the monitor commands `info pic`. The output comes in two parts.

First, the state of the thread interrupt context registers is dumped for each CPU :

```
(qemu) info pic
CPU[0000]:   QW   NSR CPPR IPB LSMFB ACK# INC AGE PIPR  W2
CPU[0000]: USER    00   00  00    00   00  00  00   00  00000000
CPU[0000]:   OS    00   ff  00    00   ff  00  ff   ff  80000400
CPU[0000]: POOL    00   00  00    00   00  00  00   00  00000000
CPU[0000]: PHYS    00   00  00    00   00  00  00   ff  00000000
...
```

In the case of a `pseries` machine, QEMU acts as the hypervisor and only the O/S and USER register rings make sense. `W2` contains the vCPU CAM line which is set to the VP identifier.

Then comes the routing information which aggregates the EAS and the END configuration:

```
...
LISN          PQ    EISN     CPU/PRIO EQ
00000000 MSI --    00000010   0/6    380/16384 @1fe3e0000 ^1 [ 80000010 ... ]
00000001 MSI --    00000010   1/6    305/16384 @1fc230000 ^1 [ 80000010 ... ]
00000002 MSI --    00000010   2/6    220/16384 @1fc2f0000 ^1 [ 80000010 ... ]
00000003 MSI --    00000010   3/6    201/16384 @1fc390000 ^1 [ 80000010 ... ]
00000004 MSI -Q  M 00000000
00000005 MSI -Q  M 00000000
00000006 MSI -Q  M 00000000
00000007 MSI -Q  M 00000000
00001000 MSI --    00000012   0/6    380/16384 @1fe3e0000 ^1 [ 80000010 ... ]
00001001 MSI --    00000013   0/6    380/16384 @1fe3e0000 ^1 [ 80000010 ... ]
00001100 MSI --    00000100   1/6    305/16384 @1fc230000 ^1 [ 80000010 ... ]
00001101 MSI -Q  M 00000000
00001200 LSI -Q  M 00000000
00001201 LSI -Q  M 00000000
00001202 LSI -Q  M 00000000
00001203 LSI -Q  M 00000000
00001300 MSI --    00000102   1/6    305/16384 @1fc230000 ^1 [ 80000010 ... ]
00001301 MSI --    00000103   2/6    220/16384 @1fc2f0000 ^1 [ 80000010 ... ]
00001302 MSI --    00000104   3/6    201/16384 @1fc390000 ^1 [ 80000010 ... ]
```

The source information and configuration:

- The `LISN` column outputs the interrupt number of the source in range `[ 0x0 ... 0x1FFF ]` and its type : `MSI` or `LSI`
- The `PQ` column reflects the state of the PQ bits of the source :
  - `--` source is ready to take events
  - `P-` an event was sent and an EOI is PENDING

    - `PQ` an event was QUEUED
    - `-Q` source is OFF

  a `M` indicates that source is *MASKED* at the EAS level,

The targeting configuration :

- The `EISN` column is the event data that will be queued in the event queue of the O/S.
- The `CPU/PRIO` column is the tuple defining the CPU number and priority queue serving the source.
- The `EQ` column outputs :
    - the current index of the event queue/ the max number of entries
    - the O/S event queue address
    - the toggle bit
    - the last entries that were pushed in the event queue.

5.3 NUMA mechanics for sPAPR (pseries machines)

NUMA in sPAPR works different than the System Locality Distance Information Table (SLIT) in ACPI. The logic is explained in the LOPAPR 1.1 chapter 15, "Non Uniform Memory Access (NUMA) Option". This document aims to complement this specification, providing details of the elements that impacts how QEMU views NUMA in pseries.

5.3.1 Associativity and ibm,associativity property

Associativity is defined as a group of platform resources that has similar mean performance (or in our context here, distance) relative to everyone else outside of the group.

The format of the ibm,associativity property varies with the value of bit 0 of byte 5 of the ibm,architecture-vec-5 property. The format with bit 0 equal to zero is deprecated. The current format, with the bit 0 with the value of one, makes ibm,associativity property represent the physical hierarchy of the platform, as one or more lists that starts with the highest level grouping up to the smallest. Considering the following topology:

```
Mem M1 ---- Proc P1    |
-----------------      | Socket S1  ---|
      chip C1          |               |
                                       | HW module 1 (MOD1)
Mem M2 ---- Proc P2    |               |
-----------------      | Socket S2  ---|
      chip C2          |
```

The ibm,associativity property for the processors would be:

- P1: {MOD1, S1, C1, P1}
- P2: {MOD1, S2, C2, P2}

Each allocable resource has an ibm,associativity property. The LOPAPR specification allows multiple lists to be present in this property, considering that the same resource can have multiple connections to the platform.

5.3.2 Relative Performance Distance and ibm,associativity-reference-points

The ibm,associativity-reference-points property is an array that is used to define the relevant performance/distance related boundaries, defining the NUMA levels for the platform.

The definition of its elements also varies with the value of bit 0 of byte 5 of the ibm,architecture-vec-5 property. The format with bit 0 equal to zero is also deprecated. With the current format, each integer of the ibm,associativity-reference-points represents an 1 based ordinal index (i.e. the first element is 1) of the ibm,associativity array. The first boundary is the most significant to application performance, followed by less significant boundaries. Allocated resources that belongs to the same performance boundaries are expected to have relative NUMA distance that matches the relevancy of the boundary itself. Resources that belongs to the same first boundary will have the shortest distance from each other. Subsequent boundaries represents greater distances and degraded performance.

Using the previous example, the following setting reference points defines three NUMA levels:

- ibm,associativity-reference-points = {0x3, 0x2, 0x1}

The first NUMA level (0x3) is interpreted as the third element of each ibm,associativity array, the second level is the second element and the third level is the first element. Let's also consider that elements belonging to the first NUMA level have distance equal to 10 from each other, and each NUMA level doubles the distance from the previous. This means that the second would be 20 and the third level 40. For the P1 and P2 processors, we would have the following NUMA levels:

```
* ibm,associativity-reference-points = {0x3, 0x2, 0x1}

* P1: associativity{MOD1, S1, C1, P1}

First NUMA level (0x3) => associativity[2] = C1
Second NUMA level (0x2) => associativity[1] = S1
Third NUMA level (0x1) => associativity[0] = MOD1

* P2: associativity{MOD1, S2, C2, P2}

First NUMA level (0x3) => associativity[2] = C2
Second NUMA level (0x2) => associativity[1] = S2
Third NUMA level (0x1) => associativity[0] = MOD1

P1 and P2 have the same third NUMA level, MOD1: Distance between them = 40
```

Changing the ibm,associativity-reference-points array changes the performance distance attributes for the same associativity arrays, as the following example illustrates:

```
* ibm,associativity-reference-points = {0x2}

* P1: associativity{MOD1, S1, C1, P1}

First NUMA level (0x2) => associativity[1] = S1

* P2: associativity{MOD1, S2, C2, P2}

First NUMA level (0x2) => associativity[1] = S2

P1 and P2 does not have a common performance boundary. Since this is a one level
NUMA configuration, distance between them is one boundary above the first
level, 20.
```

In a hypothetical platform where all resources inside the same hardware module is considered to be on the same performance boundary:

```
* ibm,associativity-reference-points = {0x1}

* P1: associativity{MOD1, S1, C1, P1}

First NUMA level (0x1) => associativity[0] = MOD0

* P2: associativity{MOD1, S2, C2, P2}

First NUMA level (0x1) => associativity[0] = MOD0

P1 and P2 belongs to the same first order boundary. The distance between then
is 10.
```

5.4 How the pseries Linux guest calculates NUMA distances

Another key difference between ACPI SLIT and the LOPAPR regarding NUMA is how the distances are expressed. The SLIT table provides the NUMA distance value between the relevant resources. LOPAPR does not provide a standard way to calculate it. We have the ibm,associativity for each resource, which provides a common-performance hierarchy, and the ibm,associativity-reference-points array that tells which level of associativity is considered to be relevant or not.

The result is that each OS is free to implement and to interpret the distance as it sees fit. For the pseries Linux guest, each level of NUMA duplicates the distance of the previous level, and the maximum amount of levels is limited to MAX_DISTANCE_REF_POINTS = 4 (from arch/powerpc/mm/numa.c in the kernel tree). This results in the following distances:

- both resources in the first NUMA level: 10
- resources one NUMA level apart: 20
- resources two NUMA levels apart: 40
- resources three NUMA levels apart: 80
- resources four NUMA levels apart: 160

5.4.1 Consequences for QEMU NUMA tuning

The way the pseries Linux guest calculates NUMA distances has a direct effect on what QEMU users can expect when doing NUMA tuning. As of QEMU 5.1, this is the default ibm,associativity-reference-points being used in the pseries machine:

ibm,associativity-reference-points = {0x4, 0x4, 0x2}

The first and second level are equal, 0x4, and a third one was added in commit a6030d7e0b35 exclusively for NVLink GPUs support. This means that regardless of how the ibm,associativity properties are being created in the device tree, the pseries Linux guest will only recognize three scenarios as far as NUMA distance goes:

- if the resources belongs to the same first NUMA level = 10
- second level is skipped since it's equal to the first
- all resources that aren't a NVLink GPU, it is guaranteed that they will belong to the same third NUMA level, having distance = 40
- for NVLink GPUs, distance = 80 from everything else

In short, we can summarize the NUMA distances seem in pseries Linux guests, using QEMU up to 5.1, as follows:

- local distance, i.e. the distance of the resource to its own NUMA node: 10
- if it's a NVLink GPU device, distance: 80
- every other resource, distance: 40

This also means that user input in QEMU command line does not change the NUMA distancing inside the guest for the pseries machine.

5.5 QEMU and ACPI BIOS Generic Event Device interface

The ACPI *Generic Event Device* (GED) is a HW reduced platform specific device introduced in ACPI v6.1 that handles all platform events, including the hotplug ones. GED is modelled as a device in the namespace with a _HID defined to be ACPI0013. This document describes the interface between QEMU and the ACPI BIOS.

GED allows HW reduced platforms to handle interrupts in ACPI ASL statements. It follows a very similar approach to the _EVT method from GPIO events. All interrupts are listed in _CRS and the handler is written in _EVT method. However, the QEMU implementation uses a single interrupt for the GED device, relying on an IO memory region to communicate the type of device affected by the interrupt. This way, we can support up to 32 events with a unique interrupt.

Here is an example,

```
Device (\_SB.GED)
{
    Name (_HID, "ACPI0013")
    Name (_UID, Zero)
    Name (_CRS, ResourceTemplate ()
    {
        Interrupt (ResourceConsumer, Edge, ActiveHigh, Exclusive, ,, )
        {
            0x00000029,
        }
    })
    OperationRegion (EREG, SystemMemory, 0x09080000, 0x04)
    Field (EREG, DWordAcc, NoLock, WriteAsZeros)
    {
        ESEL,   32
    }
    Method (_EVT, 1, Serialized)
    {
        Local0 = ESEL // ESEL = IO memory region which specifies the
                      // device type.
        If (((Local0 & One) == One))
        {
            MethodEvent1()
        }
        If ((Local0 & 0x2) == 0x2)
        {
            MethodEvent2()
        }
        ...
    }
}
```

5.5.1 GED IO interface (4 byte access)

read access:

```
[0x0-0x3] Event selector bit field (32 bit) set by QEMU.

 bits:
    0: Memory hotplug event
    1: System power down event
    2: NVDIMM hotplug event
 3-31: Reserved
```

write_access:

Nothing is expected to be written into GED IO memory

5.6 QEMU TPM Device

5.6.1 Guest-side hardware interface

TIS interface

The QEMU TPM emulation implements a TPM TIS hardware interface following the Trusted Computing Group's specification "TCG PC Client Specific TPM Interface Specification (TIS)", Specification Version 1.3, 21 March 2013. (see the TIS specification, or a later version of it).

The TIS interface makes a memory mapped IO region in the area 0xfed40000-0xfed44fff available to the guest operating system.

QEMU files related to TPM TIS interface:

- `hw/tpm/tpm_tis_common.c`
- `hw/tpm/tpm_tis_isa.c`
- `hw/tpm/tpm_tis_sysbus.c`
- `hw/tpm/tpm_tis.h`

Both an ISA device and a sysbus device are available. The former is used with pc/q35 machine while the latter can be instantiated in the Arm virt machine.

CRB interface

QEMU also implements a TPM CRB interface following the Trusted Computing Group's specification "TCG PC Client Platform TPM Profile (PTP) Specification", Family "2.0", Level 00 Revision 01.03 v22, May 22, 2017. (see the CRB specification, or a later version of it)

The CRB interface makes a memory mapped IO region in the area 0xfed40000-0xfed40fff (1 locality) available to the guest operating system.

QEMU files related to TPM CRB interface:

- `hw/tpm/tpm_crb.c`

SPAPR interface

pSeries (ppc64) machines offer a tpm-spapr device model.

QEMU files related to the SPAPR interface:

- hw/tpm/tpm_spapr.c

5.6.2 fw_cfg interface

The bios/firmware may read the "etc/tpm/config" fw_cfg entry for configuring the guest appropriately.

The entry of 6 bytes has the following content, in little-endian:

```
#define TPM_VERSION_UNSPEC          0
#define TPM_VERSION_1_2             1
#define TPM_VERSION_2_0             2

#define TPM_PPI_VERSION_NONE        0
#define TPM_PPI_VERSION_1_30        1

struct FwCfgTPMConfig {
    uint32_t tpmppi_address;         /* PPI memory location */
    uint8_t tpm_version;             /* TPM version */
    uint8_t tpmppi_version;          /* PPI version */
};
```

5.6.3 ACPI interface

The TPM device is defined with ACPI ID "PNP0C31". QEMU builds a SSDT and passes it into the guest through the fw_cfg device. The device description contains the base address of the TIS interface 0xfed40000 and the size of the MMIO area (0x5000). In case a TPM2 is used by QEMU, a TPM2 ACPI table is also provided. The device is described to be used in polling mode rather than interrupt mode primarily because no unused IRQ could be found.

To support measurement logs to be written by the firmware, e.g. SeaBIOS, a TCPA table is implemented. This table provides a 64kb buffer where the firmware can write its log into. For TPM 2 only a more recent version of the TPM2 table provides support for measurements logs and a TCPA table does not need to be created.

The TCPA and TPM2 ACPI tables follow the Trusted Computing Group specification "TCG ACPI Specification" Family "1.2" and "2.0", Level 00 Revision 00.37. (see the ACPI specification, or a later version of it)

ACPI PPI Interface

QEMU supports the Physical Presence Interface (PPI) for TPM 1.2 and TPM 2. This interface requires ACPI and firmware support. (see the PPI specification)

PPI enables a system administrator (root) to request a modification to the TPM upon reboot. The PPI specification defines the operation requests and the actions the firmware has to take. The system administrator passes the operation request number to the firmware through an ACPI interface which writes this number to a memory location that the firmware knows. Upon reboot, the firmware finds the number and sends commands to the TPM. The firmware writes the TPM result code and the operation request number to a memory location that ACPI can read from and pass the result on to the administrator.

The PPI specification defines a set of mandatory and optional operations for the firmware to implement. The ACPI interface also allows an administrator to list the supported operations. In QEMU the ACPI code is generated by QEMU, yet the firmware needs to implement support on a per-operations basis, and different firmwares may support a different subset. Therefore, QEMU introduces the virtual memory device for PPI where the firmware can indicate which operations it supports and ACPI can enable the ones that are supported and disable all others. This interface lies in main memory and has the following layout:

Field	Length	Offset	Description
`func`	0x100	0x000	Firmware sets values for each supported operation. See defined values below.
`ppin`	0x1	0x100	SMI interrupt to use. Set by firmware. Not supported.
`ppip`	0x4	0x101	ACPI function index to pass to SMM code. Set by ACPI. Not supported.
`pprp`	0x4	0x105	Result of last executed operation. Set by firmware. See function index 5 for values.
`pprq`	0x4	0x109	Operation request number to execute. See 'Physical Presence Interface Operation Summary' tables in specs. Set by ACPI.
`pprm`	0x4	0x10d	Operation request optional parameter. Values depend on operation. Set by ACPI.
`lppr`	0x4	0x111	Last executed operation request number. Copied from pprq field by firmware.
`fret`	0x4	0x115	Result code from SMM function. Not supported.
`res1`	0x40	0x119	Reserved for future use
`next_step`	0x1	0x159	Operation to execute after reboot by firmware. Used by firmware.
`movv`	0x1	0x15a	Memory overwrite variable

The following values are supported for the `func` field. They correspond to the values used by ACPI function index 8.

Value	Description
0	Operation is not implemented.
1	Operation is only accessible through firmware.
2	Operation is blocked for OS by firmware configuration.
3	Operation is allowed and physically present user required.
4	Operation is allowed and physically present user is not required.

The location of the table is given by the fw_cfg `tpmppi_address` field. The PPI memory region size is 0x400 (`TPM_PPI_ADDR_SIZE`) to leave enough room for future updates.

QEMU files related to TPM ACPI tables:

- `hw/i386/acpi-build.c`
- `include/hw/acpi/tpm.h`

5.6.4 TPM backend devices

The TPM implementation is split into two parts, frontend and backend. The frontend part is the hardware interface, such as the TPM TIS interface described earlier, and the other part is the TPM backend interface. The backend interfaces implement the interaction with a TPM device, which may be a physical or an emulated device. The split between the front- and backend devices allows a frontend to be connected with any available backend. This enables the TIS interface to be used with the passthrough backend or the swtpm backend.

QEMU files related to TPM backends:

- `backends/tpm.c`
- `include/sysemu/tpm.h`

- `include/sysemu/tpm_backend.h`

The QEMU TPM passthrough device

In case QEMU is run on Linux as the host operating system it is possible to make the hardware TPM device available to a single QEMU guest. In this case the user must make sure that no other program is using the device, e.g., /dev/tpm0, before trying to start QEMU with it.

The passthrough driver uses the host's TPM device for sending TPM commands and receiving responses from. Besides that it accesses the TPM device's sysfs entry for support of command cancellation. Since none of the state of a hardware TPM can be migrated between hosts, virtual machine migration is disabled when the TPM passthrough driver is used.

Since the host's TPM device will already be initialized by the host's firmware, certain commands, e.g. `TPM_Startup()`, sent by the virtual firmware for device initialization, will fail. In this case the firmware should not use the TPM.

Sharing the device with the host is generally not a recommended usage scenario for a TPM device. The primary reason for this is that two operating systems can then access the device's single set of resources, such as platform configuration registers (PCRs). Applications or kernel security subsystems, such as the Linux Integrity Measurement Architecture (IMA), are not expecting to share PCRs.

QEMU files related to the TPM passthrough device:

- `backends/tpm/tpm_passthrough.c`
- `backends/tpm/tpm_util.c`
- `include/sysemu/tpm_util.h`

Command line to start QEMU with the TPM passthrough device using the host's hardware TPM `/dev/tpm0`:

```
qemu-system-x86_64 -display sdl -accel kvm \
-m 1024 -boot d -bios bios-256k.bin -boot menu=on \
-tpmdev passthrough,id=tpm0,path=/dev/tpm0 \
-device tpm-tis,tpmdev=tpm0 test.img
```

The following commands should result in similar output inside the VM with a Linux kernel that either has the TPM TIS driver built-in or available as a module:

```
# dmesg | grep -i tpm
[    0.711310] tpm_tis 00:06: 1.2 TPM (device=id 0x1, rev-id 1)

# dmesg | grep TCPA
[    0.000000] ACPI: TCPA 0x0000000003FFD191C 000032 (v02 BOCHS  \
    BXPCTCPA 0000001 BXPC 00000001)

# ls -l /dev/tpm*
crw-------. 1 root root 10, 224 Jul 11 10:11 /dev/tpm0

# find /sys/devices/ | grep pcrs$ | xargs cat
PCR-00: 35 4E 3B CE 23 9F 38 59 ...
...
PCR-23: 00 00 00 00 00 00 00 00 ...
```

The QEMU TPM emulator device

The TPM emulator device uses an external TPM emulator called 'swtpm' for sending TPM commands to and receiving responses from. The swtpm program must have been started before trying to access it through the TPM emulator with QEMU.

The TPM emulator implements a command channel for transferring TPM commands and responses as well as a control channel over which control commands can be sent. (see the SWTPM protocol specification)

The control channel serves the purpose of resetting, initializing, and migrating the TPM state, among other things.

The swtpm program behaves like a hardware TPM and therefore needs to be initialized by the firmware running inside the QEMU virtual machine. One necessary step for initializing the device is to send the TPM_Startup command to it. SeaBIOS, for example, has been instrumented to initialize a TPM 1.2 or TPM 2 device using this command.

QEMU files related to the TPM emulator device:

- `backends/tpm/tpm_emulator.c`
- `backends/tpm/tpm_util.c`
- `include/sysemu/tpm_util.h`

The following commands start the swtpm with a UnixIO control channel over a socket interface. They do not need to be run as root.

```
mkdir /tmp/mytpm1
swtpm socket --tpmstate dir=/tmp/mytpm1 \
  --ctrl type=unixio,path=/tmp/mytpm1/swtpm-sock \
  --log level=20
```

Command line to start QEMU with the TPM emulator device communicating with the swtpm (x86):

```
qemu-system-x86_64 -display sdl -accel kvm \
  -m 1024 -boot d -bios bios-256k.bin -boot menu=on \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis,tpmdev=tpm0 test.img
```

In case a pSeries machine is emulated, use the following command line:

```
qemu-system-ppc64 -display sdl -machine pseries,accel=kvm \
  -m 1024 -bios slof.bin -boot menu=on \
  -nodefaults -device VGA -device pci-ohci -device usb-kbd \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-spapr,tpmdev=tpm0 \
  -device spapr-vscsi,id=scsi0,reg=0x00002000 \
  -device virtio-blk-pci,scsi=off,bus=pci.0,addr=0x3,drive=drive-virtio-disk0,
↪id=virtio-disk0 \
  -drive file=test.img,format=raw,if=none,id=drive-virtio-disk0
```

In case an Arm virt machine is emulated, use the following command line:

```
qemu-system-aarch64 -machine virt,gic-version=3,accel=kvm \
  -cpu host -m 4G \
  -nographic -no-acpi \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis-device,tpmdev=tpm0 \
```

(continues on next page)

(continued from previous page)

```
  -device virtio-blk-pci,drive=drv0 \
  -drive format=qcow2,file=hda.qcow2,if=none,id=drv0 \
  -drive if=pflash,format=raw,file=flash0.img,readonly \
  -drive if=pflash,format=raw,file=flash1.img
```

In case SeaBIOS is used as firmware, it should show the TPM menu item after entering the menu with 'ESC'.

```
Select boot device:
1. DVD/CD [ata1-0: QEMU DVD-ROM ATAPI-4 DVD/CD]
[...]
5. Legacy option rom

t. TPM Configuration
```

The following commands should result in similar output inside the VM with a Linux kernel that either has the TPM TIS driver built-in or available as a module:

```
# dmesg | grep -i tpm
[    0.711310] tpm_tis 00:06: 1.2 TPM (device=id 0x1, rev-id 1)

# dmesg | grep TCPA
[    0.000000] ACPI: TCPA 0x0000000003FFD191C 000032 (v02 BOCHS  \
    BXPCTCPA 0000001 BXPC 00000001)

# ls -l /dev/tpm*
crw-------. 1 root root 10, 224 Jul 11 10:11 /dev/tpm0

# find /sys/devices/ | grep pcrs$ | xargs cat
PCR-00: 35 4E 3B CE 23 9F 38 59 ...
...
PCR-23: 00 00 00 00 00 00 00 00 ...
```

5.6.5 Migration with the TPM emulator

The TPM emulator supports the following types of virtual machine migration:

- VM save / restore (migration into a file)
- Network migration
- Snapshotting (migration into storage like QoW2 or QED)

The following command sequences can be used to test VM save / restore.

In a 1st terminal start an instance of a swtpm using the following command:

```
mkdir /tmp/mytpm1
swtpm socket --tpmstate dir=/tmp/mytpm1 \
  --ctrl type=unixio,path=/tmp/mytpm1/swtpm-sock \
  --log level=20 --tpm2
```

In a 2nd terminal start the VM:

```
qemu-system-x86_64 -display sdl -accel kvm \
  -m 1024 -boot d -bios bios-256k.bin -boot menu=on \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
```

(continues on next page)

(continued from previous page)

```
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis,tpmdev=tpm0 \
  -monitor stdio \
  test.img
```

Verify that the attached TPM is working as expected using applications inside the VM.

To store the state of the VM use the following command in the QEMU monitor in the 2nd terminal:

```
(qemu) migrate "exec:cat > testvm.bin"
(qemu) quit
```

At this point a file called `testvm.bin` should exists and the swtpm and QEMU processes should have ended.

To test 'VM restore' you have to start the swtpm with the same parameters as before. If previously a TPM 2 [–tpm2] was saved, –tpm2 must now be passed again on the command line.

In the 1st terminal restart the swtpm with the same command line as before:

```
swtpm socket --tpmstate dir=/tmp/mytpm1 \
  --ctrl type=unixio,path=/tmp/mytpm1/swtpm-sock \
  --log level=20 --tpm2
```

In the 2nd terminal restore the state of the VM using the additional '-incoming' option.

```
qemu-system-x86_64 -display sdl -accel kvm \
  -m 1024 -boot d -bios bios-256k.bin -boot menu=on \
  -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
  -tpmdev emulator,id=tpm0,chardev=chrtpm \
  -device tpm-tis,tpmdev=tpm0 \
  -incoming "exec:cat < testvm.bin" \
  test.img
```

Troubleshooting migration

There are several reasons why migration may fail. In case of problems, please ensure that the command lines adhere to the following rules and, if possible, that identical versions of QEMU and swtpm are used at all times.

VM save and restore:

- QEMU command line parameters should be identical apart from the '-incoming' option on VM restore
- swtpm command line parameters should be identical

VM migration to 'localhost':

- QEMU command line parameters should be identical apart from the '-incoming' option on the destination side
- swtpm command line parameters should point to two different directories on the source and destination swtpm (–tpmstate dir=. . .) (especially if different versions of libtpms were to be used on the same machine).

VM migration across the network:

- QEMU command line parameters should be identical apart from the '-incoming' option on the destination side
- swtpm command line parameters should be identical

VM Snapshotting:

- QEMU command line parameters should be identical

- swtpm command line parameters should be identical

Besides that, migration failure reasons on the swtpm level may include the following:

- the versions of the swtpm on the source and destination sides are incompatible
  - downgrading of TPM state may not be supported
  - the source and destination libtpms were compiled with different compile-time options and the destination side refuses to accept the state
- different migration keys are used on the source and destination side and the destination side cannot decrypt the migrated state (swtpm … –migration-key …)

5.7 APEI tables generating and CPER record

5.7.1 Design Details

```
       etc/acpi/tables                           etc/hardware_errors
    ====================                      ===============================
+ +--------------------------+            +----------------------------+
| | HEST                     | +--------->|    error_block_address1    |------+
| +--------------------------+ |          +----------------------------+      |
| | GHES1                    | | +------->|    error_block_address2    |------+-+
| +--------------------------+ | |        +----------------------------+      | |
| | .................        | | |        |      ..............        |      | |
| | error_status_address-----+-+ |        -----------------------------+      | |
| | .................        |   |   +--->|    error_block_addressN    |------+-+---+
| | read_ack_register--------+-+ |   |    +----------------------------+      | |   |
| | read_ack_preserve        | +-+---+--->|     read_ack_register1     |      | |   |
| | read_ack_write           |   |   |    +----------------------------+      | |   |
+ +--------------------------+   | +-+--->|     read_ack_register2     |      | |   |
| | GHES2                    |   | | |    +----------------------------+      | |   |
+ +--------------------------+   | | |    |       .............        |      | |   |
| | .................        |   | | |    +----------------------------+      | |   |
| | error_status_address-----+---+ | | +->|     read_ack_registerN     |      | |   |
| | .................        |     | | |  +----------------------------+      | |   |
| | read_ack_register--------+-----+ | |  |Generic Error Status Block 1|<-----+ |   |
| | read_ack_preserve        |       | |  |-+------------------------+-+        |   |
| | read_ack_write           |       | |  | |          CPER          | |        |   |
+ +--------------------------|       | |  | |          CPER          | |        |   |
| | ...............          |       | |  | |          ....          | |        |   |
+ +--------------------------+       | |  | |          CPER          | |        |   |
| | GHESN                    |       | |  |-+------------------------+-|        |   |
+ +--------------------------+       | |  |Generic Error Status Block 2|<-------+   |
| | .................        |       | |  |-+------------------------+-+            |
| | error_status_address-----+-------+ |  | |           CPER         | |            |
| | .................        |         |  | |           CPER         | |            |
| | read_ack_register--------+---------+  | |           ....         | |            |
| | read_ack_preserve        |            | |           CPER         | |            |
| | read_ack_write           |            +-+------------------------+-+            |
+ +--------------------------+            |         ..........         |            |
                                          |----------------------------+            |
                                          |Generic Error Status Block N |<----------+
                                          |-+-------------------------+-+
                                          | |          CPER           | |
                                          | |          CPER           | |
```

(continues on next page)

(continued from previous page)

```
                                                        | |          ....             | |
                                                        | |          CPER             | |
                                                        +-+--------------------------+-+
```

(1) QEMU generates the ACPI HEST table. This table goes in the current "etc/acpi/tables" fw_cfg blob. Each error source has different notification types.

(2) A new fw_cfg blob called "etc/hardware_errors" is introduced. QEMU also needs to populate this blob. The "etc/hardware_errors" fw_cfg blob contains an address registers table and an Error Status Data Block table.

(3) The address registers table contains N Error Block Address entries and N Read Ack Register entries. The size for each entry is 8-byte. The Error Status Data Block table contains N Error Status Data Block entries. The size for each entry is 4096(0x1000) bytes. The total size for the "etc/hardware_errors" fw_cfg blob is (N * 8 * 2 + N * 4096) bytes. N is the number of the kinds of hardware error sources.

(4) QEMU generates the ACPI linker/loader script for the firmware. The firmware pre-allocates memory for "etc/acpi/tables", "etc/hardware_errors" and copies blob contents there.

(5) QEMU generates N ADD_POINTER commands, which patch addresses in the "error_status_address" fields of the HEST table with a pointer to the corresponding "address registers" in the "etc/hardware_errors" blob.

(6) QEMU generates N ADD_POINTER commands, which patch addresses in the "read_ack_register" fields of the HEST table with a pointer to the corresponding "read_ack_register" within the "etc/hardware_errors" blob.

(7) QEMU generates N ADD_POINTER commands for the firmware, which patch addresses in the "error_block_address" fields with a pointer to the respective "Error Status Data Block" in the "etc/hardware_errors" blob.

(8) QEMU defines a third and write-only fw_cfg blob which is called "etc/hardware_errors_addr". Through that blob, the firmware can send back the guest-side allocation addresses to QEMU. The "etc/hardware_errors_addr" blob contains a 8-byte entry. QEMU generates a single WRITE_POINTER command for the firmware. The firmware will write back the start address of "etc/hardware_errors" blob to the fw_cfg file "etc/hardware_errors_addr".

(9) When QEMU gets a SIGBUS from the kernel, QEMU writes CPER into corresponding "Error Status Data Block", guest memory, and then injects platform specific interrupt (in case of arm/virt machine it's Synchronous External Abort) as a notification which is necessary for notifying the guest.

(10) This notification (in virtual hardware) will be handled by the guest kernel, on receiving notification, guest APEI driver could read the CPER error and take appropriate action.

(11) kvm_arch_on_sigbus_vcpu() uses source_id as index in "etc/hardware_errors" to find out "Error Status Data Block" entry corresponding to error source. So supported source_id values should be assigned here and not be changed afterwards to make sure that guest will write error into expected "Error Status Data Block" even if guest was migrated to a newer QEMU.

CHAPTER 6

QEMU Developer's Guide

This manual documents various parts of the internals of QEMU. You only need to read it if you are interested in reading or modifying QEMU's source code.

Contents:

6.1 The QEMU build system architecture

This document aims to help developers understand the architecture of the QEMU build system. As with projects using GNU autotools, 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. There is about where the similarities with GNU autotools end, so try to forget what you know about them.

6.1.1 Stage 1: configure

The QEMU configure script is written directly in shell, and should be compatible with any POSIX shell, hence it uses #!/bin/sh. An important implication of this is that it is important to avoid using bash-isms on development platforms where bash is the primary host.

In contrast to autoconf scripts, QEMU's configure is expected to be silent while it is checking for features. It will only display output when an error occurs, or to show the final feature enablement summary on completion.

Because QEMU uses the Meson build system under the hood, only VPATH builds are supported. There are two general ways to invoke configure & perform a build:

- VPATH, build artifacts outside of QEMU source tree entirely:

```
cd ../
mkdir build
cd build
../qemu/configure
make
```

- VPATH, build artifacts in a subdir of QEMU source tree:

```
mkdir build
cd build
../configure
make
```

For now, checks on the compilation environment are found in configure rather than meson.build, though this is expected to change. The command line is parsed in the configure script and, whenever needed, converted into the appropriate options to Meson.

New checks should be added to Meson, which usually comprises the following tasks:

- Add a Meson build option to meson_options.txt.
- Add support to the command line arg parser to handle any new *–enable-XXX*/*–disable-XXX* flags required by the feature.
- Add information to the help output message to report on the new feature flag.
- Add code to perform the actual feature check.
- Add code to include the feature status in *config-host.h*
- Add code to print out the feature status in the configure summary upon completion.

