ITC QEMU Manual

NASA IV&V ITC Team

Version 0.1

Table of Contents

Introduction

QEMU (Quick Emulator) is a generic open source machine emulator and virtualizer. The ITC team has explored the viability of using QEMU for spacecraft simulation environments as an alternative to the commercial Wind River Simics product used extensively on previous missions. Initially, QEMU was explored as a simple feasibility study running basic applications on both the PPC and SPARC LEON3 architectures. The WFIRST mission will be the first pilot project to utilize QEMU as the core emulation engine of the ITC developed WFIRST Integrated Simulation Platform (WISP). The main driver for using the WFIRST mission as a pilot project was the lack of availability of the GR740 LEON4 board in Simics or via other alternatives, which was a perfect opportunity to evaluate QEMU since modeling had to start from scratch and there was no opportunity for reuse.

Purpose

A major goal of the pilot project is the generation of this manual, in which the primary purpose is to serve as documentation for future QEMU modeling efforts and training for ITC team members. Although written from experience during pilot project development, which utilizes a LEON4 (SPARCv8) architecture, this document will remain generic, containing basic QEMU runtime and modeling info applicable to any mission and/or architecture.

Although this manual can be read standalone, it is primarily meant to be a hands-on interactive training to increase learning. In order to support hands-on training, this repository contains a number of useful features in addition to the various manuals that are produced as build artifacts.

Features

To simplify development, the QEMU repository is included as a git submodule within this repository (`qemu`) for easy source code access along with a lightweight Alpine Linux image for convenience (`support/alpine.img`). This provides a complete package that the user can use while learning QEMU before moving to project specific repositories, which may differ. It also ensures that the proper version of QEMU supported by the is manual is checked out.

NOTE	QEMU is actively developed with evolving APIs. This document assumes the tagged **v4.2.0** for development.

This repository also includes the necessary support to generate a development virtual machine (VM) and a Docker based build image. Of course the user may also install all build dependencies and work directly on the host if desired, but these tools ensure a consistent QEMU build and development environment.

The QEMU development VM is automatically generated using Vagrant. The QEMU VM is an Ubuntu MATE distribution with all necessary development prerequisite packages installed, along with other useful utilities, such as Visual Studio Code, Git, Python, the `ghex` hex viewer, GDB multiarch debugger, and numerous others. The QEMU repository is also cloned directly in the VM. In order to generate the development QEMU VM, simply install Vagrant and run the following commands from within this repository:

```
$ cd support/
$ vagrant up
```

If the user chooses to run directly on the host, there are a few options available: using the included Docker build image, running the Ansible provisioning scripts directly, or manually installing all dependencies. Creating the Docker build image is beyond the scope of this manual, but more advanced users can customize and manually generate the image. The Docker image entrypoint is a simple build script (`support/docker/build.sh`) that passes arguments to the QEMU build system. The `support/docker-build.sh` helper script was created to simplify building QEMU via the Docker image. When this script is executed, it:

1. Prompts the user to automatically generate the Docker build image if it doesn't already exist
2. Automatically create the Docker build cache volume
3. Run the Docker QEMU build container to build QEMU (installed in `build/qemu`)

Challenges

Although QEMU is an active open source project, the following challenges exist when learning to both use and model hardware using the platform:

- In general, QEMU is not well documented, which results in a steep learning curve. Developer's must pull info from source code, sparse documents in the source tree, the project Wiki, and tidbits of info available throughout the internet in the forms of presentations, Stack Overflow articles, GitHub repositories, etc.
- QEMU is under active development, which is generally positive, however this leads to frequent updates, API changes, etc. The ITC team evaluates each release to determine whether to upgrade for bug fixes, enhancements, etc.
- QEMU has an overlap in APIs (legacy vs modern). The legacy APIs are still in use for backwards compatibility so it is often difficult to determine the proper API to use or if the modern APIs are complete. On several occasions new APIs or features are committed incomplete.
- Even for simple QEMU usage, the number of command line arguments available in the man pages is overwhelming, as devices can be dynamically added to *machine* (board) models, devices models can be *connected* to host devices (i.e.; character devices, network TAP devices, etc.), etc.
- Although processor models and basic boards exist for both PPC and SPARC (LEON3), the desired boards used by NASA do not exist and require development. For Simics based projects, we typically start with a board model and the primary effort is on modeling cPCI cards in the flight computer chasis. QEMU requires a more difficult and low-level starting point.
- Machine configurations, which will be discussed in detail in a future section, are mainly static, although it seems configurable machines are in the QEMU roadmap.

Resources

This section describes general QEMU resources. Since QEMU documentation is limited and

resources are spread throughout code and the internet with random talks and topics, resources for more specific aspects of QEMU will be listed in the appropriate sections of this document.

- QEMU User Manual
- QEMU Wiki
- QEMU Mailing Lists
- IRC (#qemu on irc.oftc.net)
- Bug Tracker
- QEMU Source (see the docs directory)
- KVM Forums (KVM centric but occassionally has general QEMU info)
- WIKIBOOKS QEMU
- archlinux QEMU wiki

Building

This chapter introduces the QEMU build system and provides a basic overview for building the software. User's are expected to learn how to build QEMU prior to following along with other sections of this manual. This is necessary due to most Linux distributions containing ancient versions of QEMU in the default repositories (for example, Ubuntu includes v2.11.1).

Build System

The QEMU build system utilizes the standard `configure` and `make` utilities. It appears to be modeled closely after the Linux kernel build system (Kbuild) due to numerous similarities. Recently, in QEMU v4.0, the Kconfig language was added, although it is only partially complete. The Kconfig language organizes all configuration options into a tree structure, with various objects and their dependencies enabled/disabled based on selected items in the tree. The goal is to be able to completely customize QEMU builds, similar to the Linux kernel, by selecting individual components, hardware devices, etc. to include in the build. For example, all PCI devices can be disabled from the build by simply de-selecting the PCI parent node. Although Kconfig has been integrated into the build, the following features are incomplete and are expected in fugure QEMU releases:

- Kconfig front-ends unimplemented, implying unable to run `make config` or any of its variants
- Building modules partially implemented, but not integrated with Kconfig (see Modules)

NOTE	Once complete, the QEMU module system may simplify separating core ITC devices and libraries from the main QEMU executable.