Taking the probe for SDL2_Image as an example, we have the following pieces in configure:

```
# Initial variable state
sdl_image=auto

..snip..

# Configure flag processing
--disable-sdl-image) sdl_image=disabled
;;
--enable-sdl-image) sdl_image=enabled
;;

..snip..

# Help output feature message
sdl-image         SDL Image support for icons

..snip..

# Meson invocation
-Dsdl_image=$sdl_image
```

In meson_options.txt:

```
option('sdl', type : 'feature', value : 'auto',
       description: 'SDL Image support for icons')
```

In meson.build:

```
# Detect dependency
sdl_image = dependency('SDL2_image', required: get_option('sdl_image'),
                       method: 'pkg-config',
                       static: enable_static)
```

(continues on next page)

(continued from previous page)

```
# Create config-host.h (if applicable)
config_host_data.set('CONFIG_SDL_IMAGE', sdl_image.found())

# Summary
summary_info += {'SDL image support': sdl_image.found()}
```

Helper functions

The configure script provides a variety of helper functions to assist developers in checking for system features:

do_cc $ARGS… Attempt to run the system C compiler passing it $ARGS…

do_cxx $ARGS… Attempt to run the system C++ compiler passing it $ARGS…

compile_object $CFLAGS Attempt to compile a test program with the system C compiler using $CFLAGS. The test program must have been previously written to a file called $TMPC.

compile_prog $CFLAGS $LDFLAGS Attempt to compile a test program with the system C compiler using $CFLAGS and link it with the system linker using $LDFLAGS. The test program must have been previously written to a file called $TMPC.

has $COMMAND Determine if $COMMAND exists in the current environment, either as a shell builtin, or executable binary, returning 0 on success.

path_of $COMMAND Return the fully qualified path of $COMMAND, printing it to stdout, and returning 0 on success.

check_define $NAME Determine if the macro $NAME is defined by the system C compiler

check_include $NAME Determine if the include $NAME file is available to the system C compiler

write_c_skeleton Write a minimal C program main() function to the temporary file indicated by $TMPC

feature_not_found $NAME $REMEDY Print a message to stderr that the feature $NAME was not available on the system, suggesting the user try $REMEDY to address the problem.

error_exit $MESSAGE $MORE… Print $MESSAGE to stderr, followed by $MORE… and then exit from the configure script with non-zero status

query_pkg_config $ARGS… Run pkg-config passing it $ARGS. If QEMU is doing a static build, then –static will be automatically added to $ARGS

6.1.2 Stage 2: Meson

The Meson build system is currently used to describe the build process for:

1) executables, which include:
   - Tools - qemu-img, qemu-nbd, qga (guest agent), etc
   - System emulators - qemu-system-$ARCH
   - Userspace emulators - qemu-$ARCH
   - Some (but not all) unit tests
2) documentation
3) ROMs, which can be either installed as binary blobs or compiled

4) other data files, such as icons or desktop files

The source code is highly modularized, split across many files to facilitate building of all of these components with as little duplicated compilation as possible. The Meson "sourceset" functionality is used to list the files and their dependency on various configuration symbols.

Various subsystems that are common to both tools and emulators have their own sourceset, for example *block_ss* for the block device subsystem, *chardev_ss* for the character device subsystem, etc. These sourcesets are then turned into static libraries as follows:

```
libchardev = static_library('chardev', chardev_ss.sources(),
                            name_suffix: 'fa',
                            build_by_default: false)

chardev = declare_dependency(link_whole: libchardev)
```

The special *.fa* suffix is needed as long as unit tests are built with the older Makefile infrastructure, and will go away later.

Files linked into emulator targets there can be split into two distinct groups of files, those which are independent of the QEMU emulation target and those which are dependent on the QEMU emulation target.

In the target-independent set lives various general purpose helper code, such as error handling infrastructure, standard data structures, platform portability wrapper functions, etc. This code can be compiled once only and the .o files linked into all output binaries. Target-independent code lives in the *common_ss*, *softmmu_ss* and *user_ss* sourcesets. *common_ss* is linked into all emulators, *softmmu_ss* only in system emulators, *user_ss* only in user-mode emulators.

In the target-dependent set lives CPU emulation, device emulation and much glue code. This sometimes also has to be compiled multiple times, once for each target being built.

All binaries link with a static library *libqemuutil.a*, which is then linked to all the binaries. *libqemuutil.a* is built from several sourcesets; most of them however host generated code, and the only two of general interest are *util_ss* and *stub_ss*.

The separation between these two is purely for documentation purposes. *util_ss* contains generic utility files. Even though this code is only linked in some binaries, sometimes it requires hooks only in some of these and depend on other functions that are not fully implemented by all QEMU binaries. *stub_ss* links dummy stubs that will only be linked into the binary if the real implementation is not present. In a way, the stubs can be thought of as a portable implementation of the weak symbols concept.

The following files concur in the definition of which files are linked into each emulator:

default-configs/*.mak The files under default-configs/ control what emulated hardware is built into each QEMU system and userspace emulator targets. They merely contain a list of config variable definitions like the machines that should be included. For example, default-configs/aarch64-softmmu.mak has:

```
include arm-softmmu.mak
CONFIG_XLNX_ZYNQMP_ARM=y
CONFIG_XLNX_VERSAL=y
```

/Kconfig These files are processed together with *default-configs/*.mak* and describe the dependencies between various features, subsystems and device models. They are described in kconfig.rst.

These files rarely need changing unless new devices / hardware need to be enabled for a particular system/userspace emulation target

Support scripts

Meson has a special convention for invoking Python scripts: if their first line is *#! /usr/bin/env python3* and the file is *not* executable, find_program() arranges to invoke the script under the same Python interpreter that was used to invoke Meson. This is the most common and preferred way to invoke support scripts from Meson build files, because it automatically uses the value of configure's –python= option.

In case the script is not written in Python, use a *#! /usr/bin/env …* line and make the script executable.

Scripts written in Python, where it is desirable to make the script executable (for example for test scripts that developers may want to invoke from the command line, such as tests/qapi-schema/test-qapi.py), should be invoked through the *python* variable in meson.build. For example:

```
test('QAPI schema regression tests', python,
     args: files('test-qapi.py'),
     env: test_env, suite: ['qapi-schema', 'qapi-frontend'])
```

This is needed to obey the –python= option passed to the configure script, which may point to something other than the first python3 binary on the path.

6.1.3 Stage 3: makefiles

The use of GNU make is required with the QEMU build system.

The output of Meson is a build.ninja file, which is used with the Ninja build system. QEMU uses a different approach, where Makefile rules are synthesized from the build.ninja file. The main Makefile includes these rules and wraps them so that e.g. submodules are built before QEMU. The resulting build system is largely non-recursive in nature, in contrast to common practices seen with automake.

Tests are also ran by the Makefile with the traditional *make check* phony target. Meson test suites such as *unit* can be ran with *make check-unit* too. It is also possible to run tests defined in meson.build with *meson test*.

The following text is only relevant for unit tests which still have to be converted to Meson.

All binaries should link to *libqemuutil.a*, e.g.:

> qemu-img$(EXESUF): qemu-img.o ..snip.. libqemuutil.a

On Windows, all binaries have the suffix *.exe*, so all Makefile rules which create binaries must include the $(EXESUF) variable on the binary name. e.g.

> qemu-img$(EXESUF): qemu-img.o ..snip..

This expands to *.exe* on Windows, or an empty string on other platforms.

Variable naming

The QEMU convention is to define variables to list different groups of object files. These are named with the convention $PREFIX-obj-y. The Meson *chardev* variable in the previous example corresponds to a variable 'chardev-obj-y'.

Likewise, tests that are executed by *make check-unit* are grouped into a variable check-unit-y, like this:

> check-unit-y += tests/test-visitor-serialization$(EXESUF) check-unit-y += tests/test-iov$(EXESUF) check-unit-y += tests/test-bitmap$(EXESUF)

When a test or object file which needs to be conditionally built based on some characteristic of the host system, the configure script will define a variable for the conditional. For example, on Windows it will define $(CONFIG_POSIX) with a value of 'n' and $(CONFIG_WIN32) with a value of 'y'. It is now possible to use the config variables when listing object files. For example, check-unit-$(CONFIG_POSIX) += tests/test-vmstate$(EXESUF)

On Windows this expands to

check-unit-n += tests/vmstate.exe

Since the *check-unit* target only runs tests included in *$(check-unit-y)*, POSIX specific tests listed in *$(util-obj-n)* are ignored on the Windows platform builds.

CFLAGS / LDFLAGS / LIBS handling

There are many different binaries being built with differing purposes, and some of them might even be 3rd party libraries pulled in via git submodules. As such the use of the global CFLAGS variable is generally avoided in QEMU, since it would apply to too many build targets.

Flags that are needed by any QEMU code (i.e. everything *except* GIT submodule projects) are put in $(QEMU_CFLAGS) variable. For linker flags the $(LIBS) variable is sometimes used, but a couple of more targeted variables are preferred.

In addition to these variables, it is possible to provide cflags and libs against individual source code files, by defining variables of the form $FILENAME-cflags and $FILENAME-libs. For example, the test test-crypto-tlscredsx509 needs to link to the libtasn1 library, so tests/Makefile.include defines some variables:

tests/crypto-tls-x509-helpers.o-cflags := $(TASN1_CFLAGS) tests/crypto-tls-x509-helpers.o-libs := $(TASN1_LIBS)

The scope is a little different between the two variables. The libs get used when linking any target binary that includes the curl.o object file, while the cflags get used when compiling the curl.c file only.

6.1.4 Important files for the build system

Statically defined files

The following key files are statically defined in the source tree, with the rules needed to build QEMU. Their behaviour is influenced by a number of dynamically created files listed later.

Makefile The main entry point used when invoking make to build all the components of QEMU. The default 'all' target will naturally result in the build of every component. Makefile takes care of recursively building submodules directly via a non-recursive set of rules.

/meson.build The meson.build file in the root directory is the main entry point for the Meson build system, and it coordinates the configuration and build of all executables. Build rules for various subdirectories are included in other meson.build files spread throughout the QEMU source tree.

rules.mak This file provides the generic helper rules for invoking build tools, in particular the compiler and linker.

tests/Makefile.include Rules for building the unit tests. This file is included directly by the top level Makefile, so anything defined in this file will influence the entire build system. Care needs to be taken when writing rules for tests to ensure they only apply to the unit test execution / build.

tests/docker/Makefile.include Rules for Docker tests. Like tests/Makefile, this file is included directly by the top level Makefile, anything defined in this file will influence the entire build system.

tests/vm/Makefile.include Rules for VM-based tests. Like tests/Makefile, this file is included directly by the top level Makefile, anything defined in this file will influence the entire build system.

Dynamically created files

The following files are generated dynamically by configure in order to control the behaviour of the statically defined makefiles. This avoids the need for QEMU makefiles to go through any pre-processing as seen with autotools, where Makefile.am generates Makefile.in which generates Makefile.

Built by configure:

config-host.mak When configure has determined the characteristics of the build host it will write a long list of variables to config-host.mak file. This provides the various install directories, compiler / linker flags and a variety of *CONFIG_** variables related to optionally enabled features. This is imported by the top level Makefile and meson.build in order to tailor the build output.

config-host.mak is also used as a dependency checking mechanism. If make sees that the modification timestamp on configure is newer than that on config-host.mak, then configure will be re-run.

The variables defined here are those which are applicable to all QEMU build outputs. Variables which are potentially different for each emulator target are defined by the next file…

$TARGET-NAME/config-target.mak TARGET-NAME is the name of a system or userspace emulator, for example, x86_64-softmmu denotes the system emulator for the x86_64 architecture. This file contains the variables which need to vary on a per-target basis. For example, it will indicate whether KVM or Xen are enabled for the target and any other potential custom libraries needed for linking the target.

Built by Meson:

***${TARGET-NAME}-config-devices.mak*** TARGET-NAME is again the name of a system or userspace emulator. The config-devices.mak file is automatically generated by make using the scripts/make_device_config.sh program, feeding it the default-configs/$TARGET-NAME file as input.

config-host.h, $TARGET-NAME/config-target.h, $TARGET-NAME/config-devices.h These files are used by source code to determine what features are enabled. They are generated from the contents of the corresponding **.h* files using the scripts/create_config program. This extracts relevant variables and formats them as C preprocessor macros.

build.ninja The build rules.

Built by Makefile:

Makefile.ninja A Makefile conversion of the build rules in build.ninja. The conversion is straightforward and, were it necessary to debug the rules produced by Meson, it should be enough to look at build.ninja. The conversion is performed by scripts/ninjatool.py.

Makefile.mtest The Makefile definitions that let "make check" run tests defined in meson.build. The rules are produced from Meson's JSON description of tests (obtained with "meson introspect –tests") through the script scripts/mtest2make.py.

Useful make targets

help Print a help message for the most common build targets.

print-VAR Print the value of the variable VAR. Useful for debugging the build system.

6.2 QEMU and Kconfig

QEMU is a very versatile emulator; it can be built for a variety of targets, where each target can emulate various boards and at the same time different targets can share large amounts of code. For example, a POWER and an x86 board can run the same code to emulate a PCI network card, even though the boards use different PCI host bridges, and they can run the same code to emulate a SCSI disk while using different SCSI adapters. Arm, s390 and x86 boards can all present a virtio-blk disk to their guests, but with three different virtio guest interfaces.

Each QEMU target enables a subset of the boards, devices and buses that are included in QEMU's source code. As a result, each QEMU executable only links a small subset of the files that form QEMU's source code; anything that is not needed to support a particular target is culled.

QEMU uses a simple domain-specific language to describe the dependencies between components. This is useful for two reasons:

- new targets and boards can be added without knowing in detail the architecture of the hardware emulation subsystems. Boards only have to list the components they need, and the compiled executable will include all the required dependencies and all the devices that the user can add to that board;
- users can easily build reduced versions of QEMU that support only a subset of boards or devices. For example, by default most targets will include all emulated PCI devices that QEMU supports, but the build process is configurable and it is easy to drop unnecessary (or otherwise unwanted) code to make a leaner binary.

This domain-specific language is based on the Kconfig language that originated in the Linux kernel, though it was heavily simplified and the handling of dependencies is stricter in QEMU.

Unlike Linux, there is no user interface to edit the configuration, which is instead specified in per-target files under the `default-configs/` directory of the QEMU source tree. This is because, unlike Linux, configuration and dependencies can be treated as a black box when building QEMU; the default configuration that QEMU ships with should be okay in almost all cases.

6.2.1 The Kconfig language

Kconfig defines configurable components in files named `hw/*/Kconfig`. Note that configurable components are _not_ visible in C code as preprocessor symbols; they are only visible in the Makefile. Each configurable component defines a Makefile variable whose name starts with `CONFIG_`.

All elements have boolean (true/false) type; truth is written as `y`, while falsehood is written `n`. They are defined in a Kconfig stanza like the following:

```
config ARM_VIRT
   bool
   imply PCI_DEVICES
   imply VFIO_AMD_XGBE
   imply VFIO_XGMAC
   select A15MPCORE
   select ACPI
   select ARM_SMMUV3
```

The `config` keyword introduces a new configuration element. In the example above, Makefiles will have access to a variable named `CONFIG_ARM_VIRT`, with value `y` or `n` (respectively for boolean true and false).

Boolean expressions can be used within the language, whenever `<expr>` is written in the remainder of this section. The `&&`, `||` and `!` operators respectively denote conjunction (AND), disjunction (OR) and negation (NOT).

The `bool` data type declaration is optional, but it is suggested to include it for clarity and future-proofing. After `bool` the following directives can be included:

dependencies: `depends on <expr>`

> This defines a dependency for this configurable element. Dependencies evaluate an expression and force the value of the variable to false if the expression is false.

reverse dependencies: `select <symbol> [if <expr>]`

While `depends on` can force a symbol to false, reverse dependencies can be used to force another symbol to true. In the following example, `CONFIG_BAZ` will be true whenever `CONFIG_FOO` is true:

```
config FOO
  select BAZ
```

The optional expression will prevent `select` from having any effect unless it is true.

Note that unlike Linux's Kconfig implementation, QEMU will detect contradictions between `depends on` and `select` statements and prevent you from building such a configuration.

default value: `default <value> [if <expr>]`

Default values are assigned to the config symbol if no other value was set by the user via `default-configs/*.mak` files, and only if `select` or `depends on` directives do not force the value to true or false respectively. `<value>` can be `y` or `n`; it cannot be an arbitrary Boolean expression. However, a condition for applying the default value can be added with `if`.

A configuration element can have any number of default values (usually, if more than one default is present, they will have different conditions). If multiple default values satisfy their condition, only the first defined one is active.

reverse default (weak reverse dependency): `imply <symbol> [if <expr>]`

This is similar to `select` as it applies a lower limit of `y` to another symbol. However, the lower limit is only a default and the "implied" symbol's value may still be set to `n` from a `default-configs/*.mak` files. The following two examples are equivalent:

```
config FOO
  bool
  imply BAZ

config BAZ
  bool
  default y if FOO
```

The next section explains where to use `imply` or `default y`.

6.2.2 Guidelines for writing Kconfig files

Configurable elements in QEMU fall under five broad groups. Each group declares its dependencies in different ways:

subsystems, of which **buses** are a special case

Example:

```
config SCSI
  bool
```

Subsystems always default to false (they have no `default` directive) and are never visible in `default-configs/*.mak` files. It's up to other symbols to `select` whatever subsystems they require.

They sometimes have `select` directives to bring in other required subsystems or buses. For example, `AUX` (the DisplayPort auxiliary channel "bus") selects `I2C` because it can act as an I2C master too.

devices

Example:

```
config MEGASAS_SCSI_PCI
  bool
  default y if PCI_DEVICES
  depends on PCI
  select SCSI
```

Devices are the most complex of the five. They can have a variety of directives that cooperate so that a default configuration includes all the devices that can be accessed from QEMU.

Devices *depend on* the bus that they lie on, for example a PCI device would specify `depends on PCI`. An MMIO device will likely have no `depends on` directive. Devices also *select* the buses that the device provides, for example a SCSI adapter would specify `select SCSI`. Finally, devices are usually `default y` if and only if they have at least one `depends on`; the default could be conditional on a device group.

Devices also select any optional subsystem that they use; for example a video card might specify `select EDID` if it needs to build EDID information and publish it to the guest.

device groups

Example:

```
config PCI_DEVICES
  bool
```

Device groups provide a convenient mechanism to enable/disable many devices in one go. This is useful when a set of devices is likely to be enabled/disabled by several targets. Device groups usually need no directive and are not used in the Makefile either; they only appear as conditions for `default y` directives.

QEMU currently has two device groups, `PCI_DEVICES` and `TEST_DEVICES`. PCI devices usually have a `default y if PCI_DEVICES` directive rather than just `default y`. This lets some boards (notably s390) easily support a subset of PCI devices, for example only VFIO (passthrough) and virtio-pci devices. `TEST_DEVICES` instead is used for devices that are rarely used on production virtual machines, but provide useful hooks to test QEMU or KVM.

boards

Example:

```
config SUN4M
  bool
  imply TCX
  imply CG3
  select CS4231
  select ECCMEMCTL
  select EMPTY_SLOT
  select ESCC
  select ESP
  select FDC
  select SLAVIO
  select LANCE
  select M48T59
  select STP2000
```

Boards specify their constituent devices using `imply` and `select` directives. A device should be listed under `select` if the board cannot be started at all without it. It should be listed under `imply` if (depending on the QEMU command line) the board may or may not be started without it. Boards also default to false; they are enabled by the `default-configs/*.mak` for the target they apply to.

internal elements

Example:

```
config ECCMEMCTL
  bool
  select ECC
```

Internal elements group code that is useful in several boards or devices. They are usually enabled with `select` and in turn select other elements; they are never visible in `default-configs/*.mak` files, and often not even in the Makefile.

6.2.3 Writing and modifying default configurations

In addition to the Kconfig files under hw/, each target also includes a file called `default-configs/TARGETNAME-softmmu.mak`. These files initialize some Kconfig variables to non-default values and provide the starting point to turn on devices and subsystems.

A file in `default-configs/` looks like the following example:

```
# Default configuration for alpha-softmmu

# Uncomment the following lines to disable these optional devices:
#
#CONFIG_PCI_DEVICES=n
#CONFIG_TEST_DEVICES=n

# Boards:
#
CONFIG_DP264=y
```

The first part, consisting of commented-out `=n` assignments, tells the user which devices or device groups are implied by the boards. The second part, consisting of `=y` assignments, tells the user which boards are supported by the target. The user will typically modify the default configuration by uncommenting lines in the first group, or commenting out lines in the second group.

It is also possible to run QEMU's configure script with the `--without-default-devices` option. When this is done, everything defaults to `n` unless it is `` select``ed or explicitly switched on in the ``.mak `` files. In other words, `default` and `imply` directives are disabled. When QEMU is built with this option, the user will probably want to change some lines in the first group, for example like this:

```
CONFIG_PCI_DEVICES=y
#CONFIG_TEST_DEVICES=n
```

and/or pick a subset of the devices in those device groups. Right now there is no single place that lists all the optional devices for `CONFIG_PCI_DEVICES` and `CONFIG_TEST_DEVICES`. In the future, we expect that `.mak` files will be automatically generated, so that they will include all these symbols and some help text on what they do.

6.2.4 `Kconfig.host`

In some special cases, a configurable element depends on host features that are detected by QEMU's configure script; for example some devices depend on the availability of KVM or on the presence of a library on the host.

These symbols should be listed in `Kconfig.host` like this:

```
config KVM
  bool
```

and also listed as follows in the top-level Makefile's `MINIKCONF_ARGS` variable:

```
MINIKCONF_ARGS = \
  $@ $*/config-devices.mak.d $< $(MINIKCONF_INPUTS) \
  CONFIG_KVM=$(CONFIG_KVM) \
  CONFIG_SPICE=$(CONFIG_SPICE) \
  CONFIG_TPM=$(CONFIG_TPM) \
  ...
```

6.3 Load and Store APIs

QEMU internally has multiple families of functions for performing loads and stores. This document attempts to enumerate them all and indicate when to use them. It does not provide detailed documentation of each API – for that you should look at the documentation comments in the relevant header files.

6.3.1 `ld*_p` and `st*_p`

These functions operate on a host pointer, and should be used when you already have a pointer into host memory (corresponding to guest ram or a local buffer). They deal with doing accesses with the desired endianness and with correctly handling potentially unaligned pointer values.

Function names follow the pattern:

load: `ld{type}{sign}{size}_{endian}_p(ptr)`

store: `st{type}{size}_{endian}_p(ptr, val)`

`type`

- (empty) : integer access
- `f` : float access

`sign`

- (empty) : for 32 or 64 bit sizes (including floats and doubles)
- `u` : unsigned
- `s` : signed

`size`

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

`endian`

- `he` : host endian
- `be` : big endian
- `le` : little endian

The `_{endian}` infix is omitted for target-endian accesses.

The target endian accessors are only available to source files which are built per-target.

There are also functions which take the size as an argument:

load: `ldn{endian}_p(ptr, sz)`

which performs an unsigned load of `sz` bytes from `ptr` as an `{endian}` order value and returns it in a uint64_t.

store: `stn{endian}_p(ptr, sz, val)`

which stores `val` to `ptr` as an `{endian}` order value of size `sz` bytes.

Regexes for git grep

- `\<ldf\?[us]\?[bwlq]\(_[hbl]e\)\?_p\>`
- `\<stf\?[bwlq]\(_[hbl]e\)\?_p\>`
- `\<ldn_\([hbl]e\)?_p\>`
- `\<stn_\([hbl]e\)?_p\>`

6.3.2 `cpu_{ld,st}*_mmuidx_ra`

These functions operate on a guest virtual address plus a context, known as a "mmu index" or `mmuidx`, which controls how that virtual address is translated. The meaning of the indexes are target specific, but specifying a particular index might be necessary if, for instance, the helper requires an "always as non-privileged" access rather that the default access for the current state of the guest CPU.

These functions may cause a guest CPU exception to be taken (e.g. for an alignment fault or MMU fault) which will result in guest CPU state being updated and control longjmp'ing out of the function call. They should therefore only be used in code that is implementing emulation of the guest CPU.

The `retaddr` parameter is used to control unwinding of the guest CPU state in case of a guest CPU exception. This is passed to `cpu_restore_state()`. Therefore the value should either be 0, to indicate that the guest CPU state is already synchronized, or the result of `GETPC()` from the top level `HELPER(foo)` function, which is a return address into the generated code.

Function names follow the pattern:

load: `cpu_ld{sign}{size}{end}_mmuidx_ra(env, ptr, mmuidx, retaddr)`

store: `cpu_st{size}{end}_mmuidx_ra(env, ptr, val, mmuidx, retaddr)`

`sign`

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

`size`

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

`end`

- (empty) : for target endian, or 8 bit sizes

- `_be` : big endian
- `_le` : little endian

Regexes for git grep:

- `\<cpu_ld[us]\?[bwlq](_[bl]e)\?_mmuidx_ra\>`
- `\<cpu_st[bwlq](_[bl]e)\?_mmuidx_ra\>`

6.3.3 `cpu_{ld,st}*_data_ra`

These functions work like the `cpu_{ld,st}_mmuidx_ra` functions except that the `mmuidx` parameter is taken from the current mode of the guest CPU, as determined by `cpu_mmu_index(env, false)`.

These are generally the preferred way to do accesses by guest virtual address from helper functions, unless the access should be performed with a context other than the default.

Function names follow the pattern:

load: `cpu_ld{sign}{size}{end}_data_ra(env, ptr, ra)`

store: `cpu_st{size}{end}_data_ra(env, ptr, val, ra)`

`sign`

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

`size`

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

`end`

- (empty) : for target endian, or 8 bit sizes
- `_be` : big endian
- `_le` : little endian

Regexes for git grep:

- `\<cpu_ld[us]\?[bwlq](_[bl]e)\?_data_ra\>`
- `\<cpu_st[bwlq](_[bl]e)\?_data_ra\>`

6.3.4 `cpu_{ld,st}*_data`

These functions work like the `cpu_{ld,st}_data_ra` functions except that the `retaddr` parameter is 0, and thus does not unwind guest CPU state.

This means they must only be used from helper functions where the translator has saved all necessary CPU state. These functions are the right choice for calls made from hooks like the CPU `do_interrupt` hook or when you know for certain that the translator had to save all the CPU state anyway.

Function names follow the pattern:

load: `cpu_ld{sign}{size}{end}_data(env, ptr)`

store: `cpu_st{size}{end}_data(env, ptr, val)`

`sign`

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

`size`

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

`end`

- (empty) : for target endian, or 8 bit sizes
- `_be` : big endian
- `_le` : little endian

Regexes for git grep

- `\<cpu_ld[us]\?[bwlq](_[bl]e)\?_data\>`
- `\<cpu_st[bwlq](_[bl]e)\?_data\+\>`

6.3.5 `cpu_ld*_code`

These functions perform a read for instruction execution. The `mmuidx` parameter is taken from the current mode of the guest CPU, as determined by `cpu_mmu_index(env, true)`. The `retaddr` parameter is 0, and thus does not unwind guest CPU state, because CPU state is always synchronized while translating instructions. Any guest CPU exception that is raised will indicate an instruction execution fault rather than a data read fault.

In general these functions should not be used directly during translation. There are wrapper functions that are to be used which also take care of plugins for tracing.

Function names follow the pattern:

load: `cpu_ld{sign}{size}_code(env, ptr)`

`sign`

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

`size`

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits

- `q` : 64 bits

Regexes for git grep:

- `\<cpu_ld[us]\?[bwlq]_code\>`

6.3.6 `translator_ld*`

These functions are a wrapper for `cpu_ld*_code` which also perform any actions required by any tracing plugins. They are only to be called during the translator callback `translate_insn`.

There is a set of functions ending in `_swap` which, if the parameter is true, returns the value in the endianness that is the reverse of the guest native endianness, as determined by `TARGET_WORDS_BIGENDIAN`.

Function names follow the pattern:

load: `translator_ld{sign}{size}(env, ptr)`

swap: `translator_ld{sign}{size}_swap(env, ptr, swap)`

`sign`

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

`size`

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

Regexes for git grep

- `\<translator_ld[us]\?[bwlq]\(_swap\)\?\>`

6.3.7 `helper_*_{ld,st}*_mmu`

These functions are intended primarily to be called by the code generated by the TCG backend. They may also be called by target CPU helper function code. Like the `cpu_{ld,st}_mmuidx_ra` functions they perform accesses by guest virtual address, with a given `mmuidx`.

These functions specify an `opindex` parameter which encodes (among other things) the mmu index to use for the access. This parameter should be created by calling `make_memop_idx()`.

The `retaddr` parameter should be the result of GETPC() called directly from the top level HELPER(foo) function (or 0 if no guest CPU state unwinding is required).

TODO The names of these functions are a bit odd for historical reasons because they were originally expected to be called only from within generated code. We should rename them to bring them more in line with the other memory access functions. The explicit endianness is the only feature they have beyond `*_mmuidx_ra`.

load: `helper_{endian}_ld{sign}{size}_mmu(env, addr, opindex, retaddr)`

store: `helper_{endian}_st{size}_mmu(env, addr, val, opindex, retaddr)`

`sign`

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned
- `s` : signed

`size`

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

`endian`

- `le` : little endian
- `be` : big endian
- `ret` : target endianness

Regexes for git grep

- `\<helper_\(le\|be\|ret\)_ld[us]\?[bwlq]_mmu\>`
- `\<helper_\(le\|be\|ret\)_st[bwlq]_mmu\>`

6.3.8 `address_space_*`

These functions are the primary ones to use when emulating CPU or device memory accesses. They take an AddressSpace, which is the way QEMU defines the view of memory that a device or CPU has. (They generally correspond to being the "master" end of a hardware bus or bus fabric.)

Each CPU has an AddressSpace. Some kinds of CPU have more than one AddressSpace (for instance Arm guest CPUs have an AddressSpace for the Secure world and one for NonSecure if they implement TrustZone). Devices which can do DMA-type operations should generally have an AddressSpace. There is also a "system address space" which typically has all the devices and memory that all CPUs can see. (Some older device models use the "system address space" rather than properly modelling that they have an AddressSpace of their own.)

Functions are provided for doing byte-buffer reads and writes, and also for doing one-data-item loads and stores.

In all cases the caller provides a MemTxAttrs to specify bus transaction attributes, and can check whether the memory transaction succeeded using a MemTxResult return code.

`address_space_read(address_space, addr, attrs, buf, len)`

`address_space_write(address_space, addr, attrs, buf, len)`

`address_space_rw(address_space, addr, attrs, buf, len, is_write)`

`address_space_ld{sign}{size}_{endian}(address_space, addr, attrs, txresult)`

`address_space_st{size}_{endian}(address_space, addr, val, attrs, txresult)`

`sign`

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned

(No signed load operations are provided.)

`size`

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits
- `q` : 64 bits

`endian`

- `le` : little endian
- `be` : big endian

The `_{endian}` suffix is omitted for byte accesses.

Regexes for git grep

- `\<address_space_\(read\|write\|rw\)\>`
- `\<address_space_ldu\?[bwql]\(_[lb]e\)\?\>`
- `\<address_space_st[bwql]\(_[lb]e\)\?\>`

6.3.9 `address_space_write_rom`

This function performs a write by physical address like `address_space_write`, except that if the write is to a ROM then the ROM contents will be modified, even though a write by the guest CPU to the ROM would be ignored. This is used for non-guest writes like writes from the gdb debug stub or initial loading of ROM contents.

Note that portions of the write which attempt to write data to a device will be silently ignored – only real RAM and ROM will be written to.

Regexes for git grep

- `address_space_write_rom`

6.3.10 `{ld,st}*_phys`

These are functions which are identical to `address_space_{ld,st}*`, except that they always pass `MEMTXATTRS_UNSPECIFIED` for the transaction attributes, and ignore whether the transaction succeeded or failed.

The fact that they ignore whether the transaction succeeded means they should not be used in new code, unless you know for certain that your code will only be used in a context where the CPU or device doing the access has no way to report such an error.

`load:  ld{sign}{size}_{endian}_phys`

`store:  st{size}_{endian}_phys`

`sign`

- (empty) : for 32 or 64 bit sizes
- `u` : unsigned

(No signed load operations are provided.)

`size`

- `b` : 8 bits
- `w` : 16 bits
- `l` : 32 bits

- `q` : 64 bits

`endian`

- `le` : little endian
- `be` : big endian

The `_{endian}_` infix is omitted for byte accesses.

Regexes for git grep

- `\<ldu\?[bwlq]\(_[bl]e\)\?_phys\>`
- `\<st[bwlq]\(_[bl]e\)\?_phys\>`

6.3.11 `cpu_physical_memory_*`

These are convenience functions which are identical to `address_space_*` but operate specifically on the system address space, always pass a `MEMTXATTRS_UNSPECIFIED` set of memory attributes and ignore whether the memory transaction succeeded or failed. For new code they are better avoided:

- there is likely to be behaviour you need to model correctly for a failed read or write operation
- a device should usually perform operations on its own AddressSpace rather than using the system address space

`cpu_physical_memory_read`

`cpu_physical_memory_write`

`cpu_physical_memory_rw`

Regexes for git grep

- `\<cpu_physical_memory_\(read\|write\|rw\)\>`

6.3.12 `cpu_memory_rw_debug`

Access CPU memory by virtual address for debug purposes.

This function is intended for use by the GDB stub and similar code. It takes a virtual address, converts it to a physical address via an MMU lookup using the current settings of the specified CPU, and then performs the access (using `address_space_rw` for reads or `cpu_physical_memory_write_rom` for writes). This means that if the access is a write to a ROM then this function will modify the contents (whereas a normal guest CPU access would ignore the write attempt).

`cpu_memory_rw_debug`

6.3.13 `dma_memory_*`

These behave like `address_space_*`, except that they perform a DMA barrier operation first.

TODO: We should provide guidance on when you need the DMA barrier operation and when it's OK to use `address_space_*`, and make sure our existing code is doing things correctly.

`dma_memory_read`

`dma_memory_write`

`dma_memory_rw`

Regexes for git grep

  - `\<dma_memory_\(read\|write\|rw\)\>`

6.3.14 `pci_dma_*` and `{ld,st}*_pci_dma`

These functions are specifically for PCI device models which need to perform accesses where the PCI device is a bus master. You pass them a `PCIDevice *` and they will do `dma_memory_*` operations on the correct address space for that device.

`pci_dma_read`

`pci_dma_write`

`pci_dma_rw`

`load:  ld{sign}{size}_{endian}_pci_dma`

`store:  st{size}_{endian}_pci_dma`

`sign`

  - (empty) : for 32 or 64 bit sizes
  - `u` : unsigned

(No signed load operations are provided.)

`size`

  - `b` : 8 bits
  - `w` : 16 bits
  - `l` : 32 bits
  - `q` : 64 bits

`endian`

  - `le` : little endian
  - `be` : big endian

The `_{endian}_` infix is omitted for byte accesses.

Regexes for git grep

  - `\<pci_dma_\(read\|write\|rw\)\>`
  - `\<ldu\?[bwlq]\(_[bl]e\)\?_pci_dma\>`
  - `\<st[bwlq]\(_[bl]e\)\?_pci_dma\>`

6.4 The memory API

The memory API models the memory and I/O buses and controllers of a QEMU machine. It attempts to allow modelling of:

- ordinary RAM
- memory-mapped I/O (MMIO)
- memory controllers that can dynamically reroute physical memory regions to different destinations

The memory model provides support for

- tracking RAM changes by the guest
- setting up coalesced memory for kvm
- setting up ioeventfd regions for kvm

Memory is modelled as an acyclic graph of MemoryRegion objects. Sinks (leaves) are RAM and MMIO regions, while other nodes represent buses, memory controllers, and memory regions that have been rerouted.

In addition to MemoryRegion objects, the memory API provides AddressSpace objects for every root and possibly for intermediate MemoryRegions too. These represent memory as seen from the CPU or a device's viewpoint.

6.4.1 Types of regions

There are multiple types of memory regions (all represented by a single C type MemoryRegion):

- RAM: a RAM region is simply a range of host memory that can be made available to the guest. You typically initialize these with memory_region_init_ram(). Some special purposes require the variants memory_region_init_resizeable_ram(), memory_region_init_ram_from_file(), or memory_region_init_ram_ptr().
- MMIO: a range of guest memory that is implemented by host callbacks; each read or write causes a callback to be called on the host. You initialize these with memory_region_init_io(), passing it a MemoryRegionOps structure describing the callbacks.
- ROM: a ROM memory region works like RAM for reads (directly accessing a region of host memory), and forbids writes. You initialize these with memory_region_init_rom().
- ROM device: a ROM device memory region works like RAM for reads (directly accessing a region of host memory), but like MMIO for writes (invoking a callback). You initialize these with memory_region_init_rom_device().
- IOMMU region: an IOMMU region translates addresses of accesses made to it and forwards them to some other target memory region. As the name suggests, these are only needed for modelling an IOMMU, not for simple devices. You initialize these with memory_region_init_iommu().
- container: a container simply includes other memory regions, each at a different offset. Containers are useful for grouping several regions into one unit. For example, a PCI BAR may be composed of a RAM region and an MMIO region.

  A container's subregions are usually non-overlapping. In some cases it is useful to have overlapping regions; for example a memory controller that can overlay a subregion of RAM with MMIO or ROM, or a PCI controller that does not prevent card from claiming overlapping BARs.

  You initialize a pure container with memory_region_init().
- alias: a subsection of another region. Aliases allow a region to be split apart into discontiguous regions. Examples of uses are memory banks used when the guest address space is smaller than the amount of RAM addressed, or a memory controller that splits main memory to expose a "PCI hole". Aliases may point to any type of region, including other aliases, but an alias may not point back to itself, directly or indirectly. You initialize these with memory_region_init_alias().
- reservation region: a reservation region is primarily for debugging. It claims I/O space that is not supposed to be handled by QEMU itself. The typical use is to track parts of the address space which will be handled by the host kernel when KVM is enabled. You initialize these by passing a NULL callback parameter to memory_region_init_io().

It is valid to add subregions to a region which is not a pure container (that is, to an MMIO, RAM or ROM region). This means that the region will act like a container, except that any addresses within the container's region which are not claimed by any subregion are handled by the container itself (ie by its MMIO callbacks or RAM backing). However it is generally possible to achieve the same effect with a pure container one of whose subregions is a low priority "background" region covering the whole address range; this is often clearer and is preferred. Subregions cannot be added to an alias region.

6.4.2 Migration

Where the memory region is backed by host memory (RAM, ROM and ROM device memory region types), this host memory needs to be copied to the destination on migration. These APIs which allocate the host memory for you will also register the memory so it is migrated:

- memory_region_init_ram()
- memory_region_init_rom()
- memory_region_init_rom_device()

For most devices and boards this is the correct thing. If you have a special case where you need to manage the migration of the backing memory yourself, you can call the functions:

- memory_region_init_ram_nomigrate()
- memory_region_init_rom_nomigrate()
- memory_region_init_rom_device_nomigrate()

which only initialize the MemoryRegion and leave handling migration to the caller.

The functions:

- memory_region_init_resizeable_ram()
- memory_region_init_ram_from_file()
- memory_region_init_ram_from_fd()
- memory_region_init_ram_ptr()
- memory_region_init_ram_device_ptr()

are for special cases only, and so they do not automatically register the backing memory for migration; the caller must manage migration if necessary.

6.4.3 Region names

Regions are assigned names by the constructor. For most regions these are only used for debugging purposes, but RAM regions also use the name to identify live migration sections. This means that RAM region names need to have ABI stability.

6.4.4 Region lifecycle

A region is created by one of the memory_region_init*() functions and attached to an object, which acts as its owner or parent. QEMU ensures that the owner object remains alive as long as the region is visible to the guest, or as long as the region is in use by a virtual CPU or another device. For example, the owner object will not die between an address_space_map operation and the corresponding address_space_unmap.

After creation, a region can be added to an address space or a container with memory_region_add_subregion(), and removed using memory_region_del_subregion().

Various region attributes (read-only, dirty logging, coalesced mmio, ioeventfd) can be changed during the region lifecycle. They take effect as soon as the region is made visible. This can be immediately, later, or never.

Destruction of a memory region happens automatically when the owner object dies.

If however the memory region is part of a dynamically allocated data structure, you should call object_unparent() to destroy the memory region before the data structure is freed. For an example see VFIOMSIXInfo and VFIOQuirk in hw/vfio/pci.c.

You must not destroy a memory region as long as it may be in use by a device or CPU. In order to do this, as a general rule do not create or destroy memory regions dynamically during a device's lifetime, and only call object_unparent() in the memory region owner's instance_finalize callback. The dynamically allocated data structure that contains the memory region then should obviously be freed in the instance_finalize callback as well.

If you break this rule, the following situation can happen:

- the memory region's owner had a reference taken via memory_region_ref (for example by address_space_map)
- the region is unparented, and has no owner anymore
- when address_space_unmap is called, the reference to the memory region's owner is leaked.

There is an exception to the above rule: it is okay to call object_unparent at any time for an alias or a container region. It is therefore also okay to create or destroy alias and container regions dynamically during a device's lifetime.

This exceptional usage is valid because aliases and containers only help QEMU building the guest's memory map; they are never accessed directly. memory_region_ref and memory_region_unref are never called on aliases or containers, and the above situation then cannot happen. Exploiting this exception is rarely necessary, and therefore it is discouraged, but nevertheless it is used in a few places.

For regions that "have no owner" (NULL is passed at creation time), the machine object is actually used as the owner. Since instance_finalize is never called for the machine object, you must never call object_unparent on regions that have no owner, unless they are aliases or containers.

6.4.5 Overlapping regions and priority

Usually, regions may not overlap each other; a memory address decodes into exactly one target. In some cases it is useful to allow regions to overlap, and sometimes to control which of an overlapping regions is visible to the guest. This is done with memory_region_add_subregion_overlap(), which allows the region to overlap any other region in the same container, and specifies a priority that allows the core to decide which of two regions at the same address are visible (highest wins). Priority values are signed, and the default value is zero. This means that you can use memory_region_add_subregion_overlap() both to specify a region that must sit 'above' any others (with a positive priority) and also a background region that sits 'below' others (with a negative priority).

If the higher priority region in an overlap is a container or alias, then the lower priority region will appear in any "holes" that the higher priority region has left by not mapping subregions to that area of its address range. (This applies recursively – if the subregions are themselves containers or aliases that leave holes then the lower priority region will appear in these holes too.)

For example, suppose we have a container A of size 0x8000 with two subregions B and C. B is a container mapped at 0x2000, size 0x4000, priority 2; C is an MMIO region mapped at 0x0, size 0x6000, priority 1. B currently has two of its own subregions: D of size 0x1000 at offset 0 and E of size 0x1000 at offset 0x2000. As a diagram:

```
      0      1000   2000   3000   4000   5000   6000   7000   8000
      |------|------|------|------|------|------|------|------|
A:    [                                                       ]
C:    [CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC]
B:                  [                           ]
D:                  [DDDDD]
E:                                [EEEEE]
```

The regions that will be seen within this address range then are:

```
[CCCCCCCCCCCC][DDDDD][CCCCC][EEEEE][CCCCC]
```

Since B has higher priority than C, its subregions appear in the flat map even where they overlap with C. In ranges where B has not mapped anything C's region appears.

If B had provided its own MMIO operations (ie it was not a pure container) then these would be used for any addresses in its range not handled by D or E, and the result would be:

```
[CCCCCCCCCCCC][DDDDD][BBBBB][EEEEE][BBBBB]
```

Priority values are local to a container, because the priorities of two regions are only compared when they are both children of the same container. This means that the device in charge of the container (typically modelling a bus or a memory controller) can use them to manage the interaction of its child regions without any side effects on other parts of the system. In the example above, the priorities of D and E are unimportant because they do not overlap each other. It is the relative priority of B and C that causes D and E to appear on top of C: D and E's priorities are never compared against the priority of C.

6.4.6 Visibility

The memory core uses the following rules to select a memory region when the guest accesses an address:

- all direct subregions of the root region are matched against the address, in descending priority order
  - if the address lies outside the region offset/size, the subregion is discarded
  - if the subregion is a leaf (RAM or MMIO), the search terminates, returning this leaf region
  - if the subregion is a container, the same algorithm is used within the subregion (after the address is adjusted by the subregion offset)
  - if the subregion is an alias, the search is continued at the alias target (after the address is adjusted by the subregion offset and alias offset)
  - if a recursive search within a container or alias subregion does not find a match (because of a "hole" in the container's coverage of its address range), then if this is a container with its own MMIO or RAM backing the search terminates, returning the container itself. Otherwise we continue with the next subregion in priority order
- if none of the subregions match the address then the search terminates with no match found

6.4.7 Example memory map

```
system_memory: container@0-2^48-1
 |
 +---- lomem: alias@0-0xdfffffff ---> #ram (0-0xdfffffff)
 |
 +---- himem: alias@0x100000000-0x11fffffff ---> #ram (0xe0000000-0xffffffff)
 |
 +---- vga-window: alias@0xa0000-0xbffff ---> #pci (0xa0000-0xbffff)
 |      (prio 1)
 |
 +---- pci-hole: alias@0xe0000000-0xffffffff ---> #pci (0xe0000000-0xffffffff)

pci (0-2^32-1)
 |
 +--- vga-area: container@0xa0000-0xbffff
```

(continues on next page)

(continued from previous page)

```
  |      |
  |      +--- alias@0x00000-0x7fff  ---> #vram (0x010000-0x017fff)
  |      |
  |      +--- alias@0x08000-0xffff  ---> #vram (0x020000-0x027fff)
  |
  +---- vram: ram@0xe1000000-0xe1ffffff
  |
  +---- vga-mmio: mmio@0xe2000000-0xe200ffff

ram: ram@0x00000000-0xffffffff
```

This is a (simplified) PC memory map. The 4GB RAM block is mapped into the system address space via two aliases: “lomem” is a 1:1 mapping of the first 3.5GB; “himem” maps the last 0.5GB at address 4GB. This leaves 0.5GB for the so-called PCI hole, that allows a 32-bit PCI bus to exist in a system with 4GB of memory.

The memory controller diverts addresses in the range 640K-768K to the PCI address space. This is modelled using the “vga-window” alias, mapped at a higher priority so it obscures the RAM at the same addresses. The vga window can be removed by programming the memory controller; this is modelled by removing the alias and exposing the RAM underneath.

The pci address space is not a direct child of the system address space, since we only want parts of it to be visible (we accomplish this using aliases). It has two subregions: vga-area models the legacy vga window and is occupied by two 32K memory banks pointing at two sections of the framebuffer. In addition the vram is mapped as a BAR at address e1000000, and an additional BAR containing MMIO registers is mapped after it.

Note that if the guest maps a BAR outside the PCI hole, it would not be visible as the pci-hole alias clips it to a 0.5GB range.

6.4.8 MMIO Operations

MMIO regions are provided with ->read() and ->write() callbacks, which are sufficient for most devices. Some devices change behaviour based on the attributes used for the memory transaction, or need to be able to respond that the access should provoke a bus error rather than completing successfully; those devices can use the ->read_with_attrs() and ->write_with_attrs() callbacks instead.

In addition various constraints can be supplied to control how these callbacks are called:

- .valid.min_access_size, .valid.max_access_size define the access sizes (in bytes) which the device accepts; accesses outside this range will have device and bus specific behaviour (ignored, or machine check)
- .valid.unaligned specifies that the *device being modelled* supports unaligned accesses; if false, unaligned accesses will invoke the appropriate bus or CPU specific behaviour.
- .impl.min_access_size, .impl.max_access_size define the access sizes (in bytes) supported by the *implementation*; other access sizes will be emulated using the ones available. For example a 4-byte write will be emulated using four 1-byte writes, if .impl.max_access_size = 1.
- .impl.unaligned specifies that the *implementation* supports unaligned accesses; if false, unaligned accesses will be emulated by two aligned accesses.

6.4.9 API Reference

struct **MemoryListener**
  callbacks structure for updates to the physical memory map

Definition

```
struct MemoryListener {
  void (*begin)(MemoryListener *listener);
  void (*commit)(MemoryListener *listener);
  void (*region_add)(MemoryListener *listener, MemoryRegionSection *section);
  void (*region_del)(MemoryListener *listener, MemoryRegionSection *section);
  void (*region_nop)(MemoryListener *listener, MemoryRegionSection *section);
  void (*log_start)(MemoryListener *listener, MemoryRegionSection *section, int old,␣
↪int new);
  void (*log_stop)(MemoryListener *listener, MemoryRegionSection *section, int old,␣
↪int new);
  void (*log_sync)(MemoryListener *listener, MemoryRegionSection *section);
  void (*log_clear)(MemoryListener *listener, MemoryRegionSection *section);
  void (*log_global_start)(MemoryListener *listener);
  void (*log_global_stop)(MemoryListener *listener);
  void (*log_global_after_sync)(MemoryListener *listener);
  void (*eventfd_add)(MemoryListener *listener, MemoryRegionSection *section, bool␣
↪match_data, uint64_t data, EventNotifier *e);
  void (*eventfd_del)(MemoryListener *listener, MemoryRegionSection *section, bool␣
↪match_data, uint64_t data, EventNotifier *e);
  void (*coalesced_io_add)(MemoryListener *listener, MemoryRegionSection *section,␣
↪hwaddr addr, hwaddr len);
  void (*coalesced_io_del)(MemoryListener *listener, MemoryRegionSection *section,␣
↪hwaddr addr, hwaddr len);
  unsigned priority;
};
```

Members

`begin` Called at the beginning of an address space update transaction. Followed by calls to *`MemoryListener.region_add()`*, *`MemoryListener.region_del()`*, *`MemoryListener.region_nop()`*, *`MemoryListener.log_start()`* and *`MemoryListener.log_stop()`* in increasing address order.

listener: The *`MemoryListener`*.

`commit` Called at the end of an address space update transaction, after the last call to *`MemoryListener.region_add()`*, *`MemoryListener.region_del()`* or *`MemoryListener.region_nop()`*, *`MemoryListener.log_start()`* and *`MemoryListener.log_stop()`*.

listener: The *`MemoryListener`*.

`region_add` Called during an address space update transaction, for a section of the address space that is new in this address space space since the last transaction.

listener: The *`MemoryListener`*. **section**: The new *`MemoryRegionSection`*.

`region_del` Called during an address space update transaction, for a section of the address space that has disappeared in the address space since the last transaction.

listener: The *`MemoryListener`*. **section**: The old *`MemoryRegionSection`*.

`region_nop` Called during an address space update transaction, for a section of the address space that is in the same place in the address space as in the last transaction.

listener: The *`MemoryListener`*. **section**: The *`MemoryRegionSection`*.

`log_start` Called during an address space update transaction, after one of *`MemoryListener.region_add()`*,:c:type:*MemoryListener.region_del() <MemoryListener>* or *`MemoryListener.region_nop()`*, if dirty memory logging clients have become active since the last transaction.

listener: The *`MemoryListener`*. **section**: The *`MemoryRegionSection`*. **old**: A bitmap of dirty memory logging clients that were active in the previous transaction. **new**: A bitmap of dirty memory logging clients that are active in the current transaction.

`log_stop` Called during an address space update transaction, after one of *`MemoryListener.region_add()`*, *`MemoryListener.region_del()`* or *`MemoryListener.region_nop()`* and possibly after *`MemoryListener.log_start()`*, if dirty memory logging clients have become inactive since the last transaction.

listener: The *`MemoryListener`*. **section**: The *`MemoryRegionSection`*. **old**: A bitmap of dirty memory logging clients that were active in the previous transaction. **new**: A bitmap of dirty memory logging clients that are active in the current transaction.

`log_sync` Called by memory_region_snapshot_and_clear_dirty() and memory_global_dirty_log_sync(), before accessing QEMU's "official" copy of the dirty memory bitmap for a *`MemoryRegionSection`*.

listener: The *`MemoryListener`*. **section**: The *`MemoryRegionSection`*.

`log_clear` Called before reading the dirty memory bitmap for a *`MemoryRegionSection`*.

listener: The *`MemoryListener`*. **section**: The *`MemoryRegionSection`*.

`log_global_start` Called by memory_global_dirty_log_start(), which enables the `DIRTY_LOG_MIGRATION` client on all memory regions in the address space. *`MemoryListener.log_global_start()`* is also called when a *`MemoryListener`* is added, if global dirty logging is active at that time.

listener: The *`MemoryListener`*.

`log_global_stop` Called by memory_global_dirty_log_stop(), which disables the `DIRTY_LOG_MIGRATION` client on all memory regions in the address space.

listener: The *`MemoryListener`*.

`log_global_after_sync` Called after reading the dirty memory bitmap for any *`MemoryRegionSection`*.

listener: The *`MemoryListener`*.

`eventfd_add` Called during an address space update transaction, for a section of the address space that has had a new ioeventfd registration since the last transaction.

listener: The *`MemoryListener`*. **section**: The new *`MemoryRegionSection`*. **match_data**: The **match_data** parameter for the new ioeventfd. **data**: The **data** parameter for the new ioeventfd. **e**: The `EventNotifier` parameter for the new ioeventfd.

`eventfd_del` Called during an address space update transaction, for a section of the address space that has dropped an ioeventfd registration since the last transaction.

listener: The *`MemoryListener`*. **section**: The new *`MemoryRegionSection`*. **match_data**: The **match_data** parameter for the dropped ioeventfd. **data**: The **data** parameter for the dropped ioeventfd. **e**: The `EventNotifier` parameter for the dropped ioeventfd.

`coalesced_io_add` Called during an address space update transaction, for a section of the address space that has had a new coalesced MMIO range registration since the last transaction.

listener: The *`MemoryListener`*. **section**: The new *`MemoryRegionSection`*. **addr**: The starting address for the coalesced MMIO range. **len**: The length of the coalesced MMIO range.

`coalesced_io_del` Called during an address space update transaction, for a section of the address space that has dropped a coalesced MMIO range since the last transaction.

listener: The *`MemoryListener`*. **section**: The new *`MemoryRegionSection`*. **addr**: The starting address for the coalesced MMIO range. **len**: The length of the coalesced MMIO range.

`priority` Govern the order in which memory listeners are invoked. Lower priorities are invoked earlier for "add" or "start" callbacks, and later for "delete" or "stop" callbacks.

Description

Allows a component to adjust to changes in the guest-visible memory map. Use with memory_listener_register() and memory_listener_unregister().

struct **AddressSpace**
describes a mapping of addresses to `MemoryRegion` objects

Definition

```
struct AddressSpace {
};
```

Members

struct **MemoryRegionSection**
describes a fragment of a `MemoryRegion`

Definition

```
struct MemoryRegionSection {
  Int128 size;
  MemoryRegion *mr;
  FlatView *fv;
  hwaddr offset_within_region;
  hwaddr offset_within_address_space;
  bool readonly;
  bool nonvolatile;
};
```

Members

size the size of the section; will not exceed **mr**'s boundaries

mr the region, or `NULL` if empty

fv the flat view of the address space the region is mapped in

offset_within_region the beginning of the section, relative to **mr**'s start

offset_within_address_space the address of the first byte of the section relative to the region's address space

readonly writes to this section are ignored

nonvolatile this section is non-volatile

void **memory_region_init** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*)
Initialize a memory region

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized

struct Object * owner the object that tracks the region's reference count

const char * name used for debugging; not visible to the user or ABI

uint64_t size size of the region; any subregions beyond this size will be clipped

Description

The region typically acts as a container for other memory regions. Use memory_region_add_subregion() to add subregions.

void **memory_region_ref** (MemoryRegion * *mr*)
Add 1 to a memory region's reference count

Parameters

MemoryRegion * mr the `MemoryRegion`

Description

Whenever memory regions are accessed outside the BQL, they need to be preserved against hot-unplug. MemoryRegions actually do not have their own reference count; they piggyback on a QOM object, their "owner". This function adds a reference to the owner.

All MemoryRegions must have an owner if they can disappear, even if the device they belong to operates exclusively under the BQL. This is because the region could be returned at any time by memory_region_find, and this is usually under guest control.

void **memory_region_unref** (MemoryRegion * *mr*)
Remove 1 to a memory region's reference count

Parameters

MemoryRegion * mr the `MemoryRegion`

Description

Whenever memory regions are accessed outside the BQL, they need to be preserved against hot-unplug. MemoryRegions actually do not have their own reference count; they piggyback on a QOM object, their "owner". This function removes a reference to the owner and possibly destroys it.

void **memory_region_init_io** (MemoryRegion * *mr*, struct Object * *owner*, const MemoryRegionOps * *ops*, void * *opaque*, const char * *name*, uint64_t *size*)
Initialize an I/O memory region.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const MemoryRegionOps * ops a structure containing read and write callbacks to be used when I/O is performed on the region.

void * opaque passed to the read and write callbacks of the **ops** structure.

const char * name used for debugging; not visible to the user or ABI

uint64_t size size of the region.

Description

Accesses into the region will cause the callbacks in **ops** to be called. if **size** is nonzero, subregions will be clipped to **size**.

void **memory_region_init_ram_nomigrate** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, Error ** *errp*)
Initialize RAM memory region. Accesses into the region will modify memory directly.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const char * name Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size size of the region.

Error ** errp pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

void **memory_region_init_ram_shared_nomigrate** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, bool *share*, Error ** *errp*)

Initialize RAM memory region. Accesses into the region will modify memory directly.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const char * name Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size size of the region.

bool share allow remapping RAM to different addresses

Error ** errp pointer to Error*, to store an error if it happens.

Description

Note that this function is similar to memory_region_init_ram_nomigrate. The only difference is part of the RAM region can be remapped.

void **memory_region_init_resizeable_ram** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, uint64_t *max_size*, void (**resized*)(const char*, uint64_t length, void *host), Error ** *errp*)

Initialize memory region with resizeable RAM. Accesses into the region will modify memory directly. Only an initial portion of this RAM is actually used. The used size can change across reboots.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const char * name Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size used size of the region.

uint64_t max_size max size of the region.

void (*)(const char*, uint64_t length, void *host) resized callback to notify owner about used size change.

Error ** errp pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

void **memory_region_init_ram_from_file** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, uint64_t *align*, uint32_t *ram_flags*, const char * *path*, Error ** *errp*)

Initialize RAM memory region with a mmap-ed backend.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const char * name Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size size of the region.

uint64_t align alignment of the region base address; if 0, the default alignment (getpagesize()) will be used.

uint32_t ram_flags Memory region features: - RAM_SHARED: memory must be mmaped with the MAP_SHARED flag - RAM_PMEM: the memory is persistent memory Other bits are ignored now.

const char * path the path in which to allocate the RAM.

Error ** errp pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

void **memory_region_init_ram_from_fd** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, bool *share*, int *fd*, Error ** *errp*)

Initialize RAM memory region with a mmap-ed backend.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const char * name the name of the region.

uint64_t size size of the region.

bool share `true` if memory must be mmaped with the MAP_SHARED flag

int fd the fd to mmap.

Error ** errp pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

void **memory_region_init_ram_ptr** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, void * *ptr*)

Initialize RAM memory region from a user-provided pointer. Accesses into the region will modify memory directly.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const char * name Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

`uint64_t size` size of the region.

`void * ptr` memory to be mapped; must contain at least **size** bytes.

Description

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller.

void **`memory_region_init_ram_device_ptr`** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, void * *ptr*)

Initialize RAM device memory region from a user-provided pointer.

Parameters

`MemoryRegion * mr` the `MemoryRegion` to be initialized.

`struct Object * owner` the object that tracks the region's reference count

`const char * name` the name of the region.

`uint64_t size` size of the region.

`void * ptr` memory to be mapped; must contain at least **size** bytes.

Description

A RAM device represents a mapping to a physical device, such as to a PCI MMIO BAR of an vfio-pci assigned device. The memory region may be mapped into the VM address space and access to the region will modify memory directly. However, the memory region should not be included in a memory dump (device may not be enabled/mapped at the time of the dump), and operations incompatible with manipulating MMIO should be avoided. Replaces skip_dump flag.

Note that this function does not do anything to cause the data in the RAM memory region to be migrated; that is the responsibility of the caller. (For RAM device memory regions, migrating the contents rarely makes sense.)

void **`memory_region_init_alias`** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, MemoryRegion * *orig*, hwaddr *offset*, uint64_t *size*)

Initialize a memory region that aliases all or a part of another memory region.

Parameters

`MemoryRegion * mr` the `MemoryRegion` to be initialized.

`struct Object * owner` the object that tracks the region's reference count

`const char * name` used for debugging; not visible to the user or ABI

`MemoryRegion * orig` the region to be referenced; **mr** will be equivalent to **orig** between **offset** and **offset** + **size** - 1.

`hwaddr offset` start of the section in **orig** to be referenced.

`uint64_t size` size of the region.

void **`memory_region_init_rom_nomigrate`** (MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, Error ** *errp*)

Initialize a ROM memory region.

Parameters

`MemoryRegion * mr` the `MemoryRegion` to be initialized.

`struct Object * owner` the object that tracks the region's reference count

`const char * name` Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size size of the region.

Error ** errp pointer to Error*, to store an error if it happens.

Description

This has the same effect as calling memory_region_init_ram_nomigrate() and then marking the resulting region read-only with memory_region_set_readonly().

Note that this function does not do anything to cause the data in the RAM side of the memory region to be migrated; that is the responsibility of the caller.

void **memory_region_init_rom_device_nomigrate** (MemoryRegion * *mr*, struct Object * *owner*, const MemoryRegionOps * *ops*, void * *opaque*, const char * *name*, uint64_t *size*, Error ** *errp*)

Initialize a ROM memory region. Writes are handled via callbacks.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const MemoryRegionOps * ops callbacks for write access handling (must not be NULL).

void * opaque passed to the read and write callbacks of the **ops** structure.

const char * name Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size size of the region.

Error ** errp pointer to Error*, to store an error if it happens.

Description

Note that this function does not do anything to cause the data in the RAM side of the memory region to be migrated; that is the responsibility of the caller.

void **memory_region_init_iommu** (void * *_iommu_mr*, size_t *instance_size*, const char * *mrtypename*, Object * *owner*, const char * *name*, uint64_t *size*)

Initialize a memory region of a custom type that translates addresses

Parameters

void * _iommu_mr the `IOMMUMemoryRegion` to be initialized

size_t instance_size the IOMMUMemoryRegion subclass instance size

const char * mrtypename the type name of the `IOMMUMemoryRegion`

Object * owner the object that tracks the region's reference count

const char * name used for debugging; not visible to the user or ABI

uint64_t size size of the region.

Description

An IOMMU region translates addresses and forwards accesses to a target memory region.

The IOMMU implementation must define a subclass of TYPE_IOMMU_MEMORY_REGION. **_iommu_mr** should be a pointer to enough memory for an instance of that subclass, **instance_size** is the size of that subclass, and **mrtypename** is its name. This function will initialize **_iommu_mr** as an instance of the subclass, and its methods will then be called to handle accesses to the memory region. See the documentation of `IOMMUMemoryRegionClass` for further details.

void **memory_region_init_ram**(MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, Error ** *errp*)

Initialize RAM memory region. Accesses into the region will modify memory directly.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized

struct Object * owner the object that tracks the region's reference count (must be TYPE_DEVICE or a subclass of TYPE_DEVICE, or NULL)

const char * name name of the memory region

uint64_t size size of the region in bytes

Error ** errp pointer to Error*, to store an error if it happens.

Description

This function allocates RAM for a board model or device, and arranges for it to be migrated (by calling vmstate_register_ram() if **owner** is a DeviceState, or vmstate_register_ram_global() if **owner** is NULL).

TODO: Currently we restrict **owner** to being either NULL (for global RAM regions with no owner) or devices, so that we can give the RAM block a unique name for migration purposes. We should lift this restriction and allow arbitrary Objects. If you pass a non-NULL non-device **owner** then we will assert.

void **memory_region_init_rom**(MemoryRegion * *mr*, struct Object * *owner*, const char * *name*, uint64_t *size*, Error ** *errp*)

Initialize a ROM memory region.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const char * name Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size size of the region.

Error ** errp pointer to Error*, to store an error if it happens.

Description

This has the same effect as calling memory_region_init_ram() and then marking the resulting region read-only with memory_region_set_readonly(). This includes arranging for the contents to be migrated.

TODO: Currently we restrict **owner** to being either NULL (for global RAM regions with no owner) or devices, so that we can give the RAM block a unique name for migration purposes. We should lift this restriction and allow arbitrary Objects. If you pass a non-NULL non-device **owner** then we will assert.

void **memory_region_init_rom_device**(MemoryRegion * *mr*, struct Object * *owner*, const MemoryRegionOps * *ops*, void * *opaque*, const char * *name*, uint64_t *size*, Error ** *errp*)

Initialize a ROM memory region. Writes are handled via callbacks.

Parameters

MemoryRegion * mr the `MemoryRegion` to be initialized.

struct Object * owner the object that tracks the region's reference count

const MemoryRegionOps * ops callbacks for write access handling (must not be NULL).

void * opaque passed to the read and write callbacks of the **ops** structure.

const char * name Region name, becomes part of RAMBlock name used in migration stream must be unique within any device

uint64_t size size of the region.

Error ** errp pointer to Error*, to store an error if it happens.

Description

This function initializes a memory region backed by RAM for reads and callbacks for writes, and arranges for the RAM backing to be migrated (by calling vmstate_register_ram() if **owner** is a DeviceState, or vmstate_register_ram_global() if **owner** is NULL).

TODO: Currently we restrict **owner** to being either NULL (for global RAM regions with no owner) or devices, so that we can give the RAM block a unique name for migration purposes. We should lift this restriction and allow arbitrary Objects. If you pass a non-NULL non-device **owner** then we will assert.

struct Object * **memory_region_owner**(MemoryRegion * *mr*)
get a memory region's owner.

Parameters

MemoryRegion * mr the memory region being queried.

uint64_t **memory_region_size**(MemoryRegion * *mr*)
get a memory region's size.

Parameters

MemoryRegion * mr the memory region being queried.

bool **memory_region_is_ram**(MemoryRegion * *mr*)
check whether a memory region is random access

Parameters

MemoryRegion * mr the memory region being queried

Description

Returns `true` if a memory region is random access.

bool **memory_region_is_ram_device**(MemoryRegion * *mr*)
check whether a memory region is a ram device

Parameters

MemoryRegion * mr the memory region being queried

Description

Returns `true` if a memory region is a device backed ram region

bool **memory_region_is_romd**(MemoryRegion * *mr*)
check whether a memory region is in ROMD mode

Parameters

MemoryRegion * mr the memory region being queried

Description

Returns `true` if a memory region is a ROM device and currently set to allow direct reads.

IOMMUMemoryRegion * **memory_region_get_iommu**(MemoryRegion * *mr*)
check whether a memory region is an iommu

Parameters

MemoryRegion * mr the memory region being queried

Description

Returns pointer to IOMMUMemoryRegion if a memory region is an iommu, otherwise NULL.

IOMMUMemoryRegionClass * **memory_region_get_iommu_class_nocheck** (IOMMUMemoryRegion * *iommu_mr*)
returns iommu memory region class if an iommu or NULL if not

Parameters

IOMMUMemoryRegion * iommu_mr the memory region being queried

Description

Returns pointer to IOMMUMemoryRegionClass if a memory region is an iommu, otherwise NULL. This is fast path avoiding QOM checking, use with caution.

uint64_t **memory_region_iommu_get_min_page_size** (IOMMUMemoryRegion * *iommu_mr*)
get minimum supported page size for an iommu

Parameters

IOMMUMemoryRegion * iommu_mr the memory region being queried

Description

Returns minimum supported page size for an iommu.

void **memory_region_notify_iommu** (IOMMUMemoryRegion * *iommu_mr*, int *iommu_idx*, IOMMUTLBEntry *entry*)
notify a change in an IOMMU translation entry.

Parameters

IOMMUMemoryRegion * iommu_mr the memory region that was changed

int iommu_idx the IOMMU index for the translation table which has changed

IOMMUTLBEntry entry the new entry in the IOMMU translation table. The entry replaces all old entries for the same virtual I/O address range. Deleted entries have .**perm** == 0.

Description

The notification type will be decided by entry.perm bits:

- For UNMAP (cache invalidation) notifies: set entry.perm to IOMMU_NONE.
- For MAP (newly added entry) notifies: set entry.perm to the permission of the page (which is definitely !IOMMU_NONE).