Each QEMU supported architecture is compiled into a separate binary. For example, a 32-bit PPC system is emulated using the `qemu-system-ppc` executable. These executables are standalone, meaning QEMU statically links everything, which makes it easy to install the binary to any desired location. QEMU also includes numerous utilities that are built, such as `qemu-img`, to create/convert between various image formats.

References:

- Cody, Jeff. New to QEMU: A Developer's Guide to Contributing. KVM 2014. [slides][video]

Dependencies

QEMU is complex software that serves many purposes and includes many features to allow the guest to interact with the host, including USB devices, TPM, smartcards, etc. Due to this the dependency list is large and beyond the scope of this document. The QEMU Hosts wiki page includes basic instructions, including dependencies, however, it seems to be incomplete and outdated. Also, the list of required build dependencies depend on which features are enabled via the `configure` script.

If building on an Ubuntu host, the simplest way to get started is to enable the source repository and install all build dependencies via `apt`.

/etc/apt/sources.list

```
deb http://us.archive.ubuntu.com/ubuntu/ bionic main restricted
# deb-src http://us.archive.ubuntu.com/ubuntu/ bionic main restricted ①
```

① Uncomment the source repository line to enable installing source packages

Once the source repository has been added, the following command will install the QEMU build dependencies.

```
$ sudo apt update
$ sudo apt build-dep qemu
$ sudo apt install bison flex
```

NOTE It appears the Ubuntu QEMU build dependencies are incomplete. All of the installed packages allow building QEMU with default config options, however, the build will fail due to missing dependencies if various additional libraries are not installed. For example, the `libsdl2-dev` package is required if graphical SDL support is required.

Building

Once all of the build dependencies have been installed, QEMU can be cloned directly from the git repository (https://git.qemu.org/git/qemu) and built. For convenience, it is also included as a submodule within this repository. The following command can be used to build QEMU. This command will build QEMU with all default configure options. By default, in addition to support libraries and utilities, it will also build QEMU executables for all supported architectures.

NOTE It is recommended to perform an out of source build.

```
$ mkdir build
$ cd build
$ /path/to/qemu/configure
$ make
$ make install
```

The QEMU build generates numerous executables, depending on the chosen configuration options. Along with general utilities, a separate binary is generated for each chosen architecture: `qemu-system-<arch>`.

For ITC projects, it is recommended to view the configure script help (`configure --help`) to view all available options. The build time can be drastically reduced by disabling all features that are not required for a project. At a minimum, it is recommended to enable only the specific targets desired for the project using the `--target-list` config option. Many other options irrelevant to ITC simulations can also be disabled. For example, for the WFIRST WISP simulator, there are over 30 items disabled in the configure script, related to items such as display, KVM, disk image support, docs, etc.

NOTE	The ITC team has found that there are sometimes inconsistencies w/ the configure script help output and what can actually be disabled. The configure script should be manually inspected to further eliminate non-required dependencies.

It is beyond the scope of this document to discuss all available configure options, however, the following table highlights some of the more interesting items for the ITC team:

Table 1. Useful Configure Options

Option	Description
--prefix=PREFIX	install prefix [/usr/local]
--target-list=LIST	set target list (default: build everything) (`--target-list=i386-softmmu,sparc-softmmu`)
--enable-debug	enable common debug options (NOTE: this will significantly slow down execution)
--disable-werror	disable compilation abort on warning
--enable-trace-backends=B	enable additional trace backends
--disable-blobs	disable installing provided firmware blobs (NOTE: typically not necessary for ITC simulations due to custom bootloaders)
--enable-plugins	enable plugins via shared library loading (NOTE: currently only partially implemented but watch for future support)
--enable-docs	enable/disable building documentation
--enable-tools	enable building tools (qemu-img, etc.)

Source Code

This section attempts to highlight key files and directories in the QEMU source code. These tables serve as a quick overview or reference, with more detailed information provided in individual sections of this document related to the listed files/directories. It should be noted that QEMU source related to hardware models follows a convention, with hardware models grouped by type (i.e.; gpio, char, i2c, etc.). Adhering to this convention is especially important if planning to submit changes to upstream QEMU, however, the ITC team has preferred to try to keep ITC developed models in self-contained directories. This is advantageous as it simplifies future merges with updated QEMU versions, however, the disadvantage is that ITC developed code cannot be submitted to the upstream QEMU project without refactoring.

Due to lacking QEMU documentation, the source is usually the best way to understand QEMU code and operation. Unfortunately, header comments are inconsistent. Some core API headers are well documented while others are simply lists of function prototypes and the user has to look at the implementation to understand the function.

NOTE	It should be mentioned that QEMU heavily utilizes the GLib general purpose utility library, so developers may need to become familiar with it.

The following tables highlight some of the important top-level QEMU files and directories to get started with development. The listed files are ones that will most likely be edited by the ITC team when developing a simulation. Some of the termininology will be described later in this document.

NOTE	QEMU includes a complex method of auto-generating code from json files using `QAPI`. This is mainly used for marshalling by the QEMU machine protocol (QMP). It is mentioned here for awareness if unable to search and find a desired definition it may exist in a json file.

References:

- Hajnoczi, Stefan. QEMU Code Overview.