Note

for any IOMMU implementation, an in-place mapping change should be notified with an UNMAP followed by a MAP.

void **memory_region_notify_one** (IOMMUNotifier * *notifier*, IOMMUTLBEntry * *entry*)
notify a change in an IOMMU translation entry to a single notifier

Parameters

IOMMUNotifier * notifier the notifier to be notified

IOMMUTLBEntry * entry the new entry in the IOMMU translation table. The entry replaces all old entries for the same virtual I/O address range. Deleted entries have .**perm** == 0.

Description

This works just like memory_region_notify_iommu(), but it only notifies a specific notifier, not all of them.

int **memory_region_register_iommu_notifier** (MemoryRegion * *mr*, IOMMUNotifier * *n*, Error ** *errp*)
register a notifier for changes to IOMMU translation entries.

Parameters

MemoryRegion * mr the memory region to observe

IOMMUNotifier * n the IOMMUNotifier to be added; the notify callback receives a pointer to an `IOMMUTLBEntry` as the opaque value; the pointer ceases to be valid on exit from the notifier.

Error ** errp pointer to Error*, to store an error if it happens.

Description

Returns 0 on success, or a negative errno otherwise. In particular, -EINVAL indicates that at least one of the attributes of the notifier is not supported (flag/range) by the IOMMU memory region. In case of error the error object must be created.

void **memory_region_iommu_replay** (IOMMUMemoryRegion * *iommu_mr*, IOMMUNotifier * *n*)
replay existing IOMMU translations to a notifier with the minimum page granularity returned by mr->iommu_ops->get_page_size().

Parameters

IOMMUMemoryRegion * iommu_mr the memory region to observe

IOMMUNotifier * n the notifier to which to replay iommu mappings

Note

this is not related to record-and-replay functionality.

void **memory_region_unregister_iommu_notifier** (MemoryRegion * *mr*, IOMMUNotifier * *n*)
unregister a notifier for changes to IOMMU translation entries.

Parameters

MemoryRegion * mr the memory region which was observed and for which notity_stopped() needs to be called

IOMMUNotifier * n the notifier to be removed.

int **memory_region_iommu_get_attr** (IOMMUMemoryRegion * *iommu_mr*, enum IOMMUMemoryRegionAttr *attr*, void * *data*)
return an IOMMU attr if get_attr() is defined on the IOMMU.

Parameters

IOMMUMemoryRegion * iommu_mr the memory region

enum IOMMUMemoryRegionAttr attr the requested attribute

void * data a pointer to the requested attribute data

Description

Returns 0 on success, or a negative errno otherwise. In particular, -EINVAL indicates that the IOMMU does not support the requested attribute.

int **memory_region_iommu_attrs_to_index** (IOMMUMemoryRegion * *iommu_mr*, MemTxAttrs *attrs*)
return the IOMMU index to use for translations with the given memory transaction attributes.

Parameters

IOMMUMemoryRegion * iommu_mr the memory region

MemTxAttrs attrs the memory transaction attributes

int **memory_region_iommu_num_indexes** (IOMMUMemoryRegion * *iommu_mr*)
return the total number of IOMMU indexes that this IOMMU supports.

Parameters

IOMMUMemoryRegion * iommu_mr the memory region

const char * **memory_region_name** (const MemoryRegion * *mr*)
get a memory region's name

Parameters

const MemoryRegion * mr the memory region being queried

Description

Returns the string that was used to initialize the memory region.

bool **memory_region_is_logging** (MemoryRegion * *mr*, uint8_t *client*)
return whether a memory region is logging writes

Parameters

MemoryRegion * mr the memory region being queried

uint8_t client the client being queried

Description

Returns `true` if the memory region is logging writes for the given client

uint8_t **memory_region_get_dirty_log_mask** (MemoryRegion * *mr*)
return the clients for which a memory region is logging writes.

Parameters

MemoryRegion * mr the memory region being queried

Description

Returns a bitmap of clients, in which the DIRTY_MEMORY_* constants are the bit indices.

bool **memory_region_is_rom** (MemoryRegion * *mr*)
check whether a memory region is ROM

Parameters

MemoryRegion * mr the memory region being queried

Description

Returns `true` if a memory region is read-only memory.

bool **memory_region_is_nonvolatile** (MemoryRegion * *mr*)
check whether a memory region is non-volatile

Parameters

MemoryRegion * mr the memory region being queried

Description

Returns `true` is a memory region is non-volatile memory.

int **memory_region_get_fd**(MemoryRegion * *mr*)
Get a file descriptor backing a RAM memory region.

Parameters

MemoryRegion * mr the RAM or alias memory region being queried.

Description

Returns a file descriptor backing a file-based RAM memory region, or -1 if the region is not a file-based RAM memory region.

MemoryRegion * **memory_region_from_host**(void * *ptr*, ram_addr_t * *offset*)
Convert a pointer into a RAM memory region and an offset within it.

Parameters

void * ptr the host pointer to be converted

ram_addr_t * offset the offset within memory region

Description

Given a host pointer inside a RAM memory region (created with memory_region_init_ram() or memory_region_init_ram_ptr()), return the MemoryRegion and the offset within it.

Use with care; by the time this function returns, the returned pointer is not protected by RCU anymore. If the caller is not within an RCU critical section and does not hold the iothread lock, it must have other means of protecting the pointer, such as a reference to the region that includes the incoming ram_addr_t.

void * **memory_region_get_ram_ptr**(MemoryRegion * *mr*)
Get a pointer into a RAM memory region.

Parameters

MemoryRegion * mr the memory region being queried.

Description

Returns a host pointer to a RAM memory region (created with memory_region_init_ram() or memory_region_init_ram_ptr()).

Use with care; by the time this function returns, the returned pointer is not protected by RCU anymore. If the caller is not within an RCU critical section and does not hold the iothread lock, it must have other means of protecting the pointer, such as a reference to the region that includes the incoming ram_addr_t.

void **memory_region_msync**(MemoryRegion * *mr*, hwaddr *addr*, hwaddr *size*)
Synchronize selected address range of a memory mapped region

Parameters

MemoryRegion * mr the memory region to be msync

hwaddr addr the initial address of the range to be sync

hwaddr size the size of the range to be sync

void **memory_region_writeback**(MemoryRegion * *mr*, hwaddr *addr*, hwaddr *size*)
Trigger cache writeback for selected address range

Parameters

MemoryRegion * mr the memory region to be updated

hwaddr addr the initial address of the range to be written back

hwaddr size the size of the range to be written back

void **memory_region_set_log** (MemoryRegion * *mr*, bool *log*, unsigned *client*)
Turn dirty logging on or off for a region.

Parameters

MemoryRegion * mr the memory region being updated.

bool log whether dirty logging is to be enabled or disabled.

unsigned client the user of the logging information; `DIRTY_MEMORY_VGA` only.

Description

Turns dirty logging on or off for a specified client (display, migration). Only meaningful for RAM regions.

void **memory_region_set_dirty** (MemoryRegion * *mr*, hwaddr *addr*, hwaddr *size*)
Mark a range of bytes as dirty in a memory region.

Parameters

MemoryRegion * mr the memory region being dirtied.

hwaddr addr the address (relative to the start of the region) being dirtied.

hwaddr size size of the range being dirtied.

Description

Marks a range of bytes as dirty, after it has been dirtied outside guest code.

void **memory_region_clear_dirty_bitmap** (MemoryRegion * *mr*, hwaddr *start*, hwaddr *len*)
clear dirty bitmap for memory range

Parameters

MemoryRegion * mr the memory region to clear the dirty log upon

hwaddr start start address offset within the memory region

hwaddr len length of the memory region to clear dirty bitmap

Description

This function is called when the caller wants to clear the remote dirty bitmap of a memory range within the memory region. This can be used by e.g. KVM to manually clear dirty log when KVM_CAP_MANUAL_DIRTY_LOG_PROTECT is declared support by the host kernel.

DirtyBitmapSnapshot * **memory_region_snapshot_and_clear_dirty** (MemoryRegion * *mr*, hwaddr *addr*, hwaddr *size*, unsigned *client*)
Get a snapshot of the dirty bitmap and clear it.

Parameters

MemoryRegion * mr the memory region being queried.

hwaddr addr the address (relative to the start of the region) being queried.

hwaddr size the size of the range being queried.

unsigned client the user of the logging information; typically `DIRTY_MEMORY_VGA`.

Description

Creates a snapshot of the dirty bitmap, clears the dirty bitmap and returns the snapshot. The snapshot can then be used to query dirty status, using memory_region_snapshot_get_dirty. Snapshotting allows querying the same page multiple times, which is especially useful for display updates where the scanlines often are not page aligned.

The dirty bitmap region which gets copyed into the snapshot (and cleared afterwards) can be larger than requested. The boundaries are rounded up/down so complete bitmap longs (covering 64 pages on 64bit hosts) can be copied over into the bitmap snapshot. Which isn't a problem for display updates as the extra pages are outside the visible area, and in case the visible area changes a full display redraw is due anyway. Should other use cases for this function emerge we might have to revisit this implementation detail.

Use g_free to release DirtyBitmapSnapshot.

bool **memory_region_snapshot_get_dirty**(MemoryRegion * *mr*, DirtyBitmapSnapshot * *snap*, hwaddr *addr*, hwaddr *size*)
Check whether a range of bytes is dirty in the specified dirty bitmap snapshot.

Parameters

MemoryRegion * mr the memory region being queried.

DirtyBitmapSnapshot * snap the dirty bitmap snapshot

hwaddr addr the address (relative to the start of the region) being queried.

hwaddr size the size of the range being queried.

void **memory_region_reset_dirty**(MemoryRegion * *mr*, hwaddr *addr*, hwaddr *size*, unsigned *client*)
Mark a range of pages as clean, for a specified client.

Parameters

MemoryRegion * mr the region being updated.

hwaddr addr the start of the subrange being cleaned.

hwaddr size the size of the subrange being cleaned.

unsigned client the user of the logging information; `DIRTY_MEMORY_MIGRATION` or `DIRTY_MEMORY_VGA`.

Description

Marks a range of pages as no longer dirty.

void **memory_region_flush_rom_device**(MemoryRegion * *mr*, hwaddr *addr*, hwaddr *size*)
Mark a range of pages dirty and invalidate TBs (for self-modifying code).

Parameters

MemoryRegion * mr the region being flushed.

hwaddr addr the start, relative to the start of the region, of the range being flushed.

hwaddr size the size, in bytes, of the range being flushed.

Description

The MemoryRegionOps->write() callback of a ROM device must use this function to mark byte ranges that have been modified internally, such as by directly accessing the memory returned by memory_region_get_ram_ptr().

This function marks the range dirty and invalidates TBs so that TCG can detect self-modifying code.

void **memory_region_set_readonly**(MemoryRegion * *mr*, bool *readonly*)
Turn a memory region read-only (or read-write)

Parameters

MemoryRegion * mr the region being updated.

bool readonly whether rhe region is to be ROM or RAM.

Description

Allows a memory region to be marked as read-only (turning it into a ROM). only useful on RAM regions.

void **`memory_region_set_nonvolatile`**(MemoryRegion * *mr*, bool *nonvolatile*)
Turn a memory region non-volatile

Parameters

`MemoryRegion * mr` the region being updated.

`bool nonvolatile` whether rhe region is to be non-volatile.

Description

Allows a memory region to be marked as non-volatile. only useful on RAM regions.

void **`memory_region_rom_device_set_romd`**(MemoryRegion * *mr*, bool *romd_mode*)
enable/disable ROMD mode

Parameters

`MemoryRegion * mr` the memory region to be updated

`bool romd_mode` `true` to put the region into ROMD mode

Description

Allows a ROM device (initialized with memory_region_init_rom_device() to set to ROMD mode (default) or MMIO mode. When it is in ROMD mode, the device is mapped to guest memory and satisfies read access directly. When in MMIO mode, reads are forwarded to the `MemoryRegion.read` function. Writes are always handled by the `MemoryRegion.write` function.

void **`memory_region_set_coalescing`**(MemoryRegion * *mr*)
Enable memory coalescing for the region.

Parameters

`MemoryRegion * mr` the memory region to be write coalesced

Description

Enabled writes to a region to be queued for later processing. MMIO ->write callbacks may be delayed until a non-coalesced MMIO is issued. Only useful for IO regions. Roughly similar to write-combining hardware.

void **`memory_region_add_coalescing`**(MemoryRegion * *mr*, hwaddr *offset*, uint64_t *size*)
Enable memory coalescing for a sub-range of a region.

Parameters

`MemoryRegion * mr` the memory region to be updated.

`hwaddr offset` the start of the range within the region to be coalesced.

`uint64_t size` the size of the subrange to be coalesced.

Description

Like memory_region_set_coalescing(), but works on a sub-range of a region. Multiple calls can be issued coalesced disjoint ranges.

void **`memory_region_clear_coalescing`**(MemoryRegion * *mr*)
Disable MMIO coalescing for the region.

Parameters

`MemoryRegion * mr` the memory region to be updated.

Description

Disables any coalescing caused by memory_region_set_coalescing() or memory_region_add_coalescing(). Roughly equivalent to uncacheble memory hardware.

void **memory_region_set_flush_coalesced**(MemoryRegion * *mr*)
Enforce memory coalescing flush before accesses.

Parameters

MemoryRegion * mr the memory region to be updated.

Description

Ensure that pending coalesced MMIO request are flushed before the memory region is accessed. This property is automatically enabled for all regions passed to memory_region_set_coalescing() and memory_region_add_coalescing().

void **memory_region_clear_flush_coalesced**(MemoryRegion * *mr*)
Disable memory coalescing flush before accesses.

Parameters

MemoryRegion * mr the memory region to be updated.

Description

Clear the automatic coalesced MMIO flushing enabled via memory_region_set_flush_coalesced. Note that this service has no effect on memory regions that have MMIO coalescing enabled for themselves. For them, automatic flushing will stop once coalescing is disabled.

void **memory_region_clear_global_locking**(MemoryRegion * *mr*)
Declares that access processing does not depend on the QEMU global lock.

Parameters

MemoryRegion * mr the memory region to be updated.

Description

By clearing this property, accesses to the memory region will be processed outside of QEMU's global lock (unless the lock is held on when issuing the access request). In this case, the device model implementing the access handlers is responsible for synchronization of concurrency.

void **memory_region_add_eventfd**(MemoryRegion * *mr*, hwaddr *addr*, unsigned *size*, bool *match_data*, uint64_t *data*, EventNotifier * *e*)
Request an eventfd to be triggered when a word is written to a location.

Parameters

MemoryRegion * mr the memory region being updated.

hwaddr addr the address within **mr** that is to be monitored

unsigned size the size of the access to trigger the eventfd

bool match_data whether to match against **data**, instead of just **addr**

uint64_t data the data to match against the guest write

EventNotifier * e event notifier to be triggered when **addr**, **size**, and **data** all match.

Description

Marks a word in an IO region (initialized with memory_region_init_io()) as a trigger for an eventfd event. The I/O callback will not be called. The caller must be prepared to handle failure (that is, take the required action if the callback _is_ called).

void **memory_region_del_eventfd** (MemoryRegion * *mr*, hwaddr *addr*, unsigned *size*, bool *match_data*, uint64_t *data*, EventNotifier * *e*)

Cancel an eventfd.

Parameters

MemoryRegion * mr the memory region being updated.

hwaddr addr the address within **mr** that is to be monitored

unsigned size the size of the access to trigger the eventfd

bool match_data whether to match against **data**, instead of just **addr**

uint64_t data the data to match against the guest write

EventNotifier * e event notifier to be triggered when **addr**, **size**, and **data** all match.

Description

Cancels an eventfd trigger requested by a previous memory_region_add_eventfd() call.

void **memory_region_add_subregion** (MemoryRegion * *mr*, hwaddr *offset*, MemoryRegion * *subregion*)

Add a subregion to a container.

Parameters

MemoryRegion * mr the region to contain the new subregion; must be a container initialized with memory_region_init().

hwaddr offset the offset relative to **mr** where **subregion** is added.

MemoryRegion * subregion the subregion to be added.

Description

Adds a subregion at **offset**. The subregion may not overlap with other subregions (except for those explicitly marked as overlapping). A region may only be added once as a subregion (unless removed with memory_region_del_subregion()); use memory_region_init_alias() if you want a region to be a subregion in multiple locations.

void **memory_region_add_subregion_overlap** (MemoryRegion * *mr*, hwaddr *offset*, MemoryRegion * *subregion*, int *priority*)

Add a subregion to a container with overlap.

Parameters

MemoryRegion * mr the region to contain the new subregion; must be a container initialized with memory_region_init().

hwaddr offset the offset relative to **mr** where **subregion** is added.

MemoryRegion * subregion the subregion to be added.

int priority used for resolving overlaps; highest priority wins.

Description

Adds a subregion at **offset**. The subregion may overlap with other subregions. Conflicts are resolved by having a higher **priority** hide a lower **priority**. Subregions without priority are taken as **priority** 0. A region may only be added once as a subregion (unless removed with memory_region_del_subregion()); use memory_region_init_alias() if you want a region to be a subregion in multiple locations.

ram_addr_t **memory_region_get_ram_addr** (MemoryRegion * *mr*)

Get the ram address associated with a memory region

Parameters

`MemoryRegion * mr` the region to be queried

void **`memory_region_del_subregion`** (MemoryRegion * *mr*, MemoryRegion * *subregion*)
Remove a subregion.

Parameters

`MemoryRegion * mr` the container to be updated.

`MemoryRegion * subregion` the region being removed; must be a current subregion of **mr**.

Description

Removes a subregion from its container.

bool **`memory_region_present`** (MemoryRegion * *container*, hwaddr *addr*)
checks if an address relative to a **container** translates into `MemoryRegion` within **container**

Parameters

`MemoryRegion * container` a `MemoryRegion` within which **addr** is a relative address

`hwaddr addr` the area within **container** to be searched

Description

Answer whether a `MemoryRegion` within **container** covers the address **addr**.

bool **`memory_region_is_mapped`** (MemoryRegion * *mr*)
returns true if `MemoryRegion` is mapped into any address space.

Parameters

`MemoryRegion * mr` a `MemoryRegion` which should be checked if it's mapped

MemoryRegionSection **`memory_region_find`** (MemoryRegion * *mr*, hwaddr *addr*, uint64_t *size*)
translate an address/size relative to a MemoryRegion into a *`MemoryRegionSection`*.

Parameters

`MemoryRegion * mr` a MemoryRegion within which **addr** is a relative address

`hwaddr addr` start of the area within **as** to be searched

`uint64_t size` size of the area to be searched

Description

Locates the first `MemoryRegion` within **mr** that overlaps the range given by **addr** and **size**.

Returns a *`MemoryRegionSection`* that describes a contiguous overlap. It will have the following characteristics: - **size** = 0 iff no overlap was found - **mr** is non-`NULL` iff an overlap was found

Remember that in the return value the **offset_within_region** is relative to the returned region (in the .**mr** field), not to the **mr** argument.

Similarly, the .**offset_within_address_space** is relative to the address space that contains both regions, the passed and the returned one. However, in the special case where the **mr** argument has no container (and thus is the root of the address space), the following will hold: - **offset_within_address_space** >= **addr** - **offset_within_address_space** + .**size** <= **addr** + **size**

void **`memory_global_dirty_log_sync`** (void)
synchronize the dirty log for all memory

Parameters

void no arguments

Description

Synchronizes the dirty page log for all address spaces.

void **memory_global_after_dirty_log_sync**(void)
synchronize the dirty log for all memory

Parameters

void no arguments

Description

Synchronizes the vCPUs with a thread that is reading the dirty bitmap. This function must be called after the dirty log bitmap is cleared, and before dirty guest memory pages are read. If you are using `DirtyBitmapSnapshot`, memory_region_snapshot_and_clear_dirty() takes care of doing this.

void **memory_region_transaction_begin**(void)
Start a transaction.

Parameters

void no arguments

Description

During a transaction, changes will be accumulated and made visible only when the transaction ends (is committed).

void **memory_region_transaction_commit**(void)
Commit a transaction and make changes visible to the guest.

Parameters

void no arguments

void **memory_listener_register**(*MemoryListener * listener*, *AddressSpace * filter*)
register callbacks to be called when memory sections are mapped or unmapped into an address space

Parameters

MemoryListener * listener an object containing the callbacks to be called

AddressSpace * filter if non-`NULL`, only regions in this address space will be observed

void **memory_listener_unregister**(*MemoryListener * listener*)
undo the effect of memory_listener_register()

Parameters

MemoryListener * listener an object containing the callbacks to be removed

void **memory_global_dirty_log_start**(void)
begin dirty logging for all regions

Parameters

void no arguments

void **memory_global_dirty_log_stop**(void)
end dirty logging for all regions

Parameters

void no arguments

MemTxResult **memory_region_dispatch_read** (MemoryRegion * *mr*, hwaddr *addr*, uint64_t * *pval*, MemOp *op*, MemTxAttrs *attrs*)
perform a read directly to the specified MemoryRegion.

Parameters

MemoryRegion * mr `MemoryRegion` to access

hwaddr addr address within that region

uint64_t * pval pointer to uint64_t which the data is written to

MemOp op size, sign, and endianness of the memory operation

MemTxAttrs attrs memory transaction attributes to use for the access

MemTxResult **memory_region_dispatch_write** (MemoryRegion * *mr*, hwaddr *addr*, uint64_t *data*, MemOp *op*, MemTxAttrs *attrs*)
perform a write directly to the specified MemoryRegion.

Parameters

MemoryRegion * mr `MemoryRegion` to access

hwaddr addr address within that region

uint64_t data data to write

MemOp op size, sign, and endianness of the memory operation

MemTxAttrs attrs memory transaction attributes to use for the access

void **address_space_init** (*AddressSpace* * *as*, MemoryRegion * *root*, const char * *name*)
initializes an address space

Parameters

AddressSpace * as an uninitialized `AddressSpace`

MemoryRegion * root a `MemoryRegion` that routes addresses for the address space

const char * name an address space name. The name is only used for debugging output.

void **address_space_destroy** (*AddressSpace* * *as*)
destroy an address space

Parameters

AddressSpace * as address space to be destroyed

Description

Releases all resources associated with an address space. After an address space is destroyed, its root memory region (given by address_space_init()) may be destroyed as well.

void **address_space_remove_listeners** (*AddressSpace* * *as*)
unregister all listeners of an address space

Parameters

AddressSpace * as an initialized `AddressSpace`

Description

Removes all callbacks previously registered with memory_listener_register() for **as**.

MemTxResult **address_space_rw** (*AddressSpace* * *as*, hwaddr *addr*, MemTxAttrs *attrs*, void * *buf*, hwaddr *len*, bool *is_write*)
read from or write to an address space.

Parameters

`AddressSpace * as` *`AddressSpace`* to be accessed

`hwaddr addr` address within that address space

`MemTxAttrs attrs` memory transaction attributes

`void * buf` buffer with the data transferred

`hwaddr len` the number of bytes to read or write

`bool is_write` indicates the transfer direction

Description

Return a MemTxResult indicating whether the operation succeeded or failed (eg unassigned memory, device rejected the transaction, IOMMU fault).

MemTxResult **`address_space_write`** (*AddressSpace* * *as*, hwaddr *addr*, MemTxAttrs *attrs*, const void * *buf*, hwaddr *len*)

write to address space.

Parameters

`AddressSpace * as` *`AddressSpace`* to be accessed

`hwaddr addr` address within that address space

`MemTxAttrs attrs` memory transaction attributes

`const void * buf` buffer with the data transferred

`hwaddr len` the number of bytes to write

Description

Return a MemTxResult indicating whether the operation succeeded or failed (eg unassigned memory, device rejected the transaction, IOMMU fault).

MemTxResult **`address_space_write_rom`** (*AddressSpace* * *as*, hwaddr *addr*, MemTxAttrs *attrs*, const void * *buf*, hwaddr *len*)

write to address space, including ROM.

Parameters

`AddressSpace * as` *`AddressSpace`* to be accessed

`hwaddr addr` address within that address space

`MemTxAttrs attrs` memory transaction attributes

`const void * buf` buffer with the data transferred

`hwaddr len` the number of bytes to write

Description

This function writes to the specified address space, but will write data to both ROM and RAM. This is used for non-guest writes like writes from the gdb debug stub or initial loading of ROM contents.

Note that portions of the write which attempt to write data to a device will be silently ignored – only real RAM and ROM will be written to.

Return a MemTxResult indicating whether the operation succeeded or failed (eg unassigned memory, device rejected the transaction, IOMMU fault).

void **`address_space_cache_invalidate`**(MemoryRegionCache * *cache*, hwaddr *addr*, hwaddr *access_len*)
complete a write to a `MemoryRegionCache`

Parameters

`MemoryRegionCache * cache` The `MemoryRegionCache` to operate on.

`hwaddr addr` The first physical address that was written, relative to the address that was passed to **address_space_cache_init**.

`hwaddr access_len` The number of bytes that were written starting at **addr**.

void **`address_space_cache_destroy`**(MemoryRegionCache * *cache*)
free a `MemoryRegionCache`

Parameters

`MemoryRegionCache * cache` The `MemoryRegionCache` whose memory should be released.

MemTxResult **`address_space_read`**(*AddressSpace* * *as*, hwaddr *addr*, MemTxAttrs *attrs*, void * *buf*, hwaddr *len*)
read from an address space.

Parameters

`AddressSpace * as` *`AddressSpace`* to be accessed

`hwaddr addr` address within that address space

`MemTxAttrs attrs` memory transaction attributes

`void * buf` buffer with the data transferred

`hwaddr len` length of the data transferred

Description

Return a MemTxResult indicating whether the operation succeeded or failed (eg unassigned memory, device rejected the transaction, IOMMU fault). Called within RCU critical section.

MemTxResult **`address_space_read_cached`**(MemoryRegionCache * *cache*, hwaddr *addr*, void * *buf*, hwaddr *len*)
read from a cached RAM region

Parameters

`MemoryRegionCache * cache` Cached region to be addressed

`hwaddr addr` address relative to the base of the RAM region

`void * buf` buffer with the data transferred

`hwaddr len` length of the data transferred

MemTxResult **`address_space_write_cached`**(MemoryRegionCache * *cache*, hwaddr *addr*, const void * *buf*, hwaddr *len*)
write to a cached RAM region

Parameters

`MemoryRegionCache * cache` Cached region to be addressed

`hwaddr addr` address relative to the base of the RAM region

`const void * buf` buffer with the data transferred

`hwaddr len` length of the data transferred

6.5 Migration

QEMU has code to load/save the state of the guest that it is running. These are two complementary operations. Saving the state just does that, saves the state for each device that the guest is running. Restoring a guest is just the opposite operation: we need to load the state of each device.

For this to work, QEMU has to be launched with the same arguments the two times. I.e. it can only restore the state in one guest that has the same devices that the one it was saved (this last requirement can be relaxed a bit, but for now we can consider that configuration has to be exactly the same).

Once that we are able to save/restore a guest, a new functionality is requested: migration. This means that QEMU is able to start in one machine and being "migrated" to another machine. I.e. being moved to another machine.

Next was the "live migration" functionality. This is important because some guests run with a lot of state (specially RAM), and it can take a while to move all state from one machine to another. Live migration allows the guest to continue running while the state is transferred. Only while the last part of the state is transferred has the guest to be stopped. Typically the time that the guest is unresponsive during live migration is the low hundred of milliseconds (notice that this depends on a lot of things).

6.5.1 Transports

The migration stream is normally just a byte stream that can be passed over any transport.

- tcp migration: do the migration using tcp sockets
- unix migration: do the migration using unix sockets
- exec migration: do the migration using the stdin/stdout through a process.
- fd migration: do the migration using a file descriptor that is passed to QEMU. QEMU doesn't care how this file descriptor is opened.

In addition, support is included for migration using RDMA, which transports the page data using `RDMA`, where the hardware takes care of transporting the pages, and the load on the CPU is much lower. While the internals of RDMA migration are a bit different, this isn't really visible outside the RAM migration code.

All these migration protocols use the same infrastructure to save/restore state devices. This infrastructure is shared with the savevm/loadvm functionality.

6.5.2 Debugging

The migration stream can be analyzed thanks to *scripts/analyze_migration.py*.

Example usage:

```
$ qemu-system-x86_64
 (qemu) migrate "exec:cat > mig"
$ ./scripts/analyze_migration.py -f mig
{
  "ram (3)": {
      "section sizes": {
          "pc.ram": "0x0000000008000000",
...
```

See also `analyze_migration.py -h` help for more options.

6.5.3 Common infrastructure

The files, sockets or fd's that carry the migration stream are abstracted by the `QEMUFile` type (see *migration/qemu-file.h*). In most cases this is connected to a subtype of `QIOChannel` (see *io/*).

6.5.4 Saving the state of one device

For most devices, the state is saved in a single call to the migration infrastructure; these are *non-iterative* devices. The data for these devices is sent at the end of precopy migration, when the CPUs are paused. There are also *iterative* devices, which contain a very large amount of data (e.g. RAM or large tables). See the iterative device section below.

General advice for device developers

- The migration state saved should reflect the device being modelled rather than the way your implementation works. That way if you change the implementation later the migration stream will stay compatible. That model may include internal state that's not directly visible in a register.
- When saving a migration stream the device code may walk and check the state of the device. These checks might fail in various ways (e.g. discovering internal state is corrupt or that the guest has done something bad). Consider carefully before asserting/aborting at this point, since the normal response from users is that *migration broke their VM* since it had apparently been running fine until then. In these error cases, the device should log a message indicating the cause of error, and should consider putting the device into an error state, allowing the rest of the VM to continue execution.
- The migration might happen at an inconvenient point, e.g. right in the middle of the guest reprogramming the device, during guest reboot or shutdown or while the device is waiting for external IO. It's strongly preferred that migrations do not fail in this situation, since in the cloud environment migrations might happen automatically to VMs that the administrator doesn't directly control.
- If you do need to fail a migration, ensure that sufficient information is logged to identify what went wrong.
- The destination should treat an incoming migration stream as hostile (which we do to varying degrees in the existing code). Check that offsets into buffers and the like can't cause overruns. Fail the incoming migration in the case of a corrupted stream like this.
- Take care with internal device state or behaviour that might become migration version dependent. For example, the order of PCI capabilities is required to stay constant across migration. Another example would be that a special case handled by subsections (see below) might become much more common if a default behaviour is changed.
- The state of the source should not be changed or destroyed by the outgoing migration. Migrations timing out or being failed by higher levels of management, or failures of the destination host are not unusual, and in that case the VM is restarted on the source. Note that the management layer can validly revert the migration even though the QEMU level of migration has succeeded as long as it does it before starting execution on the destination.
- Buses and devices should be able to explicitly specify addresses when instantiated, and management tools should use those. For example, when hot adding USB devices it's important to specify the ports and addresses, since implicit ordering based on the command line order may be different on the destination. This can result in the device state being loaded into the wrong device.

VMState

Most device data can be described using the `VMSTATE` macros (mostly defined in `include/migration/vmstate.h`).

An example (from hw/input/pckbd.c)

```
static const VMStateDescription vmstate_kbd = {
    .name = "pckbd",
    .version_id = 3,
    .minimum_version_id = 3,
    .fields = (VMStateField[]) {
        VMSTATE_UINT8(write_cmd, KBDState),
        VMSTATE_UINT8(status, KBDState),
        VMSTATE_UINT8(mode, KBDState),
        VMSTATE_UINT8(pending, KBDState),
        VMSTATE_END_OF_LIST()
    }
};
```

We are declaring the state with name "pckbd". The *version_id* is 3, and the fields are 4 uint8_t in a KBDState structure. We registered this with:

```
vmstate_register(NULL, 0, &vmstate_kbd, s);
```

For devices that are *qdev* based, we can register the device in the class init function:

```
dc->vmsd = &vmstate_kbd_isa;
```

The VMState macros take care of ensuring that the device data section is formatted portably (normally big endian) and make some compile time checks against the types of the fields in the structures.

VMState macros can include other VMStateDescriptions to store substructures (see `VMSTATE_STRUCT_`), arrays (`VMSTATE_ARRAY_`) and variable length arrays (`VMSTATE_VARRAY_`). Various other macros exist for special cases.

Note that the format on the wire is still very raw; i.e. a VMSTATE_UINT32 ends up with a 4 byte bigendian representation on the wire; in the future it might be possible to use a more structured format.

Legacy way

This way is going to disappear as soon as all current users are ported to VMSTATE; although converting existing code can be tricky, and thus 'soon' is relative.

Each device has to register two functions, one to save the state and another to load the state back.

```
int register_savevm_live(const char *idstr,
                         int instance_id,
                         int version_id,
                         SaveVMHandlers *ops,
                         void *opaque);
```

Two functions in the `ops` structure are the *save_state* and *load_state* functions. Notice that *load_state* receives a version_id parameter to know what state format is receiving. *save_state* doesn't have a version_id parameter because it always uses the latest version.

Note that because the VMState macros still save the data in a raw format, in many cases it's possible to replace legacy code with a carefully constructed VMState description that matches the byte layout of the existing code.

Changing migration data structures

When we migrate a device, we save/load the state as a series of fields. Sometimes, due to bugs or new functionality, we need to change the state to store more/different information. Changing the migration state saved for a device can break migration compatibility unless care is taken to use the appropriate techniques. In general QEMU tries to maintain forward migration compatibility (i.e. migrating from QEMU n->n+1) and there are users who benefit from backward compatibility as well.

Subsections

The most common structure change is adding new data, e.g. when adding a newer form of device, or adding that state that you previously forgot to migrate. This is best solved using a subsection.

A subsection is “like” a device vmstate, but with a particularity, it has a Boolean function that tells if that values are needed to be sent or not. If this functions returns false, the subsection is not sent. Subsections have a unique name, that is looked for on the receiving side.

On the receiving side, if we found a subsection for a device that we don’t understand, we just fail the migration. If we understand all the subsections, then we load the state with success. There’s no check that a subsection is loaded, so a newer QEMU that knows about a subsection can (with care) load a stream from an older QEMU that didn’t send the subsection.

If the new data is only needed in a rare case, then the subsection can be made conditional on that case and the migration will still succeed to older QEMUs in most cases. This is OK for data that’s critical, but in some use cases it’s preferred that the migration should succeed even with the data missing. To support this the subsection can be connected to a device property and from there to a versioned machine type.

The ‘pre_load’ and ‘post_load’ functions on subsections are only called if the subsection is loaded.

One important note is that the outer post_load() function is called “after” loading all subsections, because a newer subsection could change the same value that it uses. A flag, and the combination of outer pre_load and post_load can be used to detect whether a subsection was loaded, and to fall back on default behaviour when the subsection isn’t present.

Example:

```
static bool ide_drive_pio_state_needed(void *opaque)
{
    IDEState *s = opaque;

    return ((s->status & DRQ_STAT) != 0)
        || (s->bus->error_status & BM_STATUS_PIO_RETRY);
}

const VMStateDescription vmstate_ide_drive_pio_state = {
    .name = "ide_drive/pio_state",
    .version_id = 1,
    .minimum_version_id = 1,
    .pre_save = ide_drive_pio_pre_save,
    .post_load = ide_drive_pio_post_load,
    .needed = ide_drive_pio_state_needed,
    .fields = (VMStateField[]) {
        VMSTATE_INT32(req_nb_sectors, IDEState),
        VMSTATE_VARRAY_INT32(io_buffer, IDEState, io_buffer_total_len, 1,
                             vmstate_info_uint8, uint8_t),
        VMSTATE_INT32(cur_io_buffer_offset, IDEState),
        VMSTATE_INT32(cur_io_buffer_len, IDEState),
        VMSTATE_UINT8(end_transfer_fn_idx, IDEState),
        VMSTATE_INT32(elementary_transfer_size, IDEState),
        VMSTATE_INT32(packet_transfer_size, IDEState),
        VMSTATE_END_OF_LIST()
    }
```

(continues on next page)

(continued from previous page)

```
};

const VMStateDescription vmstate_ide_drive = {
    .name = "ide_drive",
    .version_id = 3,
    .minimum_version_id = 0,
    .post_load = ide_drive_post_load,
    .fields = (VMStateField[]) {
        .... several fields ....
        VMSTATE_END_OF_LIST()
    },
    .subsections = (const VMStateDescription*[]) {
        &vmstate_ide_drive_pio_state,
        NULL
    }
};
```

Here we have a subsection for the pio state. We only need to save/send this state when we are in the middle of a pio operation (that is what `ide_drive_pio_state_needed()` checks). If DRQ_STAT is not enabled, the values on that fields are garbage and don't need to be sent.

Connecting subsections to properties

Using a condition function that checks a 'property' to determine whether to send a subsection allows backward migration compatibility when new subsections are added, especially when combined with versioned machine types.

For example:

a) Add a new property using `DEFINE_PROP_BOOL` - e.g. support-foo and default it to true.

b) Add an entry to the `hw_compat_` for the previous version that sets the property to false.

c) Add a static bool support_foo function that tests the property.

d) Add a subsection with a .needed set to the support_foo function

e) (potentially) Add an outer pre_load that sets up a default value for 'foo' to be used if the subsection isn't loaded.

Now that subsection will not be generated when using an older machine type and the migration stream will be accepted by older QEMU versions.

Not sending existing elements

Sometimes members of the VMState are no longer needed:

- removing them will break migration compatibility
- making them version dependent and bumping the version will break backward migration compatibility.

Adding a dummy field into the migration stream is normally the best way to preserve compatibility.

If the field really does need to be removed then:

a) Add a new property/compatibility/function in the same way for subsections above.

b) replace the VMSTATE macro with the _TEST version of the macro, e.g.:

   `VMSTATE_UINT32(foo, barstruct)`

   becomes

```
VMSTATE_UINT32_TEST(foo, barstruct, pre_version_baz)
```

Sometime in the future when we no longer care about the ancient versions these can be killed off. Note that for backward compatibility it's important to fill in the structure with data that the destination will understand.

Any difference in the predicates on the source and destination will end up with different fields being enabled and data being loaded into the wrong fields; for this reason conditional fields like this are very fragile.

Versions

Version numbers are intended for major incompatible changes to the migration of a device, and using them breaks backward-migration compatibility; in general most changes can be made by adding Subsections (see above) or _TEST macros (see above) which won't break compatibility.

Each version is associated with a series of fields saved. The *save_state* always saves the state as the newer version. But *load_state* sometimes is able to load state from an older version.

You can see that there are several version fields:

- *version_id*: the maximum version_id supported by VMState for that device.
- *minimum_version_id*: the minimum version_id that VMState is able to understand for that device.
- *minimum_version_id_old*: For devices that were not able to port to vmstate, we can assign a function that knows how to read this old state. This field is ignored if there is no *load_state_old* handler.

VMState is able to read versions from minimum_version_id to version_id. And the function `load_state_old()` (if present) is able to load state from minimum_version_id_old to minimum_version_id. This function is deprecated and will be removed when no more users are left.

There are *_V* forms of many `VMSTATE_` macros to load fields for version dependent fields, e.g.

```
VMSTATE_UINT16_V(ip_id, Slirp, 2),
```

only loads that field for versions 2 and newer.

Saving state will always create a section with the 'version_id' value and thus can't be loaded by any older QEMU.

Massaging functions

Sometimes, it is not enough to be able to save the state directly from one structure, we need to fill the correct values there. One example is when we are using kvm. Before saving the cpu state, we need to ask kvm to copy to QEMU the state that it is using. And the opposite when we are loading the state, we need a way to tell kvm to load the state for the cpu that we have just loaded from the QEMUFile.

The functions to do that are inside a vmstate definition, and are called:

- `int (*pre_load)(void *opaque);`

  This function is called before we load the state of one device.

- `int (*post_load)(void *opaque, int version_id);`

  This function is called after we load the state of one device.

- `int (*pre_save)(void *opaque);`

  This function is called before we save the state of one device.

- `int (*post_save)(void *opaque);`

  This function is called after we save the state of one device (even upon failure, unless the call to pre_save returned an error).

Example: You can look at hpet.c, that uses the first three functions to massage the state that is transferred.

The `VMSTATE_WITH_TMP` macro may be useful when the migration data doesn't match the stored device data well; it allows an intermediate temporary structure to be populated with migration data and then transferred to the main structure.

If you use memory API functions that update memory layout outside initialization (i.e., in response to a guest action), this is a strong indication that you need to call these functions in a *post_load* callback. Examples of such memory API functions are:

- memory_region_add_subregion()
- memory_region_del_subregion()
- memory_region_set_readonly()
- memory_region_set_nonvolatile()
- memory_region_set_enabled()
- memory_region_set_address()
- memory_region_set_alias_offset()

Iterative device migration

Some devices, such as RAM, Block storage or certain platform devices, have large amounts of data that would mean that the CPUs would be paused for too long if they were sent in one section. For these devices an *iterative* approach is taken.

The iterative devices generally don't use VMState macros (although it may be possible in some cases) and instead use qemu_put_*/qemu_get_* macros to read/write data to the stream. Specialist versions exist for high bandwidth IO.

An iterative device must provide:

- A `save_setup` function that initialises the data structures and transmits a first section containing information on the device. In the case of RAM this transmits a list of RAMBlocks and sizes.
- A `load_setup` function that initialises the data structures on the destination.
- A `save_live_pending` function that is called repeatedly and must indicate how much more data the iterative data must save. The core migration code will use this to determine when to pause the CPUs and complete the migration.
- A `save_live_iterate` function (called after `save_live_pending` when there is significant data still to be sent). It should send a chunk of data until the point that stream bandwidth limits tell it to stop. Each call generates one section.
- A `save_live_complete_precopy` function that must transmit the last section for the device containing any remaining data.
- A `load_state` function used to load sections generated by any of the save functions that generate sections.
- `cleanup` functions for both save and load that are called at the end of migration.

Note that the contents of the sections for iterative migration tend to be open-coded by the devices; care should be taken in parsing the results and structuring the stream to make them easy to validate.

Device ordering

There are cases in which the ordering of device loading matters; for example in some systems where a device may assert an interrupt during loading, if the interrupt controller is loaded later then it might lose the state.

Some ordering is implicitly provided by the order in which the machine definition creates devices, however this is somewhat fragile.

The `MigrationPriority` enum provides a means of explicitly enforcing ordering. Numerically higher priorities are loaded earlier. The priority is set by setting the `priority` field of the top level `VMStateDescription` for the device.

6.5.5 Stream structure

The stream tries to be word and endian agnostic, allowing migration between hosts of different characteristics running the same VM.

- Header
  - Magic
  - Version
  - VM configuration section
    - Machine type
    - Target page bits
- List of sections Each section contains a device, or one iteration of a device save.
  - section type
  - section id
  - ID string (First section of each device)
  - instance id (First section of each device)
  - version id (First section of each device)
  - <device data>
  - Footer mark
- EOF mark
- VM Description structure Consisting of a JSON description of the contents for analysis only

The `device data` in each section consists of the data produced by the code described above. For non-iterative devices they have a single section; iterative devices have an initial and last section and a set of parts in between. Note that there is very little checking by the common code of the integrity of the `device data` contents, that's up to the devices themselves. The `footer mark` provides a little bit of protection for the case where the receiving side reads more or less data than expected.

The `ID string` is normally unique, having been formed from a bus name and device address, PCI devices and storage devices hung off PCI controllers fit this pattern well. Some devices are fixed single instances (e.g. "pc-ram"). Others (especially either older devices or system devices which for some reason don't have a bus concept) make use of the `instance id` for otherwise identically named devices.

Return path

Only a unidirectional stream is required for normal migration, however a `return path` can be created when bidirectional communication is desired. This is primarily used by postcopy, but is also used to return a success flag to the source at the end of migration.

`qemu_file_get_return_path(QEMUFile* fwdpath)` gives the QEMUFile* for the return path.

Source side

> Forward path - written by migration thread Return path - opened by main thread, read by return-path thread

Destination side

> Forward path - read by main thread Return path - opened by main thread, written by main thread AND postcopy thread (protected by rp_mutex)

6.5.6 Postcopy

'Postcopy' migration is a way to deal with migrations that refuse to converge (or take too long to converge) its plus side is that there is an upper bound on the amount of migration traffic and time it takes, the down side is that during the postcopy phase, a failure of *either* side or the network connection causes the guest to be lost.

In postcopy the destination CPUs are started before all the memory has been transferred, and accesses to pages that are yet to be transferred cause a fault that's translated by QEMU into a request to the source QEMU.

Postcopy can be combined with precopy (i.e. normal migration) so that if precopy doesn't finish in a given time the switch is made to postcopy.

Enabling postcopy

To enable postcopy, issue this command on the monitor (both source and destination) prior to the start of migration:

`migrate_set_capability postcopy-ram on`

The normal commands are then used to start a migration, which is still started in precopy mode. Issuing:

`migrate_start_postcopy`

will now cause the transition from precopy to postcopy. It can be issued immediately after migration is started or any time later on. Issuing it after the end of a migration is harmless.

Blocktime is a postcopy live migration metric, intended to show how long the vCPU was in state of interruptable sleep due to pagefault. That metric is calculated both for all vCPUs as overlapped value, and separately for each vCPU. These values are calculated on destination side. To enable postcopy blocktime calculation, enter following command on destination monitor:

`migrate_set_capability postcopy-blocktime on`

Postcopy blocktime can be retrieved by query-migrate qmp command. postcopy-blocktime value of qmp command will show overlapped blocking time for all vCPU, postcopy-vcpu-blocktime will show list of blocking time per vCPU.

Note: During the postcopy phase, the bandwidth limits set using `migrate_set_speed` is ignored (to avoid delaying requested pages that the destination is waiting for).

Postcopy device transfer

Loading of device data may cause the device emulation to access guest RAM that may trigger faults that have to be resolved by the source, as such the migration stream has to be able to respond with page data *during* the device load, and hence the device data has to be read from the stream completely before the device load begins to free the stream up. This is achieved by 'packaging' the device data into a blob that's read in one go.

Source behaviour

Until postcopy is entered the migration stream is identical to normal precopy, except for the addition of a 'postcopy advise' command at the beginning, to tell the destination that postcopy might happen. When postcopy starts the source sends the page discard data and then forms the 'package' containing:

- Command: 'postcopy listen'
- The device state

  A series of sections, identical to the precopy streams device state stream containing everything except postcopiable devices (i.e. RAM)
- Command: 'postcopy run'

The 'package' is sent as the data part of a Command: `CMD_PACKAGED`, and the contents are formatted in the same way as the main migration stream.

During postcopy the source scans the list of dirty pages and sends them to the destination without being requested (in much the same way as precopy), however when a page request is received from the destination, the dirty page scanning restarts from the requested location. This causes requested pages to be sent quickly, and also causes pages directly after the requested page to be sent quickly in the hope that those pages are likely to be used by the destination soon.

Destination behaviour

Initially the destination looks the same as precopy, with a single thread reading the migration stream; the 'postcopy advise' and 'discard' commands are processed to change the way RAM is managed, but don't affect the stream processing.

```
------------------------------------------------------------------------------
                        1      2   3     4 5                      6   7
main -----DISCARD-CMD_PACKAGED ( LISTEN  DEVICE     DEVICE DEVICE RUN )
thread                             |       |
                                   |     (page request)
                                   |        \___
                                   v            \
listen thread:                     --- page -- page -- page -- page -- page --

                                   a   b        c
------------------------------------------------------------------------------
```

- On receipt of `CMD_PACKAGED` (1)

  All the data associated with the package - the (...) section in the diagram - is read into memory, and the main thread recurses into qemu_loadvm_state_main to process the contents of the package (2) which contains commands (3,6) and devices (4...)
- On receipt of 'postcopy listen' - 3 -(i.e. the 1st command in the package)

  a new thread (a) is started that takes over servicing the migration stream, while the main thread carries on loading the package. It loads normal background page data (b) but if during a device load a fault happens (5) the returned page (c) is loaded by the listen thread allowing the main threads device load to carry on.

- The last thing in the `CMD_PACKAGED` is a ‘RUN’ command (6)

  letting the destination CPUs start running. At the end of the `CMD_PACKAGED` (7) the main thread returns to normal running behaviour and is no longer used by migration, while the listen thread carries on servicing page data until the end of migration.

Postcopy states

Postcopy moves through a series of states (see postcopy_state) from ADVISE->DISCARD->LISTEN->RUNNING->END

- Advise

  Set at the start of migration if postcopy is enabled, even if it hasn’t had the start command; here the destination checks that its OS has the support needed for postcopy, and performs setup to ensure the RAM mappings are suitable for later postcopy. The destination will fail early in migration at this point if the required OS support is not present. (Triggered by reception of POSTCOPY_ADVISE command)

- Discard

  Entered on receipt of the first ‘discard’ command; prior to the first Discard being performed, hugepages are switched off (using madvise) to ensure that no new huge pages are created during the postcopy phase, and to cause any huge pages that have discards on them to be broken.

- Listen

  The first command in the package, POSTCOPY_LISTEN, switches the destination state to Listen, and starts a new thread (the ‘listen thread’) which takes over the job of receiving pages off the migration stream, while the main thread carries on processing the blob. With this thread able to process page reception, the destination now ‘sensitises’ the RAM to detect any access to missing pages (on Linux using the ‘userfault’ system).

- Running

  POSTCOPY_RUN causes the destination to synchronise all state and start the CPUs and IO devices running. The main thread now finishes processing the migration package and now carries on as it would for normal precopy migration (although it can’t do the cleanup it would do as it finishes a normal migration).

- End

  The listen thread can now quit, and perform the cleanup of migration state, the migration is now complete.

Source side page maps

The source side keeps two bitmaps during postcopy; ‘the migration bitmap’ and ‘unsent map’. The ‘migration bitmap’ is basically the same as in the precopy case, and holds a bit to indicate that page is ‘dirty’ - i.e. needs sending. During the precopy phase this is updated as the CPU dirties pages, however during postcopy the CPUs are stopped and nothing should dirty anything any more.

The ‘unsent map’ is used for the transition to postcopy. It is a bitmap that has a bit cleared whenever a page is sent to the destination, however during the transition to postcopy mode it is combined with the migration bitmap to form a set of pages that:

a) Have been sent but then redirtied (which must be discarded)

b) Have not yet been sent - which also must be discarded to cause any transparent huge pages built during precopy to be broken.

Note that the contents of the unsentmap are sacrificed during the calculation of the discard set and thus aren't valid once in postcopy. The dirtymap is still valid and is used to ensure that no page is sent more than once. Any request for a page that has already been sent is ignored. Duplicate requests such as this can happen as a page is sent at about the same time the destination accesses it.

Postcopy with hugepages

Postcopy now works with hugetlbfs backed memory:

a) The linux kernel on the destination must support userfault on hugepages.

b) The huge-page configuration on the source and destination VMs must be identical; i.e. RAMBlocks on both sides must use the same page size.

c) Note that `-mem-path /dev/hugepages` will fall back to allocating normal RAM if it doesn't have enough hugepages, triggering (b) to fail. Using `-mem-prealloc` enforces the allocation using hugepages.

d) Care should be taken with the size of hugepage used; postcopy with 2MB hugepages works well, however 1GB hugepages are likely to be problematic since it takes ~1 second to transfer a 1GB hugepage across a 10Gbps link, and until the full page is transferred the destination thread is blocked.

Postcopy with shared memory

Postcopy migration with shared memory needs explicit support from the other processes that share memory and from QEMU. There are restrictions on the type of memory that userfault can support shared.

The Linux kernel userfault support works on */dev/shm* memory and on *hugetlbfs* (although the kernel doesn't provide an equivalent to *madvise(MADV_DONTNEED)* for hugetlbfs which may be a problem in some configurations).

The vhost-user code in QEMU supports clients that have Postcopy support, and the *vhost-user-bridge* (in *tests/*) and the DPDK package have changes to support postcopy.

The client needs to open a userfaultfd and register the areas of memory that it maps with userfault. The client must then pass the userfaultfd back to QEMU together with a mapping table that allows fault addresses in the clients address space to be converted back to RAMBlock/offsets. The client's userfaultfd is added to the postcopy fault-thread and page requests are made on behalf of the client by QEMU. QEMU performs 'wake' operations on the client's userfaultfd to allow it to continue after a page has arrived.

Note:

There are two future improvements that would be nice:

a) Some way to make QEMU ignorant of the addresses in the clients address space

b) Avoiding the need for QEMU to perform ufd-wake calls after the pages have arrived

---

Retro-fitting postcopy to existing clients is possible:

a) A mechanism is needed for the registration with userfault as above, and the registration needs to be co-ordinated with the phases of postcopy. In vhost-user extra messages are added to the existing control channel.

b) Any thread that can block due to guest memory accesses must be identified and the implication understood; for example if the guest memory access is made while holding a lock then all other threads waiting for that lock will also be blocked.

6.5.7 Firmware

Migration migrates the copies of RAM and ROM, and thus when running on the destination it includes the firmware from the source. Even after resetting a VM, the old firmware is used. Only once QEMU has been restarted is the new firmware in use.

- Changes in firmware size can cause changes in the required RAMBlock size to hold the firmware and thus migration can fail. In practice it's best to pad firmware images to convenient powers of 2 with plenty of space for growth.
- Care should be taken with device emulation code so that newer emulation code can work with older firmware to allow forward migration.
- Care should be taken with newer firmware so that backward migration to older systems with older device emulation code will work.

In some cases it may be best to tie specific firmware versions to specific versioned machine types to cut down on the combinations that will need support. This is also useful when newer versions of firmware outgrow the padding.

6.6 Atomic operations in QEMU

CPUs perform independent memory operations effectively in random order. but this can be a problem for CPU-CPU interaction (including interactions between QEMU and the guest). Multi-threaded programs use various tools to instruct the compiler and the CPU to restrict the order to something that is consistent with the expectations of the programmer.

The most basic tool is locking. Mutexes, condition variables and semaphores are used in QEMU, and should be the default approach to synchronization. Anything else is considerably harder, but it's also justified more often than one would like; the most performance-critical parts of QEMU in particular require a very low level approach to concurrency, involving memory barriers and atomic operations. The semantics of concurrent memory accesses are governed by the C11 memory model.

QEMU provides a header, `qemu/atomic.h`, which wraps C11 atomics to provide better portability and a less verbose syntax. `qemu/atomic.h` provides macros that fall in three camps:

- compiler barriers: `barrier()`;
- weak atomic access and manual memory barriers: `atomic_read()`, `atomic_set()`, `smp_rmb()`, `smp_wmb()`, `smp_mb()`, `smp_mb_acquire()`, `smp_mb_release()`, `smp_read_barrier_depends()`;
- sequentially consistent atomic access: everything else.

In general, use of `qemu/atomic.h` should be wrapped with more easily used data structures (e.g. the lock-free singly-linked list operations `QSLIST_INSERT_HEAD_ATOMIC` and `QSLIST_MOVE_ATOMIC`) or synchronization primitives (such as RCU, `QemuEvent` or `QemuLockCnt`). Bare use of atomic operations and memory barriers should be limited to inter-thread checking of flags and documented thoroughly.

6.6.1 Compiler memory barrier

`barrier()` prevents the compiler from moving the memory accesses on either side of it to the other side. The compiler barrier has no direct effect on the CPU, which may then reorder things however it wishes.

`barrier()` is mostly used within `qemu/atomic.h` itself. On some architectures, CPU guarantees are strong enough that blocking compiler optimizations already ensures the correct order of execution. In this case, `qemu/atomic.h` will reduce stronger memory barriers to simple compiler barriers.

Still, `barrier()` can be useful when writing code that can be interrupted by signal handlers.

6.6.2 Sequentially consistent atomic access

Most of the operations in the `qemu/atomic.h` header ensure *sequential consistency*, where "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".

`qemu/atomic.h` provides the following set of atomic read-modify-write operations:

```
void atomic_inc(ptr)
void atomic_dec(ptr)
void atomic_add(ptr, val)
void atomic_sub(ptr, val)
void atomic_and(ptr, val)
void atomic_or(ptr, val)

typeof(*ptr) atomic_fetch_inc(ptr)
typeof(*ptr) atomic_fetch_dec(ptr)
typeof(*ptr) atomic_fetch_add(ptr, val)
typeof(*ptr) atomic_fetch_sub(ptr, val)
typeof(*ptr) atomic_fetch_and(ptr, val)
typeof(*ptr) atomic_fetch_or(ptr, val)
typeof(*ptr) atomic_fetch_xor(ptr, val)
typeof(*ptr) atomic_fetch_inc_nonzero(ptr)
typeof(*ptr) atomic_xchg(ptr, val)
typeof(*ptr) atomic_cmpxchg(ptr, old, new)
```

all of which return the old value of `*ptr`. These operations are polymorphic; they operate on any type that is as wide as a pointer or smaller.

Similar operations return the new value of `*ptr`:

```
typeof(*ptr) atomic_inc_fetch(ptr)
typeof(*ptr) atomic_dec_fetch(ptr)
typeof(*ptr) atomic_add_fetch(ptr, val)
typeof(*ptr) atomic_sub_fetch(ptr, val)
typeof(*ptr) atomic_and_fetch(ptr, val)
typeof(*ptr) atomic_or_fetch(ptr, val)
typeof(*ptr) atomic_xor_fetch(ptr, val)
```

`qemu/atomic.h` also provides loads and stores that cannot be reordered with each other:

```
typeof(*ptr) atomic_mb_read(ptr)
void         atomic_mb_set(ptr, val)
```

However these do not provide sequential consistency and, in particular, they do not participate in the total ordering enforced by sequentially-consistent operations. For this reason they are deprecated. They should instead be replaced with any of the following (ordered from easiest to hardest):

- accesses inside a mutex or spinlock
- lightweight synchronization primitives such as `QemuEvent`
- RCU operations (`atomic_rcu_read`, `atomic_rcu_set`) when publishing or accessing a new version of a data structure
- other atomic accesses: `atomic_read` and `atomic_load_acquire` for loads, `atomic_set` and `atomic_store_release` for stores, `smp_mb` to forbid reordering subsequent loads before a store.

6.6.3 Weak atomic access and manual memory barriers

Compared to sequentially consistent atomic access, programming with weaker consistency models can be considerably more complicated. The only guarantees that you can rely upon in this case are:

- atomic accesses will not cause data races (and hence undefined behavior); ordinary accesses instead cause data races if they are concurrent with other accesses of which at least one is a write. In order to ensure this, the compiler will not optimize accesses out of existence, create unsolicited accesses, or perform other similar optimzations.
- acquire operations will appear to happen, with respect to the other components of the system, before all the LOAD or STORE operations specified afterwards.
- release operations will appear to happen, with respect to the other components of the system, after all the LOAD or STORE operations specified before.
- release operations will *synchronize with* acquire operations; see *Acquire/release pairing and the synchronizes-with relation* for a detailed explanation.

When using this model, variables are accessed with:

- `atomic_read()` and `atomic_set()`; these prevent the compiler from optimizing accesses out of existence and creating unsolicited accesses, but do not otherwise impose any ordering on loads and stores: both the compiler and the processor are free to reorder them.
- `atomic_load_acquire()`, which guarantees the LOAD to appear to happen, with respect to the other components of the system, before all the LOAD or STORE operations specified afterwards. Operations coming before `atomic_load_acquire()` can still be reordered after it.
- `atomic_store_release()`, which guarantees the STORE to appear to happen, with respect to the other components of the system, after all the LOAD or STORE operations specified before. Operations coming after `atomic_store_release()` can still be reordered before it.

Restrictions to the ordering of accesses can also be specified using the memory barrier macros: `smp_rmb()`, `smp_wmb()`, `smp_mb()`, `smp_mb_acquire()`, `smp_mb_release()`, `smp_read_barrier_depends()`.

Memory barriers control the order of references to shared memory. They come in six kinds:

- `smp_rmb()` guarantees that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system.

  In other words, `smp_rmb()` puts a partial ordering on loads, but is not required to have any effect on stores.
- `smp_wmb()` guarantees that all the STORE operations specified before the barrier will appear to happen before all the STORE operations specified after the barrier with respect to the other components of the system.

  In other words, `smp_wmb()` puts a partial ordering on stores, but is not required to have any effect on loads.
- `smp_mb_acquire()` guarantees that all the LOAD operations specified before the barrier will appear to happen before all the LOAD or STORE operations specified after the barrier with respect to the other components of the system.
- `smp_mb_release()` guarantees that all the STORE operations specified *after* the barrier will appear to happen after all the LOAD or STORE operations specified *before* the barrier with respect to the other components of the system.
- `smp_mb()` guarantees that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other components of the system.

smp_mb() puts a partial ordering on both loads and stores. It is stronger than both a read and a write memory barrier; it implies both smp_mb_acquire() and smp_mb_release(), but it also prevents STOREs coming before the barrier from overtaking LOADs coming after the barrier and vice versa.

- smp_read_barrier_depends() is a weaker kind of read barrier. On most processors, whenever two loads are performed such that the second depends on the result of the first (e.g., the first load retrieves the address to which the second load will be directed), the processor will guarantee that the first LOAD will appear to happen before the second with respect to the other components of the system. However, this is not always true—for example, it was not true on Alpha processors. Whenever this kind of access happens to shared memory (that is not protected by a lock), a read barrier is needed, and smp_read_barrier_depends() can be used instead of smp_rmb().

  Note that the first load really has to have a _data_ dependency and not a control dependency. If the address for the second load is dependent on the first load, but the dependency is through a conditional rather than actually loading the address itself, then it's a _control_ dependency and a full read barrier or better is required.

Memory barriers and atomic_load_acquire/atomic_store_release are mostly used when a data structure has one thread that is always a writer and one thread that is always a reader:

thread 1	thread 2

`atomic_store_release(&a, x);`
`atomic_store_release(&b, y);`
	`y = atomic_load_acquire(&b);`
	`x = atomic_load_acquire(&a);`

In this case, correctness is easy to check for using the "pairing" trick that is explained below.

Sometimes, a thread is accessing many variables that are otherwise unrelated to each other (for example because, apart from the current thread, exactly one other thread will read or write each of these variables). In this case, it is possible to "hoist" the barriers outside a loop. For example:

before	after

`n = 0;`
`for (i = 0; i < 10; i++)`
`  n += atomic_load_acquire(&a[i]);`
	`n = 0;`
	`for (i = 0; i < 10; i++)`
	`  n += atomic_read(&a[i]);`
	`smp_mb_acquire();`

`for (i = 0; i < 10; i++)`
`  atomic_store_release(&a[i],␣`
`↪false);`
	`smp_mb_release();`
	`for (i = 0; i < 10; i++)`
	`  atomic_set(&a[i], false);`

Splitting a loop can also be useful to reduce the number of barriers:

before	after

`n = 0;`
`for (i = 0; i < 10; i++) {`
`  atomic_store_release(&a[i],␣`
`↪false);`
`  smp_mb();`
`  n += atomic_read(&b[i]);`
`}`
	`smp_mb_release();`
	`for (i = 0; i < 10; i++)`
	`  atomic_set(&a[i], false);`
	`smb_mb();`
	`n = 0;`
	`for (i = 0; i < 10; i++)`
	`  n += atomic_read(&b[i]);`

In this case, a `smp_mb_release()` is also replaced with a (possibly cheaper, and clearer as well) `smp_wmb()`:

before	after

`for (i = 0; i < 10; i++) {`
`  atomic_store_release(&a[i],␣`
`↪false);`
`  atomic_store_release(&b[i],␣`
`↪false);`
`}`
	`smp_mb_release();`
	`for (i = 0; i < 10; i++)`
	`  atomic_set(&a[i], false);`
	`smb_wmb();`
	`for (i = 0; i < 10; i++)`
	`  atomic_set(&b[i], false);`

Acquire/release pairing and the *synchronizes-with* relation

Atomic operations other than `atomic_set()` and `atomic_read()` have either *acquire* or *release* semantics[1]. This has two effects:

- within a thread, they are ordered either before subsequent operations (for acquire) or after previous operations (for release).
- if a release operation in one thread *synchronizes with* an acquire operation in another thread, the ordering constraints propagates from the first to the second thread. That is, everything before the release operation in the first thread is guaranteed to *happen before* everything after the acquire operation in the second thread.

The concept of acquire and release semantics is not exclusive to atomic operations; almost all higher-level synchronization primitives also have acquire or release semantics. For example:

- `pthread_mutex_lock` has acquire semantics, `pthread_mutex_unlock` has release semantics and synchronizes with a `pthread_mutex_lock` for the same mutex.
- `pthread_cond_signal` and `pthread_cond_broadcast` have release semantics; `pthread_cond_wait` has both release semantics (synchronizing with `pthread_mutex_lock`) and acquire semantics (synchronizing with `pthread_mutex_unlock` and signaling of the condition variable).
- `pthread_create` has release semantics and synchronizes with the start of the new thread; `pthread_join` has acquire semantics and synchronizes with the exiting of the thread.
- `qemu_event_set` has release semantics, `qemu_event_wait` has acquire semantics.

For example, in the following example there are no atomic accesses, but still thread 2 is relying on the *synchronizes-with* relation between `pthread_exit` (release) and `pthread_join` (acquire):

[1] Read-modify-write operations can have both—acquire applies to the read part, and release to the write.

thread 1	thread 2

`*a = 1;`
`pthread_exit(a);`
	`pthread_join(thread1, &a);`
	`x = *a;`

Synchronization between threads basically descends from this pairing of a release operation and an acquire operation. Therefore, atomic operations other than `atomic_set()` and `atomic_read()` will almost always be paired with another operation of the opposite kind: an acquire operation will pair with a release operation and vice versa. This rule of thumb is extremely useful; in the case of QEMU, however, note that the other operation may actually be in a driver that runs in the guest!

`smp_read_barrier_depends()`, `smp_rmb()`, `smp_mb_acquire()`, `atomic_load_acquire()` and `atomic_rcu_read()` all count as acquire operations. `smp_wmb()`, `smp_mb_release()`, `atomic_store_release()` and `atomic_rcu_set()` all count as release operations. `smp_mb()` counts as both acquire and release, therefore it can pair with any other atomic operation. Here is an example:

thread 1	thread 2

`atomic_set(&a, 1);`
`smp_wmb();`
`atomic_set(&b, 2);`
	`x = atomic_read(&b);`
	`smp_rmb();`
	`y = atomic_read(&a);`

Note that a load-store pair only counts if the two operations access the same variable: that is, a store-release on a variable `x` *synchronizes with* a load-acquire on a variable `x`, while a release barrier synchronizes with any acquire operation. The following example shows correct synchronization:

thread 1	thread 2

`atomic_set(&a, 1);`
`atomic_store_release(&b, 2);`
	`x = atomic_load_acquire(&b);`
	`y = atomic_read(&a);`

Acquire and release semantics of higher-level primitives can also be relied upon for the purpose of establishing the *synchronizes with* relation.

Note that the "writing" thread is accessing the variables in the opposite order as the "reading" thread. This is expected: stores before a release operation will normally match the loads after the acquire operation, and vice versa. In fact, this happened already in the `pthread_exit`/`pthread_join` example above.

Finally, this more complex example has more than two accesses and data dependency barriers. It also does not use atomic accesses whenever there cannot be a data race:

thread 1	thread 2

`b[2] = 1;`
`smp_wmb();`
`x->i = 2;`
`smp_wmb();`
`atomic_set(&a, x);`
	`x = atomic_read(&a);`
	`smp_read_barrier_depends();`
	`y = x->i;`
	`smp_read_barrier_depends();`
	`z = b[y];`

6.6.4 Comparison with Linux kernel primitives

Here is a list of differences between Linux kernel atomic operations and memory barriers, and the equivalents in QEMU:

- atomic operations in Linux are always on a 32-bit int type and use a boxed `atomic_t` type; atomic operations in QEMU are polymorphic and use normal C types.
- Originally, `atomic_read` and `atomic_set` in Linux gave no guarantee at all. Linux 4.1 updated them to implement volatile semantics via `ACCESS_ONCE` (or the more recent `READ`/`WRITE_ONCE`).