Table 2. Important QEMU Build Files

File	Description
default-configs/	Define which machines to enable for each architecture
hw/Kconfig	Top-level hardware configuration
hw/Makefile.objs	Add additional device directories
Makefile.objs	Add custom trace-events

Table 3. Important QEMU Source Directories and Files

File	Description
chardev/	Character device common code and backends
docs/	QEMU documentation, usually a good reference but topics are sparse
hw/	Contains all hardware models (machines, devices, core APIs, etc.) organized into subdirectories
include/	Header files
io/	Various asynchronous IO channel wrappers (built on top of the glib IO Channel API)
monitor/	QEMU monitor related files (HMP and QMP)
python/	Python package for QEMU interaction
qapi/	QMP command definitions and data types (defined in *.json files)
qobject/	QAPI related json parsing utilities
qom/	QEMU object model (QOM) related files
target/	CPU instruction translators for each supported architecture
util/	General purpose utilities
cpus.c	Top-level CPU emulation (instructions, timing, etc.)
vl.c	QEMU main loop

Table 4. Important QEMU Command (HMP/QMP) Files

File	Description
hmp-commands.hx	Define custom HMP commands
include/monitor/hmp.h	Prototypes for all HMP commands (must begin with hmp_)
qapi/Makefile.objs	Add custom QMP json files
qapi/qapi-schema.json	Main QAPI file that includes all other json files

Usage

This chapter describes general QEMU usage, including starting, stopping, and monitoring the emulation. It is beyond the scope of this manual to describe every aspect of QEMU and the different modes of operation, such as user mode emulation and KVM. This manual will strictly focus on ITC relevant usage information.

This section assumes QEMU has been built or installed from another source. This guide will utilize the i386 architecture since that probably has the most complete machine models and available documentation.

NOTE	The Ubuntu 18.04 repositories contain QEMU **v2.11.1**, which is extremely old, so the examples provided in this guide most likely will not run with that version without modification. It is recommended to use the version included with this training.

Image Creation

Numerous online resources exist describing QEMU image creation in detail. This guide briefly illustrates the process in the simplest possible manner, outlining the basic steps used to create the included lightweight `alpine.img` image. The WIKIBOOKS QEMU Images page is a good reference.

NOTE	Although beyond the scope of this guide, it should be noted that the powerful `qemu-img` utility can be used to convert between various image formats (VDI, VMDK, etc.) and also perform other management tasks.

1. Create disk image

```
$ qemu-img create -f qcow2 alpine.img 256M
```

2. Install guest OS (i386, 512MB, networking)

```
$ qemu-system-i386 -m 512 -nic user -boot d -cdrom alpine-virt-3.11.5-x86.iso
alpine.img
```

NOTE	The newer `-nic user` command line argument is an equivalent shorthand to the legacy, but still valid long `-netdev user,id=n1,ipv6=off -device e1000,netdev=n1` argument.

After the guest OS is installed to the `alpine.img` disk image, it is ready to run, similar to VirtualBox. See the section on running the guest OS in this document for details.

Running QEMU

The sample lightweight Alpine Linux image `alpine.img` can be run using the following command,

which executes the default i386 machine with 512MB RAM and networking capability. The following sections describe various command line parameters and options for interacting with the guest.

```
$ qemu-system-i386 -m 512 -nic user alpine.img
```

The default username/password is root/qemu.

By default, QEMU runs in graphical mode, however, that is of limited use to ITC systems since they typically only contain debug UARTs for interaction. In any case, it is useful to know of the various arguments related to graphical and non-graphical modes. Running QEMU with the `-nographic` option runs in non-graphics (console) mode. It is beyond the scope of this document to describe all of the various modes and how the graphics frontends/backends are tied together, however, the `-nographic` option is the simplest, as it not only disables graphic output but it also changes the serial port destination automatically, as opposed to the `-display none` argument, which just disables video output. The following section describe these modes.

Graphical Mode

QEMU includes numerous display options when running in graphical mode, selected via the `-display` command line argument, such as SDL, GTK, VNC, and SPICE. The default graphical display window depends on configuration options enabled during compilation. To complicate matters, the default configuration option state when not specified via the command line depends on libraries installed on the host. For example, if the SDL development libraries are installed, QEMU will be compiled with SDL support by default. Certain QEMU displays will *grab* the mouse when the user clicks in the graphical window. The following table lists common keybindings when working with graphical QEMU displays, including how to *release* the mouse.

Table 5. Common Graphical Mode Keybindings

Key	Description
`Ctrl-Alt`	Release *grabbed* or *captured* mouse in the VM
`Ctrl-Alt-f`	Enter/exit fullscreen mode
`Ctrl-Alt-n`	Switch to virtual console 'n' (1=guest OS, 2=monitor, 3=serial, ...)
`Ctrl-Up`, `Ctrl-Down`, `Ctrl-PgUp`, `Ctrl-PgDn`	Scroll virtual console window

Non-Graphical Mode

When running in non-graphical mode (i.e. `-nographic` option), QEMU behaves like a simple command line application. By default, the QEMU monitor is multiplexed with the console unless explicitly redirected elsewhere, as will be discussed in future sections. The monitor will be discussed later in this document.

Table 6. Common Non-Graphical Mode Keyboard Shortcuts

Key	Description
`Ctrl-a h`	Print help
`Ctrl-a c`	Rotate between front-ends connected to multiplexer (i.e.; switch between console and monitor, etc.)
`Ctrl-a x`	Exit emulator

Command Line Arguments

QEMU is a powerful and very configurable tool, which leads to a large number of command line arguments. The man page or user documentation is the recommended resource, however, this section simply highlights some of the common or useful commands from an ITC perspective. Most likely, the kernel-based virtual machine (KVM) options will not be applicable to ITC since most encountered architectures are not x86. Also, ITC developed emulations will generally not be configurable and will be a *fixed* board/machine.

It should also be mentioned that GUI tools exist (libvirt, AQEMU, QtEmu, etc.), that could possibly simplify the usage of QEMU, although it is beyond the scope of this document to discuss in detail. Many of the GUI tools are KVM and management focused, which is outside of the ITC team's use case. User's looking for a command line alternative are encouraged to investigate these utilities.