  QEMU's `atomic_read` and `atomic_set` implement C11 atomic relaxed semantics if the compiler supports it, and volatile semantics otherwise. Both semantics prevent the compiler from doing certain transformations; the difference is that atomic accesses are guaranteed to be atomic, while volatile accesses aren't. Thus, in the volatile case we just cross our fingers hoping that the compiler will generate atomic accesses, since we assume the variables passed are machine-word sized and properly aligned.

  No barriers are implied by `atomic_read` and `atomic_set` in either Linux or QEMU.
- atomic read-modify-write operations in Linux are of three kinds:

`atomic_OP`	returns void
`atomic_OP_return`	returns new value of the variable
`atomic_fetch_OP`	returns the old value of the variable
`atomic_cmpxchg`	returns the old value of the variable

  In QEMU, the second kind is named `atomic_OP_fetch`.
- different atomic read-modify-write operations in Linux imply a different set of memory barriers; in QEMU, all of them enforce sequential consistency.
- in QEMU, `atomic_read()` and `atomic_set()` do not participate in the total ordering enforced by sequentially-consistent operations. This is because QEMU uses the C11 memory model. The following example is correct in Linux but not in QEMU:

Linux (correct)	QEMU (incorrect)

`a = atomic_fetch_add(&x, 2);`
`b = READ_ONCE(&y);`
	`a = atomic_fetch_add(&x, 2);`
	`b = atomic_read(&y);`

  because the read of `y` can be moved (by either the processor or the compiler) before the write of `x`.

  Fixing this requires an `smp_mb()` memory barrier between the write of `x` and the read of `y`. In the common case where only one thread writes `x`, it is also possible to write it like this:

QEMU (correct)

```
a = atomic_read(&x);
atomic_set(&x, a + 2);
smp_mb();
b = atomic_read(&y);
```

6.6.5 Sources

- `Documentation/memory-barriers.txt` from the Linux kernel

6.7 QEMU and the stable process

6.7.1 QEMU stable releases

QEMU stable releases are based upon the last released QEMU version and marked by an additional version number, e.g. 2.10.1. Occasionally, a four-number version is released, if a single urgent fix needs to go on top.

Usually, stable releases are only provided for the last major QEMU release. For example, when QEMU 2.11.0 is released, 2.11.x or 2.11.x.y stable releases are produced only until QEMU 2.12.0 is released, at which point the stable process moves to producing 2.12.x/2.12.x.y releases.

6.7.2 What should go into a stable release?

Generally, the following patches are considered stable material:

- Patches that fix severe issues, like fixes for CVEs
- Patches that fix regressions

If you think the patch would be important for users of the current release (or for a distribution picking fixes), it is usually a good candidate for stable.

6.7.3 How to get a patch into QEMU stable

There are various ways to get a patch into stable:

- Preferred: Make sure that the stable maintainers are on copy when you send the patch by adding

  ```
  Cc: qemu-stable@nongnu.org
  ```

  to the patch description. By default, this will send a copy of the patch to `qemu-stable@nongnu.org` if you use git send-email, which is where patches that are stable candidates are tracked by the maintainers.
- You can also reply to a patch and put `qemu-stable@nongnu.org` on copy directly in your mail client if you think a previously submitted patch should be considered for a stable release.
- If a maintainer judges the patch appropriate for stable later on (or you notify them), they will add the same line to the patch, meaning that the stable maintainers will be on copy on the maintainer's pull request.

- If you judge an already merged patch suitable for stable, send a mail (preferably as a reply to the most recent patch submission) to `qemu-stable@nongnu.org` along with `qemu-devel@nongnu.org` and appropriate other people (like the patch author or the relevant maintainer) on copy.

6.7.4 Stable release process

When the stable maintainers prepare a new stable release, they will prepare a git branch with a release candidate and send the patches out to `qemu-devel@nongnu.org` for review. If any of your patches are included, please verify that they look fine, especially if the maintainer had to tweak the patch as part of back-porting things across branches. You may also nominate other patches that you think are suitable for inclusion. After review is complete (may involve more release candidates), a new stable release is made available.

6.8 Testing in QEMU

This document describes the testing infrastructure in QEMU.

6.8.1 Testing with "make check"

The "make check" testing family includes most of the C based tests in QEMU. For a quick help, run `make check-help` from the source tree.

The usual way to run these tests is:

```
make check
```

which includes QAPI schema tests, unit tests, QTests and some iotests. Different sub-types of "make check" tests will be explained below.

Before running tests, it is best to build QEMU programs first. Some tests expect the executables to exist and will fail with obscure messages if they cannot find them.

Unit tests

Unit tests, which can be invoked with `make check-unit`, are simple C tests that typically link to individual QEMU object files and exercise them by calling exported functions.

If you are writing new code in QEMU, consider adding a unit test, especially for utility modules that are relatively stateless or have few dependencies. To add a new unit test:

1. Create a new source file. For example, `tests/foo-test.c`.
2. Write the test. Normally you would include the header file which exports the module API, then verify the interface behaves as expected from your test. The test code should be organized with the glib testing framework. Copying and modifying an existing test is usually a good idea.
3. Add the test to `tests/Makefile.include`. First, name the unit test program and add it to `$(check-unit-y)`; then add a rule to build the executable. For example:

```
check-unit-y += tests/foo-test$(EXESUF)
tests/foo-test$(EXESUF): tests/foo-test.o $(test-util-obj-y)
...
```

Since unit tests don't require environment variables, the simplest way to debug a unit test failure is often directly invoking it or even running it under `gdb`. However there can still be differences in behavior between `make` invocations and your manual run, due to `$MALLOC_PERTURB_` environment variable (which affects memory reclamation and catches invalid pointers better) and gtester options. If necessary, you can run

```
make check-unit V=1
```

and copy the actual command line which executes the unit test, then run it from the command line.

QTest

QTest is a device emulation testing framework. It can be very useful to test device models; it could also control certain aspects of QEMU (such as virtual clock stepping), with a special purpose "qtest" protocol. Refer to the documentation in `qtest.c` for more details of the protocol.

QTest cases can be executed with

```
make check-qtest
```

The QTest library is implemented by `tests/qtest/libqtest.c` and the API is defined in `tests/qtest/libqtest.h`.

Consider adding a new QTest case when you are introducing a new virtual hardware, or extending one if you are adding functionalities to an existing virtual device.

On top of libqtest, a higher level library, `libqos`, was created to encapsulate common tasks of device drivers, such as memory management and communicating with system buses or devices. Many virtual device tests use libqos instead of directly calling into libqtest.

Steps to add a new QTest case are:

1. Create a new source file for the test. (More than one file can be added as necessary.) For example, `tests/qtest/foo-test.c`.
2. Write the test code with the glib and libqtest/libqos API. See also existing tests and the library headers for reference.
3. Register the new test in `tests/qtest/Makefile.include`. Add the test executable name to an appropriate `check-qtest-*-y` variable. For example:

   ```
   check-qtest-generic-y = tests/qtest/foo-test$(EXESUF)
   ```

4. Add object dependencies of the executable in the Makefile, including the test source file(s) and other interesting objects. For example:

   ```
   tests/qtest/foo-test$(EXESUF): tests/qtest/foo-test.o $(libqos-obj-y)
   ```

Debugging a QTest failure is slightly harder than the unit test because the tests look up QEMU program names in the environment variables, such as `QTEST_QEMU_BINARY` and `QTEST_QEMU_IMG`, and also because it is not easy to attach gdb to the QEMU process spawned from the test. But manual invoking and using gdb on the test is still simple to do: find out the actual command from the output of

```
make check-qtest V=1
```

which you can run manually.

QAPI schema tests

The QAPI schema tests validate the QAPI parser used by QMP, by feeding predefined input to the parser and comparing the result with the reference output.

The input/output data is managed under the `tests/qapi-schema` directory. Each test case includes four files that have a common base name:

- `${casename}.json` - the file contains the JSON input for feeding the parser
- `${casename}.out` - the file contains the expected stdout from the parser
- `${casename}.err` - the file contains the expected stderr from the parser
- `${casename}.exit` - the expected error code

Consider adding a new QAPI schema test when you are making a change on the QAPI parser (either fixing a bug or extending/modifying the syntax). To do this:

1. Add four files for the new case as explained above. For example:

   `$EDITOR tests/qapi-schema/foo.{json,out,err,exit}.`

2. Add the new test in `tests/Makefile.include`. For example:

   `qapi-schema += foo.json`

check-block

`make check-block` runs a subset of the block layer iotests (the tests that are in the “auto” group in `tests/qemu-iotests/group`). See the “QEMU iotests” section below for more information.

GCC gcov support

`gcov` is a GCC tool to analyze the testing coverage by instrumenting the tested code. To use it, configure QEMU with `--enable-gcov` option and build. Then run `make check` as usual.

If you want to gather coverage information on a single test the `make clean-gcda` target can be used to delete any existing coverage information before running a single test.

You can generate a HTML coverage report by executing `make coverage-html` which will create `meson-logs/coveragereport/index.html`.

Further analysis can be conducted by running the `gcov` command directly on the various .gcda output files. Please read the `gcov` documentation for more information.

6.8.2 QEMU iotests

QEMU iotests, under the directory `tests/qemu-iotests`, is the testing framework widely used to test block layer related features. It is higher level than “make check” tests and 99% of the code is written in bash or Python scripts. The testing success criteria is golden output comparison, and the test files are named with numbers.

To run iotests, make sure QEMU is built successfully, then switch to the `tests/qemu-iotests` directory under the build directory, and run `./check` with desired arguments from there.

By default, “raw” format and “file” protocol is used; all tests will be executed, except the unsupported ones. You can override the format and protocol with arguments:

```
# test with qcow2 format
./check -qcow2
# or test a different protocol
./check -nbd
```

It's also possible to list test numbers explicitly:

```
# run selected cases with qcow2 format
./check -qcow2 001 030 153
```

Cache mode can be selected with the "-c" option, which may help reveal bugs that are specific to certain cache mode.

More options are supported by the `./check` script, run `./check -h` for help.

Writing a new test case

Consider writing a tests case when you are making any changes to the block layer. An iotest case is usually the choice for that. There are already many test cases, so it is possible that extending one of them may achieve the goal and save the boilerplate to create one. (Unfortunately, there isn't a 100% reliable way to find a related one out of hundreds of tests. One approach is using `git grep`.)

Usually an iotest case consists of two files. One is an executable that produces output to stdout and stderr, the other is the expected reference output. They are given the same number in file names. E.g. Test script `055` and reference output `055.out`.

In rare cases, when outputs differ between cache mode `none` and others, a `.out.nocache` file is added. In other cases, when outputs differ between image formats, more than one `.out` files are created ending with the respective format names, e.g. `178.out.qcow2` and `178.out.raw`.

There isn't a hard rule about how to write a test script, but a new test is usually a (copy and) modification of an existing case. There are a few commonly used ways to create a test:

- A Bash script. It will make use of several environmental variables related to the testing procedure, and could source a group of `common.*` libraries for some common helper routines.
- A Python unittest script. Import `iotests` and create a subclass of `iotests.QMPTestCase`, then call `iotests.main` method. The downside of this approach is that the output is too scarce, and the script is considered harder to debug.
- A simple Python script without using unittest module. This could also import `iotests` for launching QEMU and utilities etc, but it doesn't inherit from `iotests.QMPTestCase` therefore doesn't use the Python unittest execution. This is a combination of 1 and 2.

Pick the language per your preference since both Bash and Python have comparable library support for invoking and interacting with QEMU programs. If you opt for Python, it is strongly recommended to write Python 3 compatible code.

Both Python and Bash frameworks in iotests provide helpers to manage test images. They can be used to create and clean up images under the test directory. If no I/O or any protocol specific feature is needed, it is often more convenient to use the pseudo block driver, `null-co://`, as the test image, which doesn't require image creation or cleaning up. Avoid system-wide devices or files whenever possible, such as `/dev/null` or `/dev/zero`. Otherwise, image locking implications have to be considered. For example, another application on the host may have locked the file, possibly leading to a test failure. If using such devices are explicitly desired, consider adding `locking=off` option to disable image locking.

6.8.3 Docker based tests

Introduction

The Docker testing framework in QEMU utilizes public Docker images to build and test QEMU in predefined and widely accessible Linux environments. This makes it possible to expand the test coverage across distros, toolchain flavors and library versions.

Prerequisites

Install "docker" with the system package manager and start the Docker service on your development machine, then make sure you have the privilege to run Docker commands. Typically it means setting up passwordless `sudo docker` command or login as root. For example:

```
$ sudo yum install docker
$ # or `apt-get install docker` for Ubuntu, etc.
$ sudo systemctl start docker
$ sudo docker ps
```

The last command should print an empty table, to verify the system is ready.

An alternative method to set up permissions is by adding the current user to "docker" group and making the docker daemon socket file (by default `/var/run/docker.sock`) accessible to the group:

```
$ sudo groupadd docker
$ sudo usermod $USER -a -G docker
$ sudo chown :docker /var/run/docker.sock
```

Note that any one of above configurations makes it possible for the user to exploit the whole host with Docker bind mounting or other privileged operations. So only do it on development machines.

Quickstart

From source tree, type `make docker` to see the help. Testing can be started without configuring or building QEMU (`configure` and `make` are done in the container, with parameters defined by the make target):

```
make docker-test-build@min-glib
```

This will create a container instance using the `min-glib` image (the image is downloaded and initialized automatically), in which the `test-build` job is executed.

Images

Along with many other images, the `min-glib` image is defined in a Dockerfile in `tests/docker/dockerfiles/`, called `min-glib.docker`. `make docker` command will list all the available images.

To add a new image, simply create a new `.docker` file under the `tests/docker/dockerfiles/` directory.

A `.pre` script can be added beside the `.docker` file, which will be executed before building the image under the build context directory. This is mainly used to do necessary host side setup. One such setup is `binfmt_misc`, for example, to make qemu-user powered cross build containers work.

Tests

Different tests are added to cover various configurations to build and test QEMU. Docker tests are the executables under `tests/docker` named `test-*`. They are typically shell scripts and are built on top of a shell library, `tests/docker/common.rc`, which provides helpers to find the QEMU source and build it.

The full list of tests is printed in the `make docker` help.

Tools

There are executables that are created to run in a specific Docker environment. This makes it easy to write scripts that have heavy or special dependencies, but are still very easy to use.

Currently the only tool is `travis`, which mimics the Travis-CI tests in a container. It runs in the `travis` image:

```
make docker-travis@travis
```

Debugging a Docker test failure

When CI tasks, maintainers or yourself report a Docker test failure, follow the below steps to debug it:

1. Locally reproduce the failure with the reported command line. E.g. run `make docker-test-mingw@fedora J=8`.
2. Add “V=1” to the command line, try again, to see the verbose output.
3. Further add “DEBUG=1” to the command line. This will pause in a shell prompt in the container right before testing starts. You could either manually build QEMU and run tests from there, or press Ctrl-D to let the Docker testing continue.
4. If you press Ctrl-D, the same building and testing procedure will begin, and will hopefully run into the error again. After that, you will be dropped to the prompt for debug.

Options

Various options can be used to affect how Docker tests are done. The full list is in the `make docker` help text. The frequently used ones are:

- `V=1`: the same as in top level `make`. It will be propagated to the container and enable verbose output.
- `J=$N`: the number of parallel tasks in make commands in the container, similar to the `-j $N` option in top level `make`. (The `-j` option in top level `make` will not be propagated into the container.)
- `DEBUG=1`: enables debug. See the previous “Debugging a Docker test failure” section.

6.8.4 Thread Sanitizer

Thread Sanitizer (TSan) is a tool which can detect data races. QEMU supports building and testing with this tool.

For more information on TSan:

https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

Thread Sanitizer in Docker

TSan is currently supported in the ubuntu2004 docker.

The test-tsan test will build using TSan and then run make check.

```
make docker-test-tsan@ubuntu2004
```

TSan warnings under docker are placed in files located at build/tsan/.

We recommend using DEBUG=1 to allow launching the test from inside the docker, and to allow review of the warnings generated by TSan.

Building and Testing with TSan

It is possible to build and test with TSan, with a few additional steps. These steps are normally done automatically in the docker.

There is a one time patch needed in clang-9 or clang-10 at this time:

```
sed -i 's/^const/static const/g' \
    /usr/lib/llvm-10/lib/clang/10.0.0/include/sanitizer/tsan_interface.h
```

To configure the build for TSan:

```
../configure --enable-tsan --cc=clang-10 --cxx=clang++-10 \
             --disable-werror --extra-cflags="-O0"
```

The runtime behavior of TSAN is controlled by the TSAN_OPTIONS environment variable.

More information on the TSAN_OPTIONS can be found here:

https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags

For example:

```
export TSAN_OPTIONS=suppressions=<path to qemu>/tests/tsan/suppressions.tsan \
                    detect_deadlocks=false history_size=7 exitcode=0 \
                    log_path=<build path>/tsan/tsan_warning
```

The above exitcode=0 has TSan continue without error if any warnings are found. This allows for running the test and then checking the warnings afterwards. If you want TSan to stop and exit with error on warnings, use exitcode=66.

TSan Suppressions

Keep in mind that for any data race warning, although there might be a data race detected by TSan, there might be no actual bug here. TSan provides several different mechanisms for suppressing warnings. In general it is recommended to fix the code if possible to eliminate the data race rather than suppress the warning.

A few important files for suppressing warnings are:

tests/tsan/suppressions.tsan - Has TSan warnings we wish to suppress at runtime. The comment on each supression will typically indicate why we are suppressing it. More information on the file format can be found here:

https://github.com/google/sanitizers/wiki/ThreadSanitizerSuppressions

tests/tsan/blacklist.tsan - Has TSan warnings we wish to disable at compile time for test or debug. Add flags to configure to enable:

"–extra-cflags=-fsanitize-blacklist=<src path>/tests/tsan/blacklist.tsan"

More information on the file format can be found here under "Blacklist Format":

https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags

TSan Annotations

include/qemu/tsan.h defines annotations. See this file for more descriptions of the annotations themselves. Annotations can be used to suppress TSan warnings or give TSan more information so that it can detect proper relationships between accesses of data.

Annotation examples can be found here:

https://github.com/llvm/llvm-project/tree/master/compiler-rt/test/tsan/

Good files to start with are: annotate_happens_before.cpp and ignore_race.cpp

The full set of annotations can be found here:

https://github.com/llvm/llvm-project/blob/master/compiler-rt/lib/tsan/rtl/tsan_interface_ann.cpp

6.8.5 VM testing

This test suite contains scripts that bootstrap various guest images that have necessary packages to build QEMU. The basic usage is documented in `Makefile` help which is displayed with `make vm-help`.

Quickstart

Run `make vm-help` to list available make targets. Invoke a specific make command to run build test in an image. For example, `make vm-build-freebsd` will build the source tree in the FreeBSD image. The command can be executed from either the source tree or the build dir; if the former, `./configure` is not needed. The command will then generate the test image in `./tests/vm/` under the working directory.

Note: images created by the scripts accept a well-known RSA key pair for SSH access, so they SHOULD NOT be exposed to external interfaces if you are concerned about attackers taking control of the guest and potentially exploiting a QEMU security bug to compromise the host.

QEMU binaries

By default, qemu-system-x86_64 is searched in $PATH to run the guest. If there isn't one, or if it is older than 2.10, the test won't work. In this case, provide the QEMU binary in env var: `QEMU=/path/to/qemu-2.10+`.

Likewise the path to qemu-img can be set in QEMU_IMG environment variable.

Make jobs

The `-j$X` option in the make command line is not propagated into the VM, specify `J=$X` to control the make jobs in the guest.

Debugging

Add `DEBUG=1` and/or `V=1` to the make command to allow interactive debugging and verbose output. If this is not enough, see the next section. `V=1` will be propagated down into the make jobs in the guest.

Manual invocation

Each guest script is an executable script with the same command line options. For example to work with the netbsd guest, use `$QEMU_SRC/tests/vm/netbsd`:

```
$ cd $QEMU_SRC/tests/vm

# To bootstrap the image
$ ./netbsd --build-image --image /var/tmp/netbsd.img
<...>

# To run an arbitrary command in guest (the output will not be echoed unless
# --debug is added)
$ ./netbsd --debug --image /var/tmp/netbsd.img uname -a

# To build QEMU in guest
$ ./netbsd --debug --image /var/tmp/netbsd.img --build-qemu $QEMU_SRC

# To get to an interactive shell
$ ./netbsd --interactive --image /var/tmp/netbsd.img sh
```

Adding new guests

Please look at existing guest scripts for how to add new guests.

Most importantly, create a subclass of BaseVM and implement `build_image()` method and define `BUILD_SCRIPT`, then finally call `basevm.main()` from the script's `main()`.

- Usually in `build_image()`, a template image is downloaded from a predefined URL. `BaseVM._download_with_cache()` takes care of the cache and the checksum, so consider using it.
- Once the image is downloaded, users, SSH server and QEMU build deps should be set up:
  - Root password set to `BaseVM.ROOT_PASS`
  - User `BaseVM.GUEST_USER` is created, and password set to `BaseVM.GUEST_PASS`
  - SSH service is enabled and started on boot, `$QEMU_SRC/tests/keys/id_rsa.pub` is added to ssh's `authorized_keys` file of both root and the normal user
  - DHCP client service is enabled and started on boot, so that it can automatically configure the virtio-net-pci NIC and communicate with QEMU user net (10.0.2.2)
  - Necessary packages are installed to untar the source tarball and build QEMU
- Write a proper `BUILD_SCRIPT` template, which should be a shell script that untars a raw virtio-blk block device, which is the tarball data blob of the QEMU source tree, then configure/build it. Running "make check" is also recommended.

6.8.6 Image fuzzer testing

An image fuzzer was added to exercise format drivers. Currently only qcow2 is supported. To start the fuzzer, run

```
tests/image-fuzzer/runner.py -c '[["qemu-img", "info", "$test_img"]]' /tmp/test qcow2
```

Alternatively, some command different from "qemu-img info" can be tested, by changing the `-c` option.

6.8.7 Acceptance tests using the Avocado Framework

The `tests/acceptance` directory hosts functional tests, also known as acceptance level tests. They're usually higher level tests, and may interact with external resources and with various guest operating systems.

These tests are written using the Avocado Testing Framework (which must be installed separately) in conjunction with a the `avocado_qemu.Test` class, implemented at `tests/acceptance/avocado_qemu`.

Tests based on `avocado_qemu.Test` can easily:

- Customize the command line arguments given to the convenience `self.vm` attribute (a QEMUMachine instance)
- Interact with the QEMU monitor, send QMP commands and check their results
- Interact with the guest OS, using the convenience console device (which may be useful to assert the effectiveness and correctness of command line arguments or QMP commands)
- Interact with external data files that accompany the test itself (see `self.get_data()`)
- Download (and cache) remote data files, such as firmware and kernel images
- Have access to a library of guest OS images (by means of the `avocado.utils.vmimage` library)
- Make use of various other test related utilities available at the test class itself and at the utility library:
  - http://avocado-framework.readthedocs.io/en/latest/api/test/avocado.html#avocado.Test
  - http://avocado-framework.readthedocs.io/en/latest/api/utils/avocado.utils.html

Running tests

You can run the acceptance tests simply by executing:

```
make check-acceptance
```

This involves the automatic creation of Python virtual environment within the build tree (at `tests/venv`) which will have all the right dependencies, and will save tests results also within the build tree (at `tests/results`).

Note: the build environment must be using a Python 3 stack, and have the `venv` and `pip` packages installed. If necessary, make sure `configure` is called with `--python=` and that those modules are available. On Debian and Ubuntu based systems, depending on the specific version, they may be on packages named `python3-venv` and `python3-pip`.

The scripts installed inside the virtual environment may be used without an "activation". For instance, the Avocado test runner may be invoked by running:

```
tests/venv/bin/avocado run $OPTION1 $OPTION2 tests/acceptance/
```

Manual Installation

To manually install Avocado and its dependencies, run:

```
pip install --user avocado-framework
```

Alternatively, follow the instructions on this link:

http://avocado-framework.readthedocs.io/en/latest/GetStartedGuide.html#installing-avocado

Overview

The `tests/acceptance/avocado_qemu` directory provides the `avocado_qemu` Python module, containing the `avocado_qemu.Test` class. Here's a simple usage example:

```
from avocado_qemu import Test


class Version(Test):
    """
    :avocado: tags=quick
    """
    def test_qmp_human_info_version(self):
        self.vm.launch()
        res = self.vm.command('human-monitor-command',
                              command_line='info version')
        self.assertRegexpMatches(res, r'^(\d+\.\d+\.\d)')
```

To execute your test, run:

```
avocado run version.py
```

Tests may be classified according to a convention by using docstring directives such as `:avocado: tags=TAG1,TAG2`. To run all tests in the current directory, tagged as "quick", run:

```
avocado run -t quick .
```

The `avocado_qemu.Test` base test class

The `avocado_qemu.Test` class has a number of characteristics that are worth being mentioned right away.

First of all, it attempts to give each test a ready to use QEMUMachine instance, available at `self.vm`. Because many tests will tweak the QEMU command line, launching the QEMUMachine (by using `self.vm.launch()`) is left to the test writer.

The base test class has also support for tests with more than one QEMUMachine. The way to get machines is through the `self.get_vm()` method which will return a QEMUMachine instance. The `self.get_vm()` method accepts arguments that will be passed to the QEMUMachine creation and also an optional *name* attribute so you can identify a specific machine and get it more than once through the tests methods. A simple and hypothetical example follows:

```
from avocado_qemu import Test


class MultipleMachines(Test):
    """
    :avocado: enable
    """
    def test_multiple_machines(self):
        first_machine = self.get_vm()
        second_machine = self.get_vm()
        self.get_vm(name='third_machine').launch()

        first_machine.launch()
        second_machine.launch()

        first_res = first_machine.command(
```

(continues on next page)

(continued from previous page)

```
            'human-monitor-command',
            command_line='info version')

        second_res = second_machine.command(
            'human-monitor-command',
            command_line='info version')

        third_res = self.get_vm(name='third_machine').command(
            'human-monitor-command',
            command_line='info version')

        self.assertEquals(first_res, second_res, third_res)
```

At test "tear down", `avocado_qemu.Test` handles all the QEMUMachines shutdown.

QEMUMachine

The QEMUMachine API is already widely used in the Python iotests, device-crash-test and other Python scripts. It's a wrapper around the execution of a QEMU binary, giving its users:

- the ability to set command line arguments to be given to the QEMU binary
- a ready to use QMP connection and interface, which can be used to send commands and inspect its results, as well as asynchronous events
- convenience methods to set commonly used command line arguments in a more succinct and intuitive way

QEMU binary selection

The QEMU binary used for the `self.vm` QEMUMachine instance will primarily depend on the value of the `qemu_bin` parameter. If it's not explicitly set, its default value will be the result of a dynamic probe in the same source tree. A suitable binary will be one that targets the architecture matching host machine.

Based on this description, test writers will usually rely on one of the following approaches:

1) Set `qemu_bin`, and use the given binary
2) Do not set `qemu_bin`, and use a QEMU binary named like "qemu-system-${arch}", either in the current working directory, or in the current source tree.

The resulting `qemu_bin` value will be preserved in the `avocado_qemu.Test` as an attribute with the same name.

Attribute reference

Besides the attributes and methods that are part of the base `avocado.Test` class, the following attributes are available on any `avocado_qemu.Test` instance.

vm

A QEMUMachine instance, initially configured according to the given `qemu_bin` parameter.

arch

The architecture can be used on different levels of the stack, e.g. by the framework or by the test itself. At the framework level, it will currently influence the selection of a QEMU binary (when one is not explicitly given).

Tests are also free to use this attribute value, for their own needs. A test may, for instance, use the same value when selecting the architecture of a kernel or disk image to boot a VM with.

The `arch` attribute will be set to the test parameter of the same name. If one is not given explicitly, it will either be set to `None`, or, if the test is tagged with one (and only one) `:avocado: tags=arch:VALUE` tag, it will be set to `VALUE`.

machine

The machine type that will be set to all QEMUMachine instances created by the test.

The `machine` attribute will be set to the test parameter of the same name. If one is not given explicitly, it will either be set to `None`, or, if the test is tagged with one (and only one) `:avocado: tags=machine:VALUE` tag, it will be set to `VALUE`.

qemu_bin

The preserved value of the `qemu_bin` parameter or the result of the dynamic probe for a QEMU binary in the current working directory or source tree.

Parameter reference

To understand how Avocado parameters are accessed by tests, and how they can be passed to tests, please refer to:

```
http://avocado-framework.readthedocs.io/en/latest/WritingTests.html#accessing-test-
↪parameters
```

Parameter values can be easily seen in the log files, and will look like the following:

```
PARAMS (key=qemu_bin, path=*, default=./qemu-system-x86_64) => './qemu-system-x86_64
```

arch

The architecture that will influence the selection of a QEMU binary (when one is not explicitly given).

Tests are also free to use this parameter value, for their own needs. A test may, for instance, use the same value when selecting the architecture of a kernel or disk image to boot a VM with.

This parameter has a direct relation with the `arch` attribute. If not given, it will default to None.

machine

The machine type that will be set to all QEMUMachine instances created by the test.

qemu_bin

The exact QEMU binary to be used on QEMUMachine.

Uninstalling Avocado

If you've followed the manual installation instructions above, you can easily uninstall Avocado. Start by listing the packages you have installed:

```
pip list --user
```

And remove any package you want with:

```
pip uninstall <package_name>
```

If you've used `make check-acceptance`, the Python virtual environment where Avocado is installed will be cleaned up as part of `make check-clean`.

6.8.8 Testing with "make check-tcg"

The check-tcg tests are intended for simple smoke tests of both linux-user and softmmu TCG functionality. However to build test programs for guest targets you need to have cross compilers available. If your distribution supports cross compilers you can do something as simple as:

```
apt install gcc-aarch64-linux-gnu
```

The configure script will automatically pick up their presence. Sometimes compilers have slightly odd names so the availability of them can be prompted by passing in the appropriate configure option for the architecture in question, for example:

```
$(configure) --cross-cc-aarch64=aarch64-cc
```

There is also a `--cross-cc-flags-ARCH` flag in case additional compiler flags are needed to build for a given target.

If you have the ability to run containers as the user you can also take advantage of the build systems "Docker" support. It will then use containers to build any test case for an enabled guest where there is no system compiler available. See :ref: *_docker-ref* for details.

Running subset of tests

You can build the tests for one architecture:

```
make build-tcg-tests-$TARGET
```

And run with:

```
make run-tcg-tests-$TARGET
```

Adding `V=1` to the invocation will show the details of how to invoke QEMU for the test which is useful for debugging tests.

TCG test dependencies

The TCG tests are deliberately very light on dependencies and are either totally bare with minimal gcc lib support (for softmmu tests) or just glibc (for linux-user tests). This is because getting a cross compiler to work with additional libraries can be challenging.

Other TCG Tests

There are a number of out-of-tree test suites that are used for more extensive testing of processor features.

KVM Unit Tests

The KVM unit tests are designed to run as a Guest OS under KVM but there is no reason why they can't exercise the TCG as well. It provides a minimal OS kernel with hooks for enabling the MMU as well as reporting test results via a special device:

```
https://git.kernel.org/pub/scm/virt/kvm/kvm-unit-tests.git
```

Linux Test Project

The LTP is focused on exercising the syscall interface of a Linux kernel. It checks that syscalls behave as documented and strives to exercise as many corner cases as possible. It is a useful test suite to run to exercise QEMU's linux-user code:

```
https://linux-test-project.github.io/
```

6.9 Decodetree Specification

A *decodetree* is built from instruction *patterns*. A pattern may represent a single architectural instruction or a group of same, depending on what is convenient for further processing.

Each pattern has both *fixedbits* and *fixedmask*, the combination of which describes the condition under which the pattern is matched:

```
(insn & fixedmask) == fixedbits
```

Each pattern may have *fields*, which are extracted from the insn and passed along to the translator. Examples of such are registers, immediates, and sub-opcodes.

In support of patterns, one may declare *fields*, *argument sets*, and *formats*, each of which may be re-used to simplify further definitions.

6.9.1 Fields

Syntax:

```
field_def     := '%' identifier ( unnamed_field )* ( !function=identifier )?
unnamed_field := number ':' ( 's' ) number
```

For *unnamed_field*, the first number is the least-significant bit position of the field and the second number is the length of the field. If the 's' is present, the field is considered signed. If multiple `unnamed_fields` are present, they are concatenated. In this way one can define disjoint fields.

If `!function` is specified, the concatenated result is passed through the named function, taking and returning an integral value.

One may use `!function` with zero `unnamed_fields`. This case is called a *parameter*, and the named function is only passed the `DisasContext` and returns an integral value extracted from there.

A field with no `unnamed_fields` and no `!function` is in error.

FIXME: the fields of the structure into which this result will be stored is restricted to `int`. Which means that we cannot expand 64-bit items.

Field examples:

Input	Generated code
%disp 0:s16	sextract(i, 0, 16)
%imm9 16:6 10:3	extract(i, 16, 6) << 3 | extract(i, 10, 3)
%disp12 0:s1 1:1 2:10	**sextract(i, 0, 1) << 11 |** extract(i, 1, 1) << 10 | extract(i, 2, 10)
%shimm8 5:s8 13:1 !function=expand_shimm8	**expand_shimm8(sextract(i, 5, 8) << 1 |** extract(i, 13, 1))

6.9.2 Argument Sets

Syntax:

```
args_def    := '&' identifier ( args_elt )+ ( !extern )?
args_elt    := identifier
```

Each *args_elt* defines an argument within the argument set. Each argument set will be rendered as a C structure "arg_$name" with each of the fields being one of the member arguments.

If `!extern` is specified, the backing structure is assumed to have been already declared, typically via a second decoder.

Argument sets are useful when one wants to define helper functions for the translator functions that can perform operations on a common set of arguments. This can ensure, for instance, that the `AND` pattern and the `OR` pattern put their operands into the same named structure, so that a common `gen_logic_insn` may be able to handle the operations common between the two.