QEMU arguments have changed throughout releases, so this guide focuses on the newer method using the single `-object` or `-device` arguments. Creation of any object in QEMU from the command line adheres to the following pattern: `-object typename,[prop1=value1,…]`. Various terms and concepts required to discuss the command line in more detail are expanded on in future secions of this document.

Along with the typical `--help` argument, QEMU allows further help for details of specific command line arguments. The typical convention is `-arg ?` or `-arg help`. This will be explained further in the following text.

QEMU system binaries exist for all supported architectures. When run, QEMU instantiates a *machine* of that architecture. Each architecture can define a default machine to use when not specified from the command line, as was the case in the previous example. The QEMU machine is selected using the `-machine` or `-M` command line argument. The *help* option can be used to inspect the available machines for a specific architecture. Notice the default machine is shown by the `(default)` text in the machine description.

```
$ qemu-system-i386 -machine ?
```

Machines are essentially just a special case of a device, therefore, machines also contain properties. QEMU is not currently very flexible in allowing a user to define custom machine properties (i.e.; EEPROM size, etc.). Most of the pre-defined and hard-coded machine properties are of limited use for ITC. Also, QEMU machines are meant to be generic, similar to real HW, where a user can *plug'n'play* components, such as CPU, SMP, etc. Although useful for QEMU, for ITC emulations the configuration is mainly static, so the CPU type, for example, will be defined in the machine, which means many of the parameters such as `-cpu` or `-smp` are not useful in our case. The following

command will list all CPU types for a given architecture:

```
$ qemu-system-i386 -cpu ?
```

The amount of machine RAM is defined using the `-m` command line argument. The user can specify a non-negative integer followed by an optional suffix, such as k, M, or G for kilobytes, megabytes, and gigabytes, respectively. The unites default to megabytes if not specified.

```
$ qemu-system-i386 -m ?
```

NOTE Although a user can configure machines from the command line, the arguments are generic for all machine types, however, the specific machine can further limit acceptable values. For example, a machine may limit the maximum amount of RAM further to 512MB.

As previously mentioned, the long form of the `-nic user` argument is `-netdev user,id=n1,ipv6=off -device e1000,netdev=n1`. The example illustrates device creation. The command argument `-netdev user,id=n1,ipv6=off` creates a user mode host network *backend* that doesn't require admin privileges. The `id` property, which can be given any arbitrary identifier, is used in the device argument to associate the actual HW model with the specified *backend*. In QEMU, the device model is considered the *frontend*, and the *backend* ties the model to a specific resource. For example, a character device can be configured to use a socket or pty backend to connect the output to the host computer. In this example, the `-device e1000,netdev=n1` argument creates an e1000 HW network device and associates it with the user mode networking *backend* via the `id` property.

NOTE Because the specific hard drive to associate the image with (i.e.; `-hda`) is missing from the command line, it defaults to the first hard drive device defined in the machine.

The following table summarizes some of the common QEMU options most likely of interest to the ITC team. Some of the options, such as related to the QEMU monitor, will be discussed in further detail in future sections of this document.

Table 7. Common QEMU Options

Option	Description
-bios	BIOS filename (ITC can use this to load the SUROM / bootloader)
-kernel	Although this is listed in the Linux specific section, it can be used to load/execute an image
-gdb	Start gdb server
-m	Set guest OS RAM size in megabytes
-machine, -M	Set the QEMU emulated machine type
-monitor	Redirect the QEMU monitor to specified host device
-no-reboot	Exit emulation instead of rebooting

Option	Description
-qmp	Start the QEMU control mode (QMP) monitor
-readconfig	Read command line options and device configuration from *ini* style file
-rtc	Various options for configuring the QEMU guest real-time clock
-s	Open GDB server on TCP port 1234 (shorthand for `-gdb tcp::1234`)
-S	Start CPU in paused state (must continue from monitor)
-trace	Specify tracing (logging) options

Monitor

QEMU includes a monitoring console which allows interaction with QEMU and the running guest machine. The monitor can be accessed via two distinct protocols, both of which provide similar capabilities, but serve different purposes.

- QEMU Human Monitor Protocol (HMP)
- QEMU Machine Protocol (QMP)

NOTE	It will be described later in this document, but HMP/QMP are methods for the ITC team to define custom commands, such as fault injection, etc.

Users are encouraged to run the provided Linux disk image and experiment with various HMP commands to gain familiarity.

Human Monitor Protocol (HMP)

The QEMU Human Monitor Protocol (HMP) is the typical human interface, or console, used to issue commands to interact with QEMU. Through various commands, the monitor allows you to inspect the running guest OS, change removable media and USB devices, connect additional hardware, take screenshots and audio grabs, and control various aspects of the virtual machine.

The monitor can be redirected by using the `-monitor <dev>` command line option. Using `-monitor stdio` will send the monitor to the standard output. Alternatively, the monitor can be run via a local socket: `-monitor tcp:127.0.0.1:55555,server,nowait`. This option creates a server on localhost and sends the monitor stdio over the specified port and is easily accessible via any socket program such as telnet or netcat: `nc localhost 55555`. By default, the monitor is redirected to the graphical virtual console (vc) device when running in graphical mode and the stdio device when running in non-graphical mode.

NOTE	When using a primitive tool such as telnet to interact with the QEMU monitor, the `rlwrap` utility can be used to add readline support (command history, etc.). For example: `rlwrap telnet localhost 5555`.

See the QEMU Monitor page for a complete list of available commands. The `help or ? [cmd]`

command can be used to get interactive help in the monitor. The following list contains a few of the more useful commands for ITC simulations:

Emulation Control

Command	Description
c|cont	Continue emulation (play/resume)
quit	Quit emulation (exit)
stop	Stop emulation (pause)
system_powerdown	Powerdown the system (if supported)
system_reset	Reset the system

Machine Info

Command	Description
info chardevs	Show character devices
info cpus	Show CPU info
info mtree	Show memory tree
info pci	Show PCI info
info qom-tree	Show QOM device model tree
info qtree	Show qdev device tree
info registers	Show CPU registers
info roms	Show ROMS
info status	Show current VM status (running/paused)
info trace-events	Print all trace events and current enable state

Machine Debugging

Command	Description
dump-guest-memory	Dump guest memory to file
gdbserver	Start GDB server on device (`(qemu) gdbserver tcp::1234`)
logfile	Output logs to filename
memsave	Save virtual memory dump to file
o	I/O port write
p|print	Print expression value ($reg for CPU registers) (`(qemu) p $eax`)
pmemsave	Save physical memory dump to file (`(qemu) pmemsave 0x0 0x400 mem.bin`)
sum	Compute checksum of memory region

Command	Description
trace-event	Enable/disable trace event (`(qemu) trace-event memory_region_ops_read on`)
x	Virtual memory dump at address
xp	Physical memory dump at address (`(qemu) xp /10xw 0x0`)

QEMU Machine Protocol (QMP)

QMP is a json based machine level protocol useful for automation, creating additional tools and/or scripts, etc. that need to interact with QEMU. Various backends can be used to access the QMP server, such as TCP: `-qmp tcp:<address>:<port>,server,nowait`. This runs on a local IP address (e.g.localhost) and port so that a simple telnet or socket program can be used to interact with qemu. Although all have not been converted, many QEMU HMP commands have been converted to QMP. The general *modern* QEMU pattern is to write the commands in QMP and provide a lightweight HMP implementation that calls the QMP command.

Although it is not neatly bundled, for example as a *pip* package, QEMU includes a core Python library for interacting with a QMP server in the source directory at `python/qemu`. There is also an example QMP shell application, although the ITC team has found it to not be very mature and buggy, at `scripts/qmp`. Eventually these libraries may form the foundation of more mature and robust ITC tools.

NOTE	The ITC will eventually utilize QMP to build more capable front-end GUIs to simplify QEMU usage and provide additional functionality related to NASA simulations.

A complete running example would then be:

```
$ qemu-system-i386 -m 512 -netdev user,id=n1,ipv6=off -device e1000,netdev=n1 -qmp
tcp:localhost:5000,server,nowait alpine.img
```

For detailed info, refer to the QEMU QMP page.

Architecture

This chapter describes high-level QEMU architecture, along with some specific topics relevant to ITC simulations.

Main Loop

The QEMU main loop lives in `vl.c`. This is a large file that contains option definitions and parsing functions, run state control and state machine related functions, and object/machine registration. In a nutshell, the main loop performs the following:

- Parse command line arguments
- Create machine and add to QOM object tree root
- Initialization (CPU, monitor, tracing/logging, graphical windows, etc.)
- Run main loop (`main_loop()`) function
- Cleanup after main loop exits

TODO: main event loop, top/bottom half, async coroutines, etc.

Memory Model

For a detailed overview of the QEMU memory model, see the `docs/devel/memory.rst` file in the QEMU source tree. The main memory related objects in QEMU are `MemoryRegion` and `AddressSpace` objects. In a nutshell, address spaces define memory as seen from the CPU or a device perspective. The `include/exec/address-spaces.h` file contains the global memory and IO address spaces, along with the `get_system_memory` function to get the top-level `MemoryRegion` object. The QEMU memory API models all memory region types using a single `MemoryRegion` type. QEMU memory is modeled as an acyclic graph of `MemoryRegion` objects. Memory regions can be nested for convenience, each memory region is mapped to a specific offset into the parent region. The `include/exec/memory.h` header file includes the memory region API with detailed function comments. The following lists some of the common memory region types:

- RAM (`memory_region_init_ram`)
- MMIO (`memory_region_init_io`)
- ROM (`memory_region_init_rom`)
- IOMMU (`memory_region_init_iommu`)

Memory regions can be mapped into an address space or other memory region using `memory_region_add_subregion`. The memory region is attached to an object and destruction is automatic upon owner object destruction.

Memory regions can be overlapped and assigned a priority to fill in **holes** using `memory_region_add_subregion_overlap`. Priority values are signed and the highest priority wins. All regions default to priority zero.

NOTE	Memory region priorities are useful for *shadow* memory or for filling in don't care regions of unmapped memory.

As will be illustrated further in the device model section, memory mapped IO (MMIO) regions allow read/write callbacks to be handled in user device code. Each device can also apply constraints, such as min/max access size, alignment, endianness, etc. All of the operation parameters are defined in a `MemoryRegionOps` structure.

Timing

TODO: general QEMU timing info (CPU modeling, timers/ptimers, rtc/icount, etc.)

Host Devices

TODO: connecting QEMU to host devices (i.e.; host char devices, networking, etc.)

QEMU APIs

Given the lack of documentation and outdated online information leading to confusion, this page attempts to highlight the various modern QEMU APIs available and which APIs are considered legacy. The legacy APIs are still widely used for backwards compatibility, so new developers must take care using existing models as examples or references. This section is meant to serve as a high level overview, with each of the APIs further demonstrated in future sections while creating sample device models.

References:

- Habkost, Eduardo. An incomplete list of QEMU APIs. Nov 29, 2016.
- Habkost, Eduardo. QEMU internal APIs. FOSDEM 2017. [pdf][video]

QEMU Object Model (QOM)

The QEMU Object Model (QOM) is the foundation of the QEMU object oriented code base. In a nutshell, QOM provides the base class/object primitives with property support. QOM also provides the framework for user defined type registration and object instantiation. The header file is heavily documented with examples (`include/qom/object.h`). All objects are stored in the QOM tree, which determines an objects lifetime. When running an emulation, the full QOM tree can be inspected in the QEMU monitor via the `info qom-tree` command. Note that unless a parent is specifically assigned, QOM objects default to anonymous/unassigned roots. Some QOM features are highlighted below:

- System for dynamically registering types
- Support for single-inheritance of types
- Multiple inheritance of stateless interfaces
- Generic property system for introspection and object/device configuration

QOM started as a generalization of `qdev`, the legacy object API. Today both the `qdev` device tree and backend objects are managed through the QOM object tree.