Argument set examples:

```
&reg3       ra rb rc
&loadstore  reg base offset
```

6.9.3 Formats

Syntax:

```
fmt_def      := '@' identifier ( fmt_elt )+
fmt_elt      := fixedbit_elt | field_elt | field_ref | args_ref
fixedbit_elt := [01.-]+
field_elt    := identifier ':' 's'? number
field_ref    := '%' identifier | identifier '=' '%' identifier
args_ref     := '&' identifier
```

Defining a format is a handy way to avoid replicating groups of fields across many instruction patterns.

A *fixedbit_elt* describes a contiguous sequence of bits that must be 1, 0, or don't care. The difference between '.' and '-' is that '.' means that the bit will be covered with a field or a final 0 or 1 from the pattern, and '-' means that the bit is really ignored by the cpu and will not be specified.

A *field_elt* describes a simple field only given a width; the position of the field is implied by its position with respect to other *fixedbit_elt* and *field_elt*.

If any *fixedbit_elt* or *field_elt* appear, then all bits must be defined. Padding with a *fixedbit_elt* of all '.' is an easy way to accomplish that.

A *field_ref* incorporates a field by reference. This is the only way to add a complex field to a format. A field may be renamed in the process via assignment to another identifier. This is intended to allow the same argument set be used with disjoint named fields.

A single *args_ref* may specify an argument set to use for the format. The set of fields in the format must be a subset of the arguments in the argument set. If an argument set is not specified, one will be inferred from the set of fields.

It is recommended, but not required, that all *field_ref* and *args_ref* appear at the end of the line, not interleaving with *fixedbit_elf* or *field_elt*.

Format examples:

```
@opr    ...... ra:5 rb:5 ... 0 ....... rc:5
@opi    ...... ra:5 lit:8    1 ....... rc:5
```

6.9.4 Patterns

Syntax:

```
pat_def      := identifier ( pat_elt )+
pat_elt      := fixedbit_elt | field_elt | field_ref | args_ref | fmt_ref | const_elt
fmt_ref      := '@' identifier
const_elt    := identifier '=' number
```

The *fixedbit_elt* and *field_elt* specifiers are unchanged from formats. A pattern that does not specify a named format will have one inferred from a referenced argument set (if present) and the set of fields.

A *const_elt* allows a argument to be set to a constant value. This may come in handy when fields overlap between patterns and one has to include the values in the *fixedbit_elt* instead.

The decoder will call a translator function for each pattern matched.

Pattern examples:

```
addl_r   010000 ..... ..... .... 0000000 ..... @opr
addl_i   010000 ..... ..... .... 0000000 ..... @opi
```

which will, in part, invoke:

```
trans_addl_r(ctx, &arg_opr, insn)
```

and:

```
trans_addl_i(ctx, &arg_opi, insn)
```

6.9.5 Pattern Groups

Syntax:

```
group            := overlap_group | no_overlap_group
overlap_group    := '{' ( pat_def | group )+ '}'
no_overlap_group := '[' ( pat_def | group )+ ']'
```

A *group* begins with a lone open-brace or open-bracket, with all subsequent lines indented two spaces, and ending with a lone close-brace or close-bracket. Groups may be nested, increasing the required indentation of the lines within the nested group to two spaces per nesting level.

Patterns within overlap groups are allowed to overlap. Conflicts are resolved by selecting the patterns in order. If all of the fixedbits for a pattern match, its translate function will be called. If the translate function returns false, then subsequent patterns within the group will be matched.

Patterns within no-overlap groups are not allowed to overlap, just the same as ungrouped patterns. Thus no-overlap groups are intended to be nested inside overlap groups.

The following example from PA-RISC shows specialization of the *or* instruction:

```
{
  {
    nop   000010 ----- ----- 0000 001001 0 00000
    copy  000010 00000 r1:5  0000 001001 0 rt:5
  }
  or      000010 rt2:5 r1:5  cf:4 001001 0 rt:5
}
```

When the *cf* field is zero, the instruction has no side effects, and may be specialized. When the *rt* field is zero, the output is discarded and so the instruction has no effect. When the *rt2* field is zero, the operation is `reg[r1] | 0` and so encodes the canonical register copy operation.

The output from the generator might look like:

```
switch (insn & 0xfc000fe0) {
case 0x08000240:
  /* 000010.. ........ ....0010 010..... */
  if ((insn & 0x0000f000) == 0x00000000) {
      /* 000010.. ........ 00000010 010..... */
      if ((insn & 0x0000001f) == 0x00000000) {
          /* 000010.. ........ 00000010 01000000 */
          extract_decode_Fmt_0(&u.f_decode0, insn);
          if (trans_nop(ctx, &u.f_decode0)) return true;
      }
      if ((insn & 0x03e00000) == 0x00000000) {
          /* 00001000 000..... 00000010 010..... */
          extract_decode_Fmt_1(&u.f_decode1, insn);
          if (trans_copy(ctx, &u.f_decode1)) return true;
      }
```

(continues on next page)

(continued from previous page)

```
  }
  extract_decode_Fmt_2(&u.f_decode2, insn);
  if (trans_or(ctx, &u.f_decode2)) return true;
  return false;
}
```

6.10 Secure Coding Practices

This document covers topics that both developers and security researchers must be aware of so that they can develop safe code and audit existing code properly.

6.10.1 Reporting Security Bugs

For details on how to report security bugs or ask questions about potential security bugs, see the Security Process wiki page.

6.10.2 General Secure C Coding Practices

Most CVEs (security bugs) reported against QEMU are not specific to virtualization or emulation. They are simply C programming bugs. Therefore it's critical to be aware of common classes of security bugs.

There is a wide selection of resources available covering secure C coding. For example, the CERT C Coding Standard covers the most important classes of security bugs.

Instead of describing them in detail here, only the names of the most important classes of security bugs are mentioned:

- Buffer overflows
- Use-after-free and double-free
- Integer overflows
- Format string vulnerabilities

Some of these classes of bugs can be detected by analyzers. Static analysis is performed regularly by Coverity and the most obvious of these bugs are even reported by compilers. Dynamic analysis is possible with valgrind, tsan, and asan.

6.10.3 Input Validation

Inputs from the guest or external sources (e.g. network, files) cannot be trusted and may be invalid. Inputs must be checked before using them in a way that could crash the program, expose host memory to the guest, or otherwise be exploitable by an attacker.

The most sensitive attack surface is device emulation. All hardware register accesses and data read from guest memory must be validated. A typical example is a device that contains multiple units that are selectable by the guest via an index register:

```
typedef struct {
    ProcessingUnit unit[2];
    ...
} MyDeviceState;
```

(continues on next page)

(continued from previous page)

```
static void mydev_writel(void *opaque, uint32_t addr, uint32_t val)
{
    MyDeviceState *mydev = opaque;
    ProcessingUnit *unit;

    switch (addr) {
    case MYDEV_SELECT_UNIT:
        unit = &mydev->unit[val];   <-- this input wasn't validated!
        ...
    }
}
```

If `val` is not in range [0, 1] then an out-of-bounds memory access will take place when `unit` is dereferenced. The code must check that `val` is 0 or 1 and handle the case where it is invalid.

6.10.4 Unexpected Device Accesses

The guest may access device registers in unusual orders or at unexpected moments. Device emulation code must not assume that the guest follows the typical "theory of operation" presented in driver writer manuals. The guest may make nonsense accesses to device registers such as starting operations before the device has been fully initialized.

A related issue is that device emulation code must be prepared for unexpected device register accesses while asynchronous operations are in progress. A well-behaved guest might wait for a completion interrupt before accessing certain device registers. Device emulation code must handle the case where the guest overwrites registers or submits further requests before an ongoing request completes. Unexpected accesses must not cause memory corruption or leaks in QEMU.

Invalid device register accesses can be reported with `qemu_log_mask(LOG_GUEST_ERROR, ...)`. The `-d guest_errors` command-line option enables these log messages.

6.10.5 Live Migration

Device state can be saved to disk image files and shared with other users. Live migration code must validate inputs when loading device state so an attacker cannot gain control by crafting invalid device states. Device state is therefore considered untrusted even though it is typically generated by QEMU itself.

6.10.6 Guest Memory Access Races

Guests with multiple vCPUs may modify guest RAM while device emulation code is running. Device emulation code must copy in descriptors and other guest RAM structures and only process the local copy. This prevents time-of-check-to-time-of-use (TOCTOU) race conditions that could cause QEMU to crash when a vCPU thread modifies guest RAM while device emulation is processing it.

6.11 Translator Internals

QEMU is a dynamic translator. When it first encounters a piece of code, it converts it to the host instruction set. Usually dynamic translators are very complicated and highly CPU dependent. QEMU uses some tricks which make it relatively easily portable and simple while achieving good performances.

QEMU's dynamic translation backend is called TCG, for "Tiny Code Generator". For more information, please take a look at `tcg/README`.

Some notable features of QEMU's dynamic translator are:

6.11.1 CPU state optimisations

The target CPUs have many internal states which change the way it evaluates instructions. In order to achieve a good speed, the translation phase considers that some state information of the virtual CPU cannot change in it. The state is recorded in the Translation Block (TB). If the state changes (e.g. privilege level), a new TB will be generated and the previous TB won't be used anymore until the state matches the state recorded in the previous TB. The same idea can be applied to other aspects of the CPU state. For example, on x86, if the SS, DS and ES segments have a zero base, then the translator does not even generate an addition for the segment base.

6.11.2 Direct block chaining

After each translated basic block is executed, QEMU uses the simulated Program Counter (PC) and other cpu state information (such as the CS segment base value) to find the next basic block.

In order to accelerate the most common cases where the new simulated PC is known, QEMU can patch a basic block so that it jumps directly to the next one.

The most portable code uses an indirect jump. An indirect jump makes it easier to make the jump target modification atomic. On some host architectures (such as x86 or PowerPC), the `JUMP` opcode is directly patched so that the block chaining has no overhead.

6.11.3 Self-modifying code and translated code invalidation

Self-modifying code is a special challenge in x86 emulation because no instruction cache invalidation is signaled by the application when code is modified.

User-mode emulation marks a host page as write-protected (if it is not already read-only) every time translated code is generated for a basic block. Then, if a write access is done to the page, Linux raises a SEGV signal. QEMU then invalidates all the translated code in the page and enables write accesses to the page. For system emulation, write protection is achieved through the software MMU.

Correct translated code invalidation is done efficiently by maintaining a linked list of every translated block contained in a given page. Other linked lists are also maintained to undo direct block chaining.

On RISC targets, correctly written software uses memory barriers and cache flushes, so some of the protection above would not be necessary. However, QEMU still requires that the generated code always matches the target instructions in memory in order to handle exceptions correctly.

6.11.4 Exception support

longjmp() is used when an exception such as division by zero is encountered.

The host SIGSEGV and SIGBUS signal handlers are used to get invalid memory accesses. QEMU keeps a map from host program counter to target program counter, and looks up where the exception happened based on the host program counter at the exception point.

On some targets, some bits of the virtual CPU's state are not flushed to the memory until the end of the translation block. This is done for internal emulation state that is rarely accessed directly by the program and/or changes very often throughout the execution of a translation block—this includes condition codes on x86, delay slots on SPARC, conditional execution on Arm, and so on. This state is stored for each target instruction, and looked up on exceptions.

6.11.5 MMU emulation

For system emulation QEMU uses a software MMU. In that mode, the MMU virtual to physical address translation is done at every memory access.

QEMU uses an address translation cache (TLB) to speed up the translation. In order to avoid flushing the translated code each time the MMU mappings change, all caches in QEMU are physically indexed. This means that each basic block is indexed with its physical address.

In order to avoid invalidating the basic block chain when MMU mappings change, chaining is only performed when the destination of the jump shares a page with the basic block that is performing the jump.

The MMU can also distinguish RAM and ROM memory areas from MMIO memory areas. Access is faster for RAM and ROM because the translation cache also hosts the offset between guest address and host memory. Accessing MMIO memory areas instead calls out to C code for device emulation. Finally, the MMU helps tracking dirty pages and pages pointed to by translation blocks.

6.12 TCG Instruction Counting

TCG has long supported a feature known as icount which allows for instruction counting during execution. This should not be confused with cycle accurate emulation - QEMU does not attempt to emulate how long an instruction would take on real hardware. That is a job for other more detailed (and slower) tools that simulate the rest of a micro-architecture.

This feature is only available for system emulation and is incompatible with multi-threaded TCG. It can be used to better align execution time with wall-clock time so a “slow” device doesn’t run too fast on modern hardware. It can also provides for a degree of deterministic execution and is an essential part of the record/replay support in QEMU.

6.12.1 Core Concepts

At its heart icount is simply a count of executed instructions which is stored in the TimersState of QEMU’s timer sub-system. The number of executed instructions can then be used to calculate QEMU_CLOCK_VIRTUAL which represents the amount of elapsed time in the system since execution started. Depending on the icount mode this may either be a fixed number of ns per instruction or adjusted as execution continues to keep wall clock time and virtual time in sync.

To be able to calculate the number of executed instructions the translator starts by allocating a budget of instructions to be executed. The budget of instructions is limited by how long it will be until the next timer will expire. We store this budget as part of a vCPU icount_decr field which shared with the machinery for handling cpu_exit(). The whole field is checked at the start of every translated block and will cause a return to the outer loop to deal with whatever caused the exit.

In the case of icount, before the flag is checked we subtract the number of instructions the translation block would execute. If this would cause the instruction budget to go negative we exit the main loop and regenerate a new translation block with exactly the right number of instructions to take the budget to 0 meaning whatever timer was due to expire will expire exactly when we exit the main run loop.

Dealing with MMIO

While we can adjust the instruction budget for known events like timer expiry we cannot do the same for MMIO. Every load/store we execute might potentially trigger an I/O event, at which point we will need an up to date and accurate reading of the icount number.

To deal with this case, when an I/O access is made we:

- restore un-executed instructions to the icount budget
- re-compile a single[1] instruction block for the current PC
- exit the cpu loop and execute the re-compiled block

The new block is created with the CF_LAST_IO compile flag which ensures the final instruction translation starts with a call to gen_io_start() so we don't enter a perpetual loop constantly recompiling a single instruction block. For translators using the common translator_loop this is done automatically.

Other I/O operations

MMIO isn't the only type of operation for which we might need a correct and accurate clock. IO port instructions and accesses to system registers are the common examples here. These instructions have to be handled by the individual translators which have the knowledge of which operations are I/O operations.

When the translator is handling an instruction of this kind:

- **it must call gen_io_start() if icount is enabled, at some** point before the generation of the code which actually does the I/O, using a code fragment similar to:

```
if (tb_cflags(s->base.tb) & CF_USE_ICOUNT) {
    gen_io_start();
}
```

- it must end the TB immediately after this instruction

Note that some older front-ends call a "gen_io_end()" function: this is obsolete and should not be used.

6.13 Introduction

This document outlines the design for multi-threaded TCG (a.k.a MTTCG) system-mode emulation. user-mode emulation has always mirrored the thread structure of the translated executable although some of the changes done for MTTCG system emulation have improved the stability of linux-user emulation.

The original system-mode TCG implementation was single threaded and dealt with multiple CPUs with simple round-robin scheduling. This simplified a lot of things but became increasingly limited as systems being emulated gained additional cores and per-core performance gains for host systems started to level off.

6.14 vCPU Scheduling

We introduce a new running mode where each vCPU will run on its own user-space thread. This is enabled by default for all FE/BE combinations where the host memory model is able to accommodate the guest (TCG_GUEST_DEFAULT_MO & ~TCG_TARGET_DEFAULT_MO is zero) and the guest has had the required work done to support this safely (TARGET_SUPPORTS_MTTCG).

System emulation will fall back to the original round robin approach if:

- forced by –accel tcg,thread=single
- enabling –icount mode
- 64 bit guests on 32 bit hosts (TCG_OVERSIZED_GUEST)

[1] sometimes two instructions if dealing with delay slots

In the general case of running translated code there should be no inter-vCPU dependencies and all vCPUs should be able to run at full speed. Synchronisation will only be required while accessing internal shared data structures or when the emulated architecture requires a coherent representation of the emulated machine state.

6.15 Shared Data Structures

6.15.1 Main Run Loop

Even when there is no code being generated there are a number of structures associated with the hot-path through the main run-loop. These are associated with looking up the next translation block to execute. These include:

> tb_jmp_cache (per-vCPU, cache of recent jumps) tb_ctx.htable (global hash table, phys address->tb lookup)

As TB linking only occurs when blocks are in the same page this code is critical to performance as looking up the next TB to execute is the most common reason to exit the generated code.

DESIGN REQUIREMENT: Make access to lookup structures safe with multiple reader/writer threads. Minimise any lock contention to do it.

The hot-path avoids using locks where possible. The tb_jmp_cache is updated with atomic accesses to ensure consistent results. The fall back QHT based hash table is also designed for lockless lookups. Locks are only taken when code generation is required or TranslationBlocks have their block-to-block jumps patched.

6.15.2 Global TCG State

User-mode emulation

We need to protect the entire code generation cycle including any post generation patching of the translated code. This also implies a shared translation buffer which contains code running on all cores. Any execution path that comes to the main run loop will need to hold a mutex for code generation. This also includes times when we need flush code or entries from any shared lookups/caches. Structures held on a per-vCPU basis won't need locking unless other vCPUs will need to modify them.

DESIGN REQUIREMENT: Add locking around all code generation and TB patching.

(Current solution)

Code generation is serialised with mmap_lock().

!User-mode emulation

Each vCPU has its own TCG context and associated TCG region, thereby requiring no locking during translation.

6.15.3 Translation Blocks

Currently the whole system shares a single code generation buffer which when full will force a flush of all translations and start from scratch again. Some operations also force a full flush of translations including:

- debugging operations (breakpoint insertion/removal)
- some CPU helper functions
- linux-user spawning its first thread

This is done with the async_safe_run_on_cpu() mechanism to ensure all vCPUs are quiescent when changes are being made to shared global structures.

More granular translation invalidation events are typically due to a change of the state of a physical page:

- code modification (self modify code, patching code)
- page changes (new page mapping in linux-user mode)

While setting the invalid flag in a TranslationBlock will stop it being used when looked up in the hot-path there are a number of other book-keeping structures that need to be safely cleared.

Any TranslationBlocks which have been patched to jump directly to the now invalid blocks need the jump patches reversing so they will return to the C code.

There are a number of look-up caches that need to be properly updated including the:

- jump lookup cache
- the physical-to-tb lookup hash table
- the global page table

The global page table (l1_map) which provides a multi-level look-up for PageDesc structures which contain pointers to the start of a linked list of all Translation Blocks in that page (see page_next).

Both the jump patching and the page cache involve linked lists that the invalidated TranslationBlock needs to be removed from.

DESIGN REQUIREMENT: Safely handle invalidation of TBs

- safely patch/revert direct jumps
- remove central PageDesc lookup entries
- ensure lookup caches/hashes are safely updated

(Current solution)

The direct jump themselves are updated atomically by the TCG tb_set_jmp_target() code. Modification to the linked lists that allow searching for linked pages are done under the protection of tb->jmp_lock, where tb is the destination block of a jump. Each origin block keeps a pointer to its destinations so that the appropriate lock can be acquired before iterating over a jump list.

The global page table is a lockless radix tree; cmpxchg is used to atomically insert new elements.

The lookup caches are updated atomically and the lookup hash uses QHT which is designed for concurrent safe lookup.

Parallel code generation is supported. QHT is used at insertion time as the synchronization point across threads, thereby ensuring that we only keep track of a single TranslationBlock for each guest code block.

6.15.4 Memory maps and TLBs

The memory handling code is fairly critical to the speed of memory access in the emulated system. The SoftMMU code is designed so the hot-path can be handled entirely within translated code. This is handled with a per-vCPU TLB structure which once populated will allow a series of accesses to the page to occur without exiting the translated code. It is possible to set flags in the TLB address which will ensure the slow-path is taken for each access. This can be done to support:

- Memory regions (dividing up access to PIO, MMIO and RAM)
- Dirty page tracking (for code gen, SMC detection, migration and display)
- Virtual TLB (for translating guest address->real address)

When the TLB tables are updated by a vCPU thread other than their own we need to ensure it is done in a safe way so no inconsistent state is seen by the vCPU thread.

Some operations require updating a number of vCPUs TLBs at the same time in a synchronised manner.

DESIGN REQUIREMENTS:

- TLB Flush All/Page - can be across-vCPUs - cross vCPU TLB flush may need other vCPU brought to halt - change may need to be visible to the calling vCPU immediately
- TLB Flag Update - usually cross-vCPU - want change to be visible as soon as possible
- TLB Update (update a CPUTLBEntry, via tlb_set_page_with_attrs) - This is a per-vCPU table - by definition can't race - updated by its own thread when the slow-path is forced

(Current solution)

We have updated cputlb.c to defer operations when a cross-vCPU operation with async_run_on_cpu() which ensures each vCPU sees a coherent state when it next runs its work (in a few instructions time).

A new set up operations (tlb_flush_*_all_cpus) take an additional flag which when set will force synchronisation by setting the source vCPUs work as "safe work" and exiting the cpu run loop. This ensure by the time execution restarts all flush operations have completed.

TLB flag updates are all done atomically and are also protected by the corresponding page lock.

(Known limitation)

Not really a limitation but the wait mechanism is overly strict for some architectures which only need flushes completed by a barrier instruction. This could be a future optimisation.

6.15.5 Emulated hardware state

Currently thanks to KVM work any access to IO memory is automatically protected by the global iothread mutex, also known as the BQL (Big Qemu Lock). Any IO region that doesn't use global mutex is expected to do its own locking.

However IO memory isn't the only way emulated hardware state can be modified. Some architectures have model specific registers that trigger hardware emulation features. Generally any translation helper that needs to update more than a single vCPUs of state should take the BQL.

As the BQL, or global iothread mutex is shared across the system we push the use of the lock as far down into the TCG code as possible to minimise contention.

(Current solution)

MMIO access automatically serialises hardware emulation by way of the BQL. Currently Arm targets serialise all ARM_CP_IO register accesses and also defer the reset/startup of vCPUs to the vCPU context by way of async_run_on_cpu().

Updates to interrupt state are also protected by the BQL as they can often be cross vCPU.

6.16 Memory Consistency

Between emulated guests and host systems there are a range of memory consistency models. Even emulating weakly ordered systems on strongly ordered hosts needs to ensure things like store-after-load re-ordering can be prevented when the guest wants to.

6.16.1 Memory Barriers

Barriers (sometimes known as fences) provide a mechanism for software to enforce a particular ordering of memory operations from the point of view of external observers (e.g. another processor core). They can apply to any memory operations as well as just loads or stores.

The Linux kernel has an excellent *write-up <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/Documentation/memory-barriers.txt>* on the various forms of memory barrier and the guarantees they can provide.

Barriers are often wrapped around synchronisation primitives to provide explicit memory ordering semantics. However they can be used by themselves to provide safe lockless access by ensuring for example a change to a signal flag will only be visible once the changes to payload are.

DESIGN REQUIREMENT: Add a new tcg_memory_barrier op

This would enforce a strong load/store ordering so all loads/stores complete at the memory barrier. On single-core non-SMP strongly ordered backends this could become a NOP.

Aside from explicit standalone memory barrier instructions there are also implicit memory ordering semantics which comes with each guest memory access instruction. For example all x86 load/stores come with fairly strong guarantees of sequential consistency whereas Arm has special variants of load/store instructions that imply acquire/release semantics.

In the case of a strongly ordered guest architecture being emulated on a weakly ordered host the scope for a heavy performance impact is quite high.

DESIGN REQUIREMENTS: Be efficient with use of memory barriers

- host systems with stronger implied guarantees can skip some barriers
- merge consecutive barriers to the strongest one

(Current solution)

The system currently has a tcg_gen_mb() which will add memory barrier operations if code generation is being done in a parallel context. The tcg_optimize() function attempts to merge barriers up to their strongest form before any load/store operations. The solution was originally developed and tested for linux-user based systems. All backends have been converted to emit fences when required. So far the following front-ends have been updated to emit fences when required:

- target-i386
- target-arm
- target-aarch64
- target-alpha
- target-mips

6.16.2 Memory Control and Maintenance

This includes a class of instructions for controlling system cache behaviour. While QEMU doesn't model cache behaviour these instructions are often seen when code modification has taken place to ensure the changes take effect.

6.16.3 Synchronisation Primitives

There are two broad types of synchronisation primitives found in modern ISAs: atomic instructions and exclusive regions.

The first type offer a simple atomic instruction which will guarantee some sort of test and conditional store will be truly atomic w.r.t. other cores sharing access to the memory. The classic example is the x86 cmpxchg instruction.

The second type offer a pair of load/store instructions which offer a guarantee that a region of memory has not been touched between the load and store instructions. An example of this is Arm's ldrex/strex pair where the strex instruction will return a flag indicating a successful store only if no other CPU has accessed the memory region since the ldrex.

Traditionally TCG has generated a series of operations that work because they are within the context of a single translation block so will have completed before another CPU is scheduled. However with the ability to have multiple threads running to emulate multiple CPUs we will need to explicitly expose these semantics.

DESIGN REQUIREMENTS:

- Support classic atomic instructions
- Support load/store exclusive (or load link/store conditional) pairs
- Generic enough infrastructure to support all guest architectures

CURRENT OPEN QUESTIONS:

- How problematic is the ABA problem in general?

(Current solution)

The TCG provides a number of atomic helpers (tcg_gen_atomic_*) which can be used directly or combined to emulate other instructions like Arm's ldrex/strex instructions. While they are susceptible to the ABA problem so far common guests have not implemented patterns where this may be a problem - typically presenting a locking ABI which assumes cmpxchg like semantics.

The code also includes a fall-back for cases where multi-threaded TCG ops can't work (e.g. guest atomic width > host atomic width). In this case an EXCP_ATOMIC exit occurs and the instruction is emulated with an exclusive lock which ensures all emulation is serialised.

While the atomic helpers look good enough for now there may be a need to look at solutions that can more closely model the guest architectures semantics.

6.17 QEMU TCG Plugins

QEMU TCG plugins provide a way for users to run experiments taking advantage of the total system control emulation can have over a guest. It provides a mechanism for plugins to subscribe to events during translation and execution and optionally callback into the plugin during these events. TCG plugins are unable to change the system state only monitor it passively. However they can do this down to an individual instruction granularity including potentially subscribing to all load and store operations.

6.17.1 API Stability

This is a new feature for QEMU and it does allow people to develop out-of-tree plugins that can be dynamically linked into a running QEMU process. However the project reserves the right to change or break the API should it need to do so. The best way to avoid this is to submit your plugin upstream so they can be updated if/when the API changes.

API versioning

All plugins need to declare a symbol which exports the plugin API version they were built against. This can be done simply by:

```
QEMU_PLUGIN_EXPORT int qemu_plugin_version = QEMU_PLUGIN_VERSION;
```

The core code will refuse to load a plugin that doesn't export a *qemu_plugin_version* symbol or if plugin version is outside of QEMU's supported range of API versions.

Additionally the *qemu_info_t* structure which is passed to the *qemu_plugin_install* method of a plugin will detail the minimum and current API versions supported by QEMU. The API version will be incremented if new APIs are added. The minimum API version will be incremented if existing APIs are changed or removed.

Exposure of QEMU internals

The plugin architecture actively avoids leaking implementation details about how QEMU's translation works to the plugins. While there are conceptions such as translation time and translation blocks the details are opaque to plugins. The plugin is able to query select details of instructions and system configuration only through the exported *qemu_plugin* functions.

Query Handle Lifetime

Each callback provides an opaque anonymous information handle which can usually be further queried to find out information about a translation, instruction or operation. The handles themselves are only valid during the lifetime of the callback so it is important that any information that is needed is extracted during the callback and saved by the plugin.

6.17.2 Usage

The QEMU binary needs to be compiled for plugin support:

```
configure --enable-plugins
```

Once built a program can be run with multiple plugins loaded each with their own arguments:

```
$QEMU $OTHER_QEMU_ARGS \
    -plugin tests/plugin/libhowvec.so,arg=inline,arg=hint \
    -plugin tests/plugin/libhotblocks.so
```

Arguments are plugin specific and can be used to modify their behaviour. In this case the howvec plugin is being asked to use inline ops to count and break down the hint instructions by type.

6.17.3 Plugin Life cycle

First the plugin is loaded and the public qemu_plugin_install function is called. The plugin will then register callbacks for various plugin events. Generally plugins will register a handler for the *atexit* if they want to dump a summary of collected information once the program/system has finished running.

When a registered event occurs the plugin callback is invoked. The callbacks may provide additional information. In the case of a translation event the plugin has an option to enumerate the instructions in a block of instructions and optionally register callbacks to some or all instructions when they are executed.

There is also a facility to add an inline event where code to increment a counter can be directly inlined with the translation. Currently only a simple increment is supported. This is not atomic so can miss counts. If you want absolute precision you should use a callback which can then ensure atomicity itself.

Finally when QEMU exits all the registered *atexit* callbacks are invoked.

6.17.4 Internals

Locking

We have to ensure we cannot deadlock, particularly under MTTCG. For this we acquire a lock when called from plugin code. We also keep the list of callbacks under RCU so that we do not have to hold the lock when calling the callbacks. This is also for performance, since some callbacks (e.g. memory access callbacks) might be called very frequently.

- A consequence of this is that we keep our own list of CPUs, so that we do not have to worry about locking order wrt cpu_list_lock.
- Use a recursive lock, since we can get registration calls from callbacks.

As a result registering/unregistering callbacks is "slow", since it takes a lock. But this is very infrequent; we want performance when calling (or not calling) callbacks, not when registering them. Using RCU is great for this.

We support the uninstallation of a plugin at any time (e.g. from plugin callbacks). This allows plugins to remove themselves if they no longer want to instrument the code. This operation is asynchronous which means callbacks may still occur after the uninstall operation is requested. The plugin isn't completely uninstalled until the safe work has executed while all vCPUs are quiescent.

6.18 Bitwise operations

The header `qemu/bitops.h` provides utility functions for performing bitwise operations.

void **`set_bit`** (long *nr*, unsigned long * *addr*)
Set a bit in memory

Parameters

`long nr` the bit to set

`unsigned long * addr` the address to start counting from

void **`set_bit_atomic`** (long *nr*, unsigned long * *addr*)
Set a bit in memory atomically

Parameters

`long nr` the bit to set

`unsigned long * addr` the address to start counting from

void **`clear_bit`** (long *nr*, unsigned long * *addr*)
Clears a bit in memory

Parameters

`long nr` Bit to clear

`unsigned long * addr` Address to start counting from

void **`change_bit`** (long *nr*, unsigned long * *addr*)
Toggle a bit in memory

Parameters

`long nr` Bit to change

`unsigned long * addr` Address to start counting from

int **`test_and_set_bit`** (long *nr*, unsigned long * *addr*)
Set a bit and return its old value

Parameters

`long nr` Bit to set

`unsigned long * addr` Address to count from

int **`test_and_clear_bit`** (long *nr*, unsigned long * *addr*)
Clear a bit and return its old value

Parameters

`long nr` Bit to clear

`unsigned long * addr` Address to count from

int **`test_and_change_bit`** (long *nr*, unsigned long * *addr*)
Change a bit and return its old value

Parameters

`long nr` Bit to change

`unsigned long * addr` Address to count from

int **`test_bit`** (long *nr*, const unsigned long * *addr*)
Determine whether a bit is set

Parameters

`long nr` bit number to test

`const unsigned long * addr` Address to start counting from

unsigned long **`find_last_bit`** (const unsigned long * *addr*, unsigned long *size*)
find the last set bit in a memory region

Parameters

`const unsigned long * addr` The address to start the search at

`unsigned long size` The maximum size to search

Description

Returns the bit number of the first set bit, or size.

unsigned long **`find_next_bit`** (const unsigned long * *addr*, unsigned long *size*, unsigned long *offset*)
find the next set bit in a memory region

Parameters

`const unsigned long * addr` The address to base the search on

`unsigned long size` The bitmap size in bits

`unsigned long offset` The bitnumber to start searching at

unsigned long **`find_next_zero_bit`** (const unsigned long * *addr*, unsigned long *size*, unsigned long *offset*)
find the next cleared bit in a memory region

Parameters

`const unsigned long * addr` The address to base the search on

`unsigned long size` The bitmap size in bits

`unsigned long offset` The bitnumber to start searching at

unsigned long **`find_first_bit`** (const unsigned long * *addr*, unsigned long *size*)
find the first set bit in a memory region

Parameters

`const unsigned long * addr` The address to start the search at

`unsigned long size` The maximum size to search

Description

Returns the bit number of the first set bit.

unsigned long **`find_first_zero_bit`** (const unsigned long * *addr*, unsigned long *size*)
find the first cleared bit in a memory region

Parameters

`const unsigned long * addr` The address to start the search at

`unsigned long size` The maximum size to search

Description

Returns the bit number of the first cleared bit.

uint8_t **`rol8`** (uint8_t *word*, unsigned int *shift*)
rotate an 8-bit value left

Parameters

`uint8_t word` value to rotate

`unsigned int shift` bits to roll

uint8_t **`ror8`** (uint8_t *word*, unsigned int *shift*)
rotate an 8-bit value right

Parameters

`uint8_t word` value to rotate

`unsigned int shift` bits to roll

uint16_t **`rol16`** (uint16_t *word*, unsigned int *shift*)
rotate a 16-bit value left

Parameters

`uint16_t word` value to rotate

`unsigned int shift` bits to roll

uint16_t **`ror16`** (uint16_t *word*, unsigned int *shift*)
rotate a 16-bit value right

Parameters

`uint16_t word` value to rotate

`unsigned int shift` bits to roll

uint32_t **`rol32`** (uint32_t *word*, unsigned int *shift*)
rotate a 32-bit value left

Parameters

`uint32_t word` value to rotate

`unsigned int shift` bits to roll

uint32_t **`ror32`** (uint32_t *word*, unsigned int *shift*)
rotate a 32-bit value right

Parameters

`uint32_t word` value to rotate

`unsigned int shift` bits to roll

uint64_t **`rol64`** (uint64_t *word*, unsigned int *shift*)
rotate a 64-bit value left

Parameters

`uint64_t word` value to rotate

`unsigned int shift` bits to roll

uint64_t **`ror64`** (uint64_t *word*, unsigned int *shift*)
rotate a 64-bit value right

Parameters

`uint64_t word` value to rotate

`unsigned int shift` bits to roll

uint32_t **`extract32`** (uint32_t *value*, int *start*, int *length*)

Parameters

`uint32_t value` the value to extract the bit field from

`int start` the lowest bit in the bit field (numbered from 0)

`int length` the length of the bit field

Description

Extract from the 32 bit input **value** the bit field specified by the **start** and **length** parameters, and return it. The bit field must lie entirely within the 32 bit word. It is valid to request that all 32 bits are returned (ie **length** 32 and **start** 0).