NOTE	QOM will most likely be of interest to the ITC team for generic or common objects not related to hardware.

NOTE	Depending on properties set and the specific machine configuration, objects may be created dynamically from the command line (`-object typename[,prop1=value1,…]`) or via monitor commands (`object_add`, `object_del`).

References:

- QEMU Wiki. QOM Conventions
- Bonzini, Paolo. QOM exegesis and apocalypse. KVM Forum 2014 [pdf][video]

QEMU Device (qdev)

The QEMU Device (qdev) API is the primitive base API for modeling hardware devices in QEMU. The qdev API is built on top of QOM, providing common functions specific for hardware. Convenience functions and macros are also provided to simplify the underlying QOM property system. The qdev API header is partially documented (`include/hw/qdev-core.h`), which the primary structures being the `DeviceClass`, which inherits the QOM `ObjectClass` and `DeviceState`, which has the QOM `Object` as a parent. Major functionality provided by qdev are highlighted below:

- Device specific object lifecycle, including realize/unrealize and reset
- Save state support via `VMStateDescription`
- Parent and child bus to form a tree hierarchy
- GPIO support (can also be used as IRQs)
- Simplified macro/function QOM property wrappers (`include/hw/qdev-properties.h`)
- Hotplug handling

NOTE	There are numerous legacy references to qdev online that state it is deprecated and was replaced by QOM. Although it is partially true that the legacy qdev has been replaced, it can more accurately be described as qdev exists but the base was rewritten on top of QOM, so from a user perspective qdev still exists and provides the same features, with a different underlying foundation. As an example, the QEMU Features/QOM wiki page refers to the legacy qdev code.

In addition to basic devices, qdev provides an additional `SysBusDevice` structure, which is a subclass of `DeviceState`, to simplify modeling devices that need memory mapped IO (MMIO) support on the QEMU system bus. The main addition of system bus devices are the addition of MMIO (`MemoryRegion` mapped at a specific address) and GPIO/IRQ specific functions that propagate across the bus. The sysbus API also includes helper functions for connecting IRQs, mapping memory, etc. (`include/hw/sysbus.h`).

The entire qdev tree can be inspected in the QEMU monitor via the `info qtree` command. Note that

this command will only show system bus devices and not general QOM objects. It should also be noted that qdev provides the mechanism for device code to register implementations of device types. Machine (or board) code, which is described in another section, uses the qdev API to configure and instantiate the devices specific to that board.

NOTE	The ITC team will most often be developing `SysBusDevice` devices

References:

- Armbruster, Markus. QEMU's new device model qdev. KVM Forum 2010
- Armbruster, Markus. QEMU's device model qdev: Where do we go from here? KVM Forum 2011 [pdf][video]
- Färber, Andreas. Modern QEMU Devices. KVM Forum 2013 [pdf][video]

Machine

In QEMU, machines (also known as boards) are the emulated computers. Machines inherit basic QOM object and are composed of a common set of properties (CPU, RAM size, etc.). The machine code (`include/hw/bords.h`) provides the `MachineState` structure that is filled in from command line arguments and passed to the machine init code. The custom machine code must populate the board with all devices, instantiating and configuring using the QOM and/or qdev APIs. The machine would also allocate the CPU and RAM and perform all memory mapping and IRQ connections. It should be noted that modern QEMU code doesn't have detailed internal device structure knowledge, and only configures the device via defined properties.

NOTE	Unfortunately, machine properties are not currently configurable in QEMU, which means useful ITC configuration parameters such as EEPROM must be either hard-coded or core QEMU code changed to support. The QEMU roadmap proposes a more dynamic machine configuration method for future releases.

VMState

The `VMState`, or migration API, uses a table-based approach to support device state saving/loading. This is similar to the VirtualBox save state functionality, or checkpointing. For ITC modeling, the `VMStateDescription` is created and registered during class initialization for qdev based devices. See the `include/migration/vmstate.h` header for more details.

NOTE	The ITC team should investigate this API more fully in the future if checkpointing support is desired.

References:

- Quintela, Juan. Migration: How to hop from machine to machine without losing state. KVM Forum 2010 [pdf]
- Quintela, Juan. Migration: One year later. KVM Forum 2010 [pdf][video]

QemuOpts

The `QemuOpts` API is a simple abstraction for parsing configuration files and command line arguments and storing the resulting parameters. This may be considered more of an internal API that may not be relevant to most ITC development, however, it is worth mentioning for retrieving parsed values. Previously, command line parameters, such as device properties, were stored as unparsed strings of key value pairs (i.e.; `"key1=value1,key2=value2,"`). The new API passes around `QemuOpts` objects and provides helper functions for retrieving the data. Currently, the API only supports booleans, numbers, sizes (accepts (K)ilo, (M)ega, (G)iga, etc. prefixes), and non-parsed raw strings. See the `include/qemu/option.h` header file for reference.

References:

- Habkost, Eduardo. QEMU APIs: Introduction to QemuOpts. Dec 22, 2016.