Return

the value of the bit field extracted from the input value.

uint8_t **`extract8`** (uint8_t *value*, int *start*, int *length*)

Parameters

`uint8_t value` the value to extract the bit field from

`int start` the lowest bit in the bit field (numbered from 0)

`int length` the length of the bit field

Description

Extract from the 8 bit input **value** the bit field specified by the **start** and **length** parameters, and return it. The bit field must lie entirely within the 8 bit word. It is valid to request that all 8 bits are returned (ie **length** 8 and **start** 0).

Return

the value of the bit field extracted from the input value.

uint16_t **`extract16`** (uint16_t *value*, int *start*, int *length*)

Parameters

`uint16_t value` the value to extract the bit field from

`int start` the lowest bit in the bit field (numbered from 0)

`int length` the length of the bit field

Description

Extract from the 16 bit input **value** the bit field specified by the **start** and **length** parameters, and return it. The bit field must lie entirely within the 16 bit word. It is valid to request that all 16 bits are returned (ie **length** 16 and **start** 0).

Return

the value of the bit field extracted from the input value.

uint64_t **`extract64`** (uint64_t *value*, int *start*, int *length*)

Parameters

`uint64_t value` the value to extract the bit field from

`int start` the lowest bit in the bit field (numbered from 0)

`int length` the length of the bit field

Description

Extract from the 64 bit input **value** the bit field specified by the **start** and **length** parameters, and return it. The bit field must lie entirely within the 64 bit word. It is valid to request that all 64 bits are returned (ie **length** 64 and **start** 0).

Return

the value of the bit field extracted from the input value.

int32_t **`sextract32`** (uint32_t *value*, int *start*, int *length*)

Parameters

`uint32_t value` the value to extract the bit field from

`int start` the lowest bit in the bit field (numbered from 0)

`int length` the length of the bit field

Description

Extract from the 32 bit input **value** the bit field specified by the **start** and **length** parameters, and return it, sign extended to an int32_t (ie with the most significant bit of the field propagated to all the upper bits of the return value). The bit field must lie entirely within the 32 bit word. It is valid to request that all 32 bits are returned (ie **length** 32 and **start** 0).

Return

the sign extended value of the bit field extracted from the input value.

int64_t **`sextract64`** (uint64_t *value*, int *start*, int *length*)

Parameters

`uint64_t value` the value to extract the bit field from

`int start` the lowest bit in the bit field (numbered from 0)

`int length` the length of the bit field

Description

Extract from the 64 bit input **value** the bit field specified by the **start** and **length** parameters, and return it, sign extended to an int64_t (ie with the most significant bit of the field propagated to all the upper bits of the return value). The bit field must lie entirely within the 64 bit word. It is valid to request that all 64 bits are returned (ie **length** 64 and **start** 0).

Return

the sign extended value of the bit field extracted from the input value.

uint32_t **`deposit32`** (uint32_t *value*, int *start*, int *length*, uint32_t *fieldval*)

Parameters

`uint32_t value` initial value to insert bit field into

`int start` the lowest bit in the bit field (numbered from 0)

`int length` the length of the bit field

`uint32_t fieldval` the value to insert into the bit field

Description

Deposit **fieldval** into the 32 bit **value** at the bit field specified by the **start** and **length** parameters, and return the modified **value**. Bits of **value** outside the bit field are not modified. Bits of **fieldval** above the least significant **length** bits are ignored. The bit field must lie entirely within the 32 bit word. It is valid to request that all 32 bits are modified (ie **length** 32 and **start** 0).

Return

the modified **value**.

uint64_t **`deposit64`** (uint64_t *value*, int *start*, int *length*, uint64_t *fieldval*)

Parameters

`uint64_t value` initial value to insert bit field into

`int start` the lowest bit in the bit field (numbered from 0)

`int length` the length of the bit field

`uint64_t fieldval` the value to insert into the bit field

Description

Deposit **fieldval** into the 64 bit **value** at the bit field specified by the **start** and **length** parameters, and return the modified **value**. Bits of **value** outside the bit field are not modified. Bits of **fieldval** above the least significant **length** bits are ignored. The bit field must lie entirely within the 64 bit word. It is valid to request that all 64 bits are modified (ie **length** 64 and **start** 0).

Return

the modified **value**.

uint32_t **`half_shuffle32`** (uint32_t *x*)

Parameters

`uint32_t x` 32-bit value (of which only the bottom 16 bits are of interest)

Description

Given an input value:

```
xxxx xxxx xxxx xxxx ABCD EFGH IJKL MNOP
```

return the value where the bottom 16 bits are spread out into the odd bits in the word, and the even bits are zeroed:

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N 0O0P
```

Any bits set in the top half of the input are ignored.

Return

the shuffled bits.

uint64_t **half_shuffle64** (uint64_t *x*)

Parameters

uint64_t x 64-bit value (of which only the bottom 32 bits are of interest)

Description

Given an input value:

```
xxxx xxxx xxxx .... xxxx xxxx ABCD EFGH IJKL MNOP QRST UVWX YZab cdef
```

return the value where the bottom 32 bits are spread out into the odd bits in the word, and the even bits are zeroed:

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N .... 0U0V 0W0X 0Y0Z 0a0b 0c0d 0e0f
```

Any bits set in the top half of the input are ignored.

Return

the shuffled bits.

uint32_t **half_unshuffle32** (uint32_t *x*)

Parameters

uint32_t x 32-bit value (of which only the odd bits are of interest)

Description

Given an input value:

```
xAxB xCxD xExF xGxH xIxJ xKxL xMxN xOxP
```

return the value where all the odd bits are compressed down into the low half of the word, and the high half is zeroed:

```
0000 0000 0000 0000 ABCD EFGH IJKL MNOP
```

Any even bits set in the input are ignored.

Return

the unshuffled bits.

uint64_t **half_unshuffle64** (uint64_t *x*)

Parameters

uint64_t x 64-bit value (of which only the odd bits are of interest)

Description

Given an input value:

```
xAxB xCxD xExF xGxH xIxJ xKxL xMxN .... xUxV xWxX xYxZ xaxb xcxd xexf
```

return the value where all the odd bits are compressed down into the low half of the word, and the high half is zeroed:

```
0000 0000 0000 .... 0000 0000 ABCD EFGH IJKL MNOP QRST UVWX YZab cdef
```

Any even bits set in the input are ignored.

Return

the unshuffled bits.

6.19 Reset in QEMU: the Resettable interface

The reset of qemu objects is handled using the resettable interface declared in `include/hw/resettable.h`.

This interface allows objects to be grouped (on a tree basis); so that the whole group can be reset consistently. Each individual member object does not have to care about others; in particular, problems of order (which object is reset first) are addressed.

As of now DeviceClass and BusClass implement this interface.

6.19.1 Triggering reset

This section documents the APIs which "users" of a resettable object should use to control it. All resettable control functions must be called while holding the iothread lock.

You can apply a reset to an object using `resettable_assert_reset()`. You need to call `resettable_release_reset()` to release the object from reset. To instantly reset an object, without keeping it in reset state, just call `resettable_reset()`. These functions take two parameters: a pointer to the object to reset and a reset type.

Several types of reset will be supported. For now only cold reset is defined; others may be added later. The Resettable interface handles reset types with an enum:

`RESET_TYPE_COLD` Cold reset is supported by every resettable object. In QEMU, it means we reset to the initial state corresponding to the start of QEMU; this might differ from what is a real hardware cold reset. It differs from other resets (like warm or bus resets) which may keep certain parts untouched.

Calling `resettable_reset()` is equivalent to calling `resettable_assert_reset()` then `resettable_release_reset()`. It is possible to interleave multiple calls to these three functions. There may be several reset sources/controllers of a given object. The interface handles everything and the different reset controllers do not need to know anything about each others. The object will leave reset state only when each other controllers end their reset operation. This point is handled internally by maintaining a count of in-progress resets; it is crucial to call `resettable_release_reset()` one time and only one time per `resettable_assert_reset()` call.

For now migration of a device or bus in reset is not supported. Care must be taken not to delay `resettable_release_reset()` after its `resettable_assert_reset()` counterpart.

Note that, since resettable is an interface, the API takes a simple Object as parameter. Still, it is a programming error to call a resettable function on a non-resettable object and it will trigger a run time assert error. Since most calls to resettable interface are done through base class functions, such an error is not likely to happen.

For Devices and Buses, the following helper functions exist:

- `device_cold_reset()`
- `bus_cold_reset()`

These are simple wrappers around resettable_reset() function; they only cast the Device or Bus into an Object and pass the cold reset type. When possible prefer to use these functions instead of `resettable_reset()`.

Device and bus functions co-exist because there can be semantic differences between resetting a bus and resetting the controller bridge which owns it. For example, consider a SCSI controller. Resetting the controller puts all its registers back to what reset state was as well as reset everything on the SCSI bus, whereas resetting just the SCSI bus only resets everything that's on it but not the controller.

6.19.2 Multi-phase mechanism

This section documents the internals of the resettable interface.

The resettable interface uses a multi-phase system to relieve objects and machines from reset ordering problems. To address this, the reset operation of an object is split into three well defined phases.

When resetting several objects (for example the whole machine at simulation startup), all first phases of all objects are executed, then all second phases and then all third phases.

The three phases are:

1. The **enter** phase is executed when the object enters reset. It resets only local state of the object; it must not do anything that has a side-effect on other objects, such as raising or lowering a qemu_irq line or reading or writing guest memory.
2. The **hold** phase is executed for entry into reset, once every object in the group which is being reset has had its *enter* phase executed. At this point devices can do actions that affect other objects.
3. The **exit** phase is executed when the object leaves the reset state. Actions affecting other objects are permitted.

As said in previous section, the interface maintains a count of reset. This count is used to ensure phases are executed only when required. *enter* and *hold* phases are executed only when asserting reset for the first time (if an object is already in reset state when calling `resettable_assert_reset()` or `resettable_reset()`, they are not executed). The *exit* phase is executed only when the last reset operation ends. Therefore the object does not need to care how many of reset controllers it has and how many of them have started a reset.

6.19.3 Handling reset in a resettable object

This section documents the APIs that an implementation of a resettable object must provide and what functions it has access to. It is intended for people who want to implement or convert a class which has the resettable interface; for example when specializing an existing device or bus.

Methods to implement

Three methods should be defined or left empty. Each method corresponds to a phase of the reset; they are name `phases.enter()`, `phases.hold()` and `phases.exit()`. They all take the object as parameter. The *enter* method also take the reset type as second parameter.

When extending an existing class, these methods may need to be extended too. The `resettable_class_set_parent_phases()` class function may be used to backup parent class methods.

Here follows an example to implement reset for a Device which sets an IO while in reset.

```
static void mydev_reset_enter(Object *obj, ResetType type)
{
    MyDevClass *myclass = MYDEV_GET_CLASS(obj);
```

(continues on next page)

(continued from previous page)

```
    MyDevState *mydev = MYDEV(obj);
    /* call parent class enter phase */
    if (myclass->parent_phases.enter) {
        myclass->parent_phases.enter(obj, type);
    }
    /* initialize local state only */
    mydev->var = 0;
}

static void mydev_reset_hold(Object *obj)
{
    MyDevClass *myclass = MYDEV_GET_CLASS(obj);
    MyDevState *mydev = MYDEV(obj);
    /* call parent class hold phase */
    if (myclass->parent_phases.hold) {
        myclass->parent_phases.hold(obj);
    }
    /* set an IO */
    qemu_set_irq(mydev->irq, 1);
}

static void mydev_reset_exit(Object *obj)
{
    MyDevClass *myclass = MYDEV_GET_CLASS(obj);
    MyDevState *mydev = MYDEV(obj);
    /* call parent class exit phase */
    if (myclass->parent_phases.exit) {
        myclass->parent_phases.exit(obj);
    }
    /* clear an IO */
    qemu_set_irq(mydev->irq, 0);
}

typedef struct MyDevClass {
    MyParentClass parent_class;
    /* to store eventual parent reset methods */
    ResettablePhases parent_phases;
} MyDevClass;

static void mydev_class_init(ObjectClass *class, void *data)
{
    MyDevClass *myclass = MYDEV_CLASS(class);
    ResettableClass *rc = RESETTABLE_CLASS(class);
    resettable_class_set_parent_reset_phases(rc,
                                             mydev_reset_enter,
                                             mydev_reset_hold,
                                             mydev_reset_exit,
                                             &myclass->parent_phases);
}
```

In the above example, we override all three phases. It is possible to override only some of them by passing NULL instead of a function pointer to `resettable_class_set_parent_reset_phases()`. For example, the following will only override the *enter* phase and leave *hold* and *exit* untouched:

```
resettable_class_set_parent_reset_phases(rc, mydev_reset_enter,
                                         NULL, NULL,
                                         &myclass->parent_phases);
```

This is equivalent to providing a trivial implementation of the hold and exit phases which does nothing but call the parent class's implementation of the phase.

Polling the reset state

Resettable interface provides the `resettable_is_in_reset()` function. This function returns true if the object parameter is currently under reset.

An object is under reset from the beginning of the *init* phase to the end of the *exit* phase. During all three phases, the function will return that the object is in reset.

This function may be used if the object behavior has to be adapted while in reset state. For example if a device has an irq input, it will probably need to ignore it while in reset; then it can for example check the reset state at the beginning of the irq callback.

Note that until migration of the reset state is supported, an object should not be left in reset. So apart from being currently executing one of the reset phases, the only cases when this function will return true is if an external interaction (like changing an io) is made during *hold* or *exit* phase of another object in the same reset group.

Helpers `device_is_in_reset()` and `bus_is_in_reset()` are also provided for devices and buses and should be preferred.

6.19.4 Base class handling of reset

This section documents parts of the reset mechanism that you only need to know about if you are extending it to work with a new base class other than DeviceClass or BusClass, or maintaining the existing code in those classes. Most people can ignore it.

Methods to implement

There are two other methods that need to exist in a class implementing the interface: `get_state()` and `child_foreach()`.

`get_state()` is simple. *resettable* is an interface and, as a consequence, does not have any class state structure. But in order to factorize the code, we need one. This method must return a pointer to `ResettableState` structure. The structure must be allocated by the base class; preferably it should be located inside the object instance structure.

`child_foreach()` is more complex. It should execute the given callback on every reset child of the given resettable object. All children must be resettable too. Additional parameters (a reset type and an opaque pointer) must be passed to the callback too.

In `DeviceClass` and `BusClass` the `ResettableState` is located `DeviceState` and `BusState` structure. `child_foreach()` is implemented to follow the bus hierarchy; for a bus, it calls the function on every child device; for a device, it calls the function on every bus child. When we reset the main system bus, we reset the whole machine bus tree.

Changing a resettable parent

One thing which should be taken care of by the base class is handling reset hierarchy changes.

The reset hierarchy is supposed to be static and built during machine creation. But there are actually some exceptions. To cope with this, the resettable API provides `resettable_change_parent()`. This function allows to set, update or remove the parent of a resettable object after machine creation is done. As parameters, it takes the object being moved, the old parent if any and the new parent if any.

This function can be used at any time when not in a reset operation. During a reset operation it must be used only in *hold* phase. Using it in *enter* or *exit* phase is an error. Also it should not be used during machine creation, although it is harmless to do so: the function is a no-op as long as old and new parent are NULL or not in reset.

There is currently 2 cases where this function is used:

1. *device hotplug*; it means a new device is introduced on a live bus.
2. *hot bus change*; it means an existing live device is added, moved or removed in the bus hierarchy. At the moment, it occurs only in the raspi machines for changing the sdbus used by sd card.

6.20 Booting from real channel-attached devices on s390x

6.20.1 s390 hardware IPL

The s390 hardware IPL process consists of the following steps.

1. A READ IPL ccw is constructed in memory location `0x0`. This ccw, by definition, reads the IPL1 record which is located on the disk at cylinder 0 track 0 record 1. Note that the chain flag is on in this ccw so when it is complete another ccw will be fetched and executed from memory location `0x08`.
2. Execute the Read IPL ccw at `0x00`, thereby reading IPL1 data into `0x00`. IPL1 data is 24 bytes in length and consists of the following pieces of information: `[psw][read ccw][tic ccw]`. When the machine executes the Read IPL ccw it read the 24-bytes of IPL1 to be read into memory starting at location `0x0`. Then the ccw program at `0x08` which consists of a read ccw and a tic ccw is automatically executed because of the chain flag from the original READ IPL ccw. The read ccw will read the IPL2 data into memory and the TIC (Transfer In Channel) will transfer control to the channel program contained in the IPL2 data. The TIC channel command is the equivalent of a branch/jump/goto instruction for channel programs.

   NOTE: The ccws in IPL1 are defined by the architecture to be format 0.
3. Execute IPL2. The TIC ccw instruction at the end of the IPL1 channel program will begin the execution of the IPL2 channel program. IPL2 is stage-2 of the boot process and will contain a larger channel program than IPL1. The point of IPL2 is to find and load either the operating system or a small program that loads the operating system from disk. At the end of this step all or some of the real operating system is loaded into memory and we are ready to hand control over to the guest operating system. At this point the guest operating system is entirely responsible for loading any more data it might need to function.

   NOTE: The IPL2 channel program might read data into memory location `0x0` thereby overwriting the IPL1 psw and channel program. This is ok as long as the data placed in location `0x0` contains a psw whose instruction address points to the guest operating system code to execute at the end of the IPL/boot process.

   NOTE: The ccws in IPL2 are defined by the architecture to be format 0.
4. Start executing the guest operating system. The psw that was loaded into memory location `0x0` as part of the ipl process should contain the needed flags for the operating system we have loaded. The psw's instruction address will point to the location in memory where we want to start executing the operating system. This psw is loaded (via LPSW instruction) causing control to be passed to the operating system code.

In a non-virtualized environment this process, handled entirely by the hardware, is kicked off by the user initiating a "Load" procedure from the hardware management console. This "Load" procedure crafts a special "Read IPL" ccw in memory location 0x0 that reads IPL1. It then executes this ccw thereby kicking off the reading of IPL1 data. Since the channel program from IPL1 will be written immediately after the special "Read IPL" ccw, the IPL1 channel program will be executed immediately (the special read ccw has the chaining bit turned on). The TIC at the end of the IPL1 channel program will cause the IPL2 channel program to be executed automatically. After this sequence completes the "Load" procedure then loads the psw from `0x0`.

6.20.2 How this all pertains to QEMU (and the kernel)

In theory we should merely have to do the following to IPL/boot a guest operating system from a DASD device:

1. Place a "Read IPL" ccw into memory location `0x0` with chaining bit on.
2. Execute channel program at `0x0`.
3. LPSW `0x0`.

However, our emulation of the machine's channel program logic within the kernel is missing one key feature that is required for this process to work: non-prefetch of ccw data.

When we start a channel program we pass the channel subsystem parameters via an ORB (Operation Request Block). One of those parameters is a prefetch bit. If the bit is on then the vfio-ccw kernel driver is allowed to read the entire channel program from guest memory before it starts executing it. This means that any channel commands that read additional channel commands will not work as expected because the newly read commands will only exist in guest memory and NOT within the kernel's channel subsystem memory. The kernel vfio-ccw driver currently requires this bit to be on for all channel programs. This is a problem because the IPL process consists of transferring control from the "Read IPL" ccw immediately to the IPL1 channel program that was read by "Read IPL".

Not being able to turn off prefetch will also prevent the TIC at the end of the IPL1 channel program from transferring control to the IPL2 channel program.

Lastly, in some cases (the zipl bootloader for example) the IPL2 program also transfers control to another channel program segment immediately after reading it from the disk. So we need to be able to handle this case.

6.20.3 What QEMU does

Since we are forced to live with prefetch we cannot use the very simple IPL procedure we defined in the preceding section. So we compensate by doing the following.

1. Place "Read IPL" ccw into memory location `0x0`, but turn off chaining bit.
2. Execute "Read IPL" at `0x0`.

   So now IPL1's psw is at `0x0` and IPL1's channel program is at `0x08`.
3. Write a custom channel program that will seek to the IPL2 record and then execute the READ and TIC ccws from IPL1. Normally the seek is not required because after reading the IPL1 record the disk is automatically positioned to read the very next record which will be IPL2. But since we are not reading both IPL1 and IPL2 as part of the same channel program we must manually set the position.
4. Grab the target address of the TIC instruction from the IPL1 channel program. This address is where the IPL2 channel program starts.

   Now IPL2 is loaded into memory somewhere, and we know the address.
5. Execute the IPL2 channel program at the address obtained in step #4.

   Because this channel program can be dynamic, we must use a special algorithm that detects a READ immediately followed by a TIC and breaks the ccw chain by turning off the chain bit in the READ ccw. When control is returned from the kernel/hardware to the QEMU bios code we immediately issue another start subchannel to execute the remaining TIC instruction. This causes the entire channel program (starting from the TIC) and all needed data to be refetched thereby stepping around the limitation that would otherwise prevent this channel program from executing properly.

   Now the operating system code is loaded somewhere in guest memory and the psw in memory location `0x0` will point to entry code for the guest operating system.

6. LPSW `0x0`

   LPSW transfers control to the guest operating system and we're done.

6.21 Modelling a clock tree in QEMU

6.21.1 What are clocks?

Clocks are QOM objects developed for the purpose of modelling the distribution of clocks in QEMU.

They allow us to model the clock distribution of a platform and detect configuration errors in the clock tree such as badly configured PLL, clock source selection or disabled clock.

The object is *Clock* and its QOM name is `clock` (in C code, the macro `TYPE_CLOCK`).

Clocks are typically used with devices where they are used to model inputs and outputs. They are created in a similar way to GPIOs. Inputs and outputs of different devices can be connected together.

In these cases a Clock object is a child of a Device object, but this is not a requirement. Clocks can be independent of devices. For example it is possible to create a clock outside of any device to model the main clock source of a machine.

Here is an example of clocks:

```
+---------+      +----------------------+   +--------------+
| Clock 1 |      |       Device B       |   |   Device C   |
|         |      | +-------+  +-------+ |   | +-------+    |
|         |>>-+-->>|Clock 2|  |Clock 3|>>--->>|Clock 6|    |
+---------+   |  | | (in)  |  | (out) | |   | | (in)  |    |
              |  | +-------+  +-------+ |   | +-------+    |
              |  |            +-------+ |   +--------------+
              |  |            |Clock 4|>>
              |  |            | (out) | |   +--------------+
              |  |            +-------+ |   |   Device D   |
              |  |            +-------+ |   | +-------+    |
              |  |            |Clock 5|>>--->>|Clock 7|    |
              |  |            | (out) | |   | | (in)  |    |
              |  |            +-------+ |   | +-------+    |
              |  +----------------------+   |              |
              |                             | +-------+    |
              +----------------------------->>|Clock 8|    |
                                            | | (in)  |    |
                                            | +-------+    |
                                            +--------------+
```

Clocks are defined in the `include/hw/clock.h` header and device related functions are defined in the `include/hw/qdev-clock.h` header.

6.21.2 The clock state

The state of a clock is its period; it is stored as an integer representing it in units of 2^{-32} ns. The special value of 0 is used to represent the clock being inactive or gated. The clocks do not model the signal itself (pin toggling) or other properties such as the duty cycle.

All clocks contain this state: outputs as well as inputs. This allows the current period of a clock to be fetched at any time. When a clock is updated, the value is immediately propagated to all connected clocks in the tree.

To ease interaction with clocks, helpers with a unit suffix are defined for every clock state setter or getter. The suffixes are:

- `_ns` for handling periods in nanoseconds
- `_hz` for handling frequencies in hertz

The 0 period value is converted to 0 in hertz and vice versa. 0 always means that the clock is disabled.

6.21.3 Adding a new clock

Adding clocks to a device must be done during the init method of the Device instance.

To add an input clock to a device, the function `qdev_init_clock_in()` must be used. It takes the name, a callback and an opaque parameter for the callback (this will be explained in a following section). Output is simpler; only the name is required. Typically:

```
qdev_init_clock_in(DEVICE(dev), "clk_in", clk_in_callback, dev);
qdev_init_clock_out(DEVICE(dev), "clk_out");
```

Both functions return the created Clock pointer, which should be saved in the device's state structure for further use.

These objects will be automatically deleted by the QOM reference mechanism.

Note that it is possible to create a static array describing clock inputs and outputs. The function `qdev_init_clocks()` must be called with the array as parameter to initialize the clocks: it has the same behaviour as calling the `qdev_init_clock_in/out()` for each clock in the array. To ease the array construction, some macros are defined in `include/hw/qdev-clock.h`. As an example, the following creates 2 clocks to a device: one input and one output.

```
/* device structure containing pointers to the clock objects */
typedef struct MyDeviceState {
    DeviceState parent_obj;
    Clock *clk_in;
    Clock *clk_out;
} MyDeviceState;

/*
 * callback for the input clock (see "Callback on input clock
 * change" section below for more information).
 */
static void clk_in_callback(void *opaque);

/*
 * static array describing clocks:
 * + a clock input named "clk_in", whose pointer is stored in
 *   the clk_in field of a MyDeviceState structure with callback
 *   clk_in_callback.
 * + a clock output named "clk_out" whose pointer is stored in
 *   the clk_out field of a MyDeviceState structure.
 */
static const ClockPortInitArray mydev_clocks = {
    QDEV_CLOCK_IN(MyDeviceState, clk_in, clk_in_callback),
    QDEV_CLOCK_OUT(MyDeviceState, clk_out),
    QDEV_CLOCK_END
};

/* device initialization function */
```

(continues on next page)

(continued from previous page)

```
static void mydev_init(Object *obj)
{
    /* cast to MyDeviceState */
    MyDeviceState *mydev = MYDEVICE(obj);
    /* create and fill the pointer fields in the MyDeviceState */
    qdev_init_clocks(mydev, mydev_clocks);
    [...]
}
```

An alternative way to create a clock is to simply call `object_new(TYPE_CLOCK)`. In that case the clock will neither be an input nor an output of a device. After the whole QOM hierarchy of the clock has been set `clock_setup_canonical_path()` should be called.

At creation, the period of the clock is 0: the clock is disabled. You can change it using `clock_set_ns()` or `clock_set_hz()`.

Note that if you are creating a clock with a fixed period which will never change (for example the main clock source of a board), then you'll have nothing else to do. This value will be propagated to other clocks when connecting the clocks together and devices will fetch the right value during the first reset.

6.21.4 Retrieving clocks from a device

`qdev_get_clock_in()` and `dev_get_clock_out()` are available to get the clock inputs or outputs of a device. For example:

```
Clock *clk = qdev_get_clock_in(DEVICE(mydev), "clk_in");
```

or:

```
Clock *clk = qdev_get_clock_out(DEVICE(mydev), "clk_out");
```

6.21.5 Connecting two clocks together

To connect two clocks together, use the `clock_set_source()` function. Given two clocks `clk1`, and `clk2`, `clock_set_source(clk2, clk1);` configures `clk2` to follow the `clk1` period changes. Every time `clk1` is updated, `clk2` will be updated too.

When connecting clock between devices, prefer using the `qdev_connect_clock_in()` function to set the source of an input device clock. For example, to connect the input clock `clk2` of `devB` to the output clock `clk1` of `devA`, do:

```
qdev_connect_clock_in(devB, "clk2", qdev_get_clock_out(devA, "clk1"))
```

We used `qdev_get_clock_out()` above, but any clock can drive an input clock, even another input clock. The following diagram shows some examples of connections. Note also that a clock can drive several other clocks.

```
+------------+  +--------------------------------------------------+
|  Device A  |  |                   Device B                       |
|            |  |               +---------------------+            |
|            |  |               |       Device C      |            |
|  +-------+ |  | +-------+     | +-------+ +-------+ |  +-------+ |
|  |Clock 1|>>-->>|Clock 2|>>+-->>|Clock 3| |Clock 5|>>>>|Clock 6|>>
|  | (out) | |  | | (in)  |  |  | | (in)  | | (out) | |  | (out) | |
```

(continues on next page)

(continued from previous page)

```
|  +--------+ |  | +--------+  |  | +--------+ +--------+ |  +--------+ |
+-------------+  |             |  +-----------------------+             |
                 |             |                                        |
                 |             |  +----------------+                    |
                 |             |  |    Device D    |                    |
                 |             |  | +--------+     |                    |
                 |             +-->>|Clock 4|      |                    |
                 |                | | (in)  |      |                    |
                 |                | +--------+     |                    |
                 |                +----------------+                    |
                 +------------------------------------------------------+
```

In the above example, when *Clock 1* is updated by *Device A*, three clocks get the new clock period value: *Clock 2*, *Clock 3* and *Clock 4*.

It is not possible to disconnect a clock or to change the clock connection after it is connected.

6.21.6 Unconnected input clocks

A newly created input clock is disabled (period of 0). This means the clock will be considered as disabled until the period is updated. If the clock remains unconnected it will always keep its initial value of 0. If this is not the desired behaviour, `clock_set()`, `clock_set_ns()` or `clock_set_hz()` should be called on the Clock object during device instance init. For example:

```
clk = qdev_init_clock_in(DEVICE(dev), "clk-in", clk_in_callback,
                         dev);
/* set initial value to 10ns / 100MHz */
clock_set_ns(clk, 10);
```

6.21.7 Fetching clock frequency/period

To get the current state of a clock, use the functions `clock_get()`, `clock_get_ns()` or `clock_get_hz()`.

It is also possible to register a callback on clock frequency changes. Here is an example:

```
void clock_callback(void *opaque) {
    MyDeviceState *s = (MyDeviceState *) opaque;
    /*
     * 'opaque' is the argument passed to qdev_init_clock_in();
     * usually this will be the device state pointer.
     */

    /* do something with the new period */
    fprintf(stdout, "device new period is %" PRIu64 "ns\n",
                    clock_get_ns(dev->my_clk_input));
}
```

6.21.8 Changing a clock period

A device can change its outputs using the `clock_update()`, `clock_update_ns()` or `clock_update_hz()` function. It will trigger updates on every connected input.

For example, let's say that we have an output clock *clkout* and we have a pointer to it in the device state because we did the following in init phase:

```
dev->clkout = qdev_init_clock_out(DEVICE(dev), "clkout");
```

Then at any time (apart from the cases listed below), it is possible to change the clock value by doing:

```
clock_update_hz(dev->clkout, 1000 * 1000 * 1000); /* 1GHz */
```

Because updating a clock may trigger any side effects through connected clocks and their callbacks, this operation must be done while holding the qemu io lock.

For the same reason, one can update clocks only when it is allowed to have side effects on other objects. In consequence, it is forbidden:

- during migration,
- and in the enter phase of reset.

Note that calling `clock_update[_ns|_hz]()` is equivalent to calling `clock_set[_ns|_hz]()` (with the same arguments) then `clock_propagate()` on the clock. Thus, setting the clock value can be separated from triggering the side-effects. This is often required to factorize code to handle reset and migration in devices.

6.21.9 Aliasing clocks

Sometimes, one needs to forward, or inherit, a clock from another device. Typically, when doing device composition, a device might expose a sub-device's clock without interfering with it. The function `qdev_alias_clock()` can be used to achieve this behaviour. Note that it is possible to expose the clock under a different name. `qdev_alias_clock()` works for both input and output clocks.

For example, if device B is a child of device A, `device_a_instance_init()` may do something like this:

```
void device_a_instance_init(Object *obj)
{
    AState *A = DEVICE_A(obj);
    BState *B;
    /* create object B as child of A */
    [...]
    qdev_alias_clock(B, "clk", A, "b_clk");
    /*
     * Now A has a clock "b_clk" which is an alias to
     * the clock "clk" of its child B.
     */
}
```

This function does not return any clock object. The new clock has the same direction (input or output) as the original one. This function only adds a link to the existing clock. In the above example, object B remains the only object allowed to use the clock and device A must not try to change the clock period or set a callback to the clock. This diagram describes the example with an input clock:

```
+--------------------------+
|        Device A          |
|         +---------------+ |
|         |   Device B    | |
|         | +-------+     | |
>>"b_clk">>>| "clk" |     | |
|   (in)    | |  (in)  |     | |
```

(continues on next page)

(continued from previous page)

```
|          | +-------+    | |
|          +--------------+ |
+---------------------------+
```

6.21.10 Migration

Clock state is not migrated automatically. Every device must handle its clock migration. Alias clocks must not be migrated.

To ensure clock states are restored correctly during migration, there are two solutions.

Clock states can be migrated by adding an entry into the device vmstate description. You should use the `VMSTATE_CLOCK` macro for this. This is typically used to migrate an input clock state. For example:

```
MyDeviceState {
    DeviceState parent_obj;
    [...] /* some fields */
    Clock *clk;
};

VMStateDescription my_device_vmstate = {
    .name = "my_device",
    .fields = (VMStateField[]) {
        [...], /* other migrated fields */
        VMSTATE_CLOCK(clk, MyDeviceState),
        VMSTATE_END_OF_LIST()
    }
};
```

The second solution is to restore the clock state using information already at our disposal. This can be used to restore output clock states using the device state. The functions `clock_set[_ns|_hz]()` can be used during the `post_load()` migration callback.

When adding clock support to an existing device, if you care about migration compatibility you will need to be careful, as simply adding a `VMSTATE_CLOCK()` line will break compatibility. Instead, you can put the `VMSTATE_CLOCK()` line into a vmstate subsection with a suitable `needed` function, and use `clock_set()` in a `pre_load()` function to set the default value that will be used if the source virtual machine in the migration does not send the clock state.

Care should be taken not to use `clock_update[_ns|_hz]()` or `clock_propagate()` during the whole migration procedure because it will trigger side effects to other devices in an unknown state.

Index