QEMU Machine Protocol (QMP)

The QEMU Machine Protocol (QMP) is similar to the QEMU human monitor, except it is meant to be used by machines instead of humans, making it ideal for interfacing/controlling QEMU from an external entity, or adding more advanced capabilities such as a more powerful user interface. QMP uses QAPI, discussed later, for serialization and code generation from the QAPI JSON schemas. The QMP interface includes the following features:

- Lightweight, easy to parse JSON based text data format
- Asynchronous message support (i.e; events)
- Capabilities negotiation

For detailed QMP information, refer to the following files in the QEMU repository:

- `docs/interop/qmp-intro.txt`
- `docs/interop/qmp-spec.txt`
- `docs/devel/writing-qmp-commands.txt`

References:

- QEMU Wiki. QMP
- QEMU User Manual. QEMU User Manual
- Capitulino, Luiz. A Quick Tour of the QEMU Monitor Protocol. KVM Forum 2010 [pdf]

QAPI and QObject

From the documentation, QAPI is a native C API within QEMU which provides management-level functionality to internal/external users. For external users/processes, this interface is made available by a JSON-based wire format for the QEMU Monitor Protocol (QMP) for controlling qemu, as well as the QEMU Guest Agent (QGA) for communicating with the guest. To map Client JSON Protocol interfaces to the native C QAPI implementations, a JSON-based schema is used to define types and function signatures, and a set of scripts is used to generate types, signatures, and

marshaling/dispatch code. The QAPI code generation is documented in the QEMU repository in `docs/devel/qapi-code-gen.txt`. QAPI objects result in a directed acyclic graph, so visitor functions are generated to traverse these graphs (see `include/qapi/visitor.h`).

QObject was added to support QMP and provides a generic QObject base data type with subtypes, including integers, strings, lists, and dictionaries and includes reference counting. It supports all data types defined in the QAPI schema.

TODO: anything else to say here?

References:

- QEMU Wiki. QAPI
- Liguori, Anthony. Code Generation for Fun and Profit. KVM Forum 2011 [pdf][video]

Modeling

This chapter describes QEMU hardware modeling and is most likely the most interesting section for ITC team members.

Building

A few changes are required to add a new device to the QEMU build. For this manual, we will place device code in an ITC directory (`hw/itc`). The first change is to add the new ITC directory to the build by editing the HW Makefile (`hw/Makefile.objs`) and adding the following line:

```
device-dirs-y += itc/
```

Second, assuming the device is in a `hw/itc/sample-device.c` file, the following `Makefile` can be added to the ITC directory in order to build.

Makefile.objs

```
common-obj-y += sample-device.o
```

Now that the necessary build files and device code has been updated QEMU can be built as usual. Note that this `Makefile` adds the device as a common object, which will always be built, regardless of selected devices or architecture. More advanced selections can also be used based on dependencies, such as `common-obj-$(CONFIG_I440FX)` to only build when that specific machine was selected. It is up to the user once comfortable with the QEMU build system to review the dependency trees and make the appropriate selections.

Objects

As highlighted in the QEMU APIs section of this document, the QEMU Object Model (QOM) provides the base functionality of all objects. In reality, base objects are rarely used, as most generic and hardware devices will use qdev, however, they are mentioned here for completeness to help understand the object hierarchy.

Objects in QEMU have both a class type and an object type. Classes are also objects which are *lazy* initialized, meaning the class object is initialized when the first object of that class is created, and only a **single instance** of each class exists. Both classes and objects can have properties. Similar to Simics, the properties allow a device to be configured and/or parameterized.

The QOM object base class header file (`qom/object.h`) is documented with examples. The base class of all objects is defined as `struct ObjectClass` and the base object is `struct Object`. The first member of all QEMU classes or objects is always a pointer to the parent, providing the object hierarchy. Since C guarantees that the first member of a structure begins at byte 0, this allows casting directly between objects and parents. The base `ObjectClass` simply contains a type, an interface list, and properties. Objects can also be declared abstract.

QOM also provides interfaces, which are stateless objects that provide a limited form of multiple inheritance. Objects can be dynamically cast to one if it's interface types and vice versa, which promotes generic code by passing around interfaces instead of specific objects.

Objects are created using the various `object_new_*` API functions, and properties can be added using the various `object_class_property_add_*` and `object_property_add_*` functions. Class properties exist on the class, so they are copied over during object creation so all objects of the class type inherit the properties.

Devices

This section describes device modeling in QEMU. Due to the evolution of QEMU code throughout the years, existing hardware models in QEMU are inconsistent, as devices are not always updated as new APIs, etc. are introduced. This makes it difficult to use existing models as examples of how to properly model hardware using the latest methods, however, QEMU maintains backwards compatibility for many APIs so devices are guaranteed to continue to operate. Modern QEMU devices are modeling using the qdev API, which is built on top of QOM.

TODO: qdev vs sysbus devices

The following QEMU sample device code illustrates the minimal required boilerplate code. All QEMU devices have this similar boilerplate code. This model contains no memory mapped IO, registers, etc. but will be built upon to illustrate each of these concepts. The entire device will first be presented followed by a breakdown of each function.

sample-device.c

```
// minimal qemu sample device
#include "qemu/osdep.h"
#include "hw/sysbus.h"

#define TYPE_SAMPLE_DEVICE "sample"
#define SAMPLE_DEVICE(obj) OBJECT_CHECK(SampleDevice, (obj), TYPE_SAMPLE_DEVICE)

typedef struct {
    DeviceState parent;
    uint32_t count;
} SampleDevice;

static void sample_device_reset(DeviceState *dev) {
    SampleDevice *sample = SAMPLE_DEVICE(dev);
    sample->count = 0;
}

static void sample_device_realize(DeviceState *dev, Error **errp) {
}

static void sample_device_class_init(ObjectClass *klass, void *data) {
    DeviceClass *dc = DEVICE_CLASS(klass);
    dc->desc = "ITC Sample Device";
    dc->realize = sample_device_realize;
    dc->reset = sample_device_reset;
}

static const TypeInfo sample_device_info = {
    .name = TYPE_SAMPLE_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(SampleDevice),
    .class_init = sample_device_class_init,
};

static void sample_device_register_types(void) {
    type_register_static(&sample_device_info);
}

type_init(sample_device_register_types)
```

In order to support QOM type registration and object casting, the QEMU convention is to provide various macros to allow casting between types. QOM defines the `OBJECT_CHECK` macro, which simply wraps the QOM dynamic cast functions. Devices should provide custom type definitions and cast macros built on top of this, typically provided in the header file. This is typical for all QEMU devices. There are other macros, in addition to `OBJECT_CHECK` that deal with casting.

sample-device.c - QOM Macros

```
#define TYPE_SAMPLE_DEVICE "sample"
#define SAMPLE_DEVICE(obj) OBJECT_CHECK(SampleDevice, (obj), TYPE_SAMPLE_DEVICE)
```

Devices typically have state associated with them, such as internal data, register values, memory regions, etc. As previously mentioned related to the QEMU object oriented C development model, the first field of each structure is the parent class, which allows easy casting between objects in the hierarchy. This sample device adds a single `counter` field, which will be used later to demonstrate other functionality.

sample-device.c - Device State

```
typedef struct {
    DeviceState parent;
    uint32_t count;
} SampleDevice;
```

The first item to point out is the QOM type registration code, which is common to all devices. The `type_init` macro in the last line is what actually adds the device type registration function, which in turn calls the QOM device type registration with the specified `TypeInfo`. User's are encouraged to review the `TypeInfo` structure to become familiar with available fields. Devices with different capabilities, such as inheritance from a custom class, etc. may need to fill in additional data. When QEMU starts, all device types are registered.

sample-device.c - Device Type Registration

```
static const TypeInfo sample_device_info = {
    .name = TYPE_SAMPLE_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(SampleDevice),
    .class_init = sample_device_class_init,
};

static void sample_device_register_types(void) {
    type_register_static(&sample_device_info);
}

type_init(sample_device_register_types)
```

The next section of code deals with device and class instantiation. As previously mentioned, QEMU devices consist of a class, which is also an object that is initialized once for each class type, and an object. Also, QEMU devices are initialized in two phases, similar to Simics, known as `init` and `realization`. It is unclear from documentation and existing QEMU model code which method to use, so the ITC team has decided that realization should be used for all cases other than setting up device properties, which will be discussed later. The rationale for adding device properties to init rather than realize is that when device properties are queried from the command line (i.e.; `-device type,?`) the device is initialized, but not realized, and then deleted to support introspection. The

class init function, defined in the `TypeInfo` struct, is called when the device class is created. In this code the device simply sets properties and function pointers for various object specific functions, such as init and reset. The class init function is highlighted below.

sample-device.c - Class Initialization

```
static void sample_device_class_init(ObjectClass *klass, void *data) {
    DeviceClass *dc = DEVICE_CLASS(klass);
    dc->desc = "ITC Sample Device";
    dc->realize = sample_device_realize;
    dc->reset = sample_device_reset;
}
```

The actual object initialization is listed in the following code snippet. For this simple model, the realization function is empty, but it would typically contain code to initialize memory regions, request GPIOs, etc.

sample-device.c - Object Initialization

```
static void sample_device_realize(DeviceState *dev, Error **errp) {
}
```

Lastly, the reset method, setup during class initialization, is called whenever the device is reset (machine reset or manually via another model, etc.). This device simply resets the internal state count variable to zero.

sample-device.c - Device Reset

```
static void sample_device_reset(DeviceState *dev) {
    SampleDevice *sample = SAMPLE_DEVICE(dev);
    sample->count = 0;
}
```

At this point the device can be dynamically added to the default i386 machine from the command line. Note that the device cannot be added using the monitor since the i386 has to have knowledge of the device in order to allow hotplugging.

```
$ ./qemu-system-i386 -device sample alpine.img
```

References:

- Huth, Thomas. Blog Post. QEMU's instance_init() vs. realize(). Sep 10, 2018.

Properties

For unknown reasons, the qdev API includes it's own property system rather than simply use the underlying QOM properties. The qdev properties are typically simpler to use. See the

`include/hw/qdev-core.h` file for details, specifically the `Property` and `PropertyInfo` structures. In addition, the `include/hw/qdev-properties.h` file provides numerous `DEFINE_PROP_*` macros for most property types that simply initialze the `Property` structures. The various `DEFINE_PROP_*` macros allow the user to specify a string name for the property, the device state structure name, the specific field in the device state structure, and typically a default value for the property to set on object creation. QEMU then handles getting/setting the property, eliminating boilerplate code in the device model, although for more advanced property types the user may have to define getter/setter functions.

NOTE	An `id` property can be set for all devices, whether or not it is defined the device. This special `id` property is used by QEMU for certain types of front and back end connections.

The following incomplete list of property types highlight some of the most useful and primitive properties for ITC development:

- Unsigned integers (uint8_t, uint16_t, uint32_t, uint64_t)
- Signed integer (int64_t)
- Bool
- Size (uint64_t)
- Arrays
- String (char*)
- Pointer (**NOTE:** recommended to avoid pointer properties and instead use links)
- Link (Type safe reference to QOM object or interface)
- OnOffAuto (enum with values on, off, and auto)
- Memory region

The following simple example adds properties to the sample device.

sample-device.c - Properties

```
#include "hw/qdev-properties.h" ①

static Property sample_device_properties[] = { ②
    DEFINE_PROP_UINT32("count", SampleDevice, count, 0),
    DEFINE_PROP_END_OF_LIST(),
};

static void sample_device_class_init(ObjectClass *klass, void *data) {
    DeviceClass *dc = DEVICE_CLASS(klass);
    dc->desc = "ITC Sample Device";
    dc->realize = sample_device_realize;
    dc->reset = sample_device_reset;
    dc->props = sample_device_properties; ③
}
```

① qdev properties header

② Property array defined using qdev convenience macros

③ Setting properties in class init

As previously mentioned, the `DEFINE_PROP_UINT32` macro defines a property named `count` which maps to the `count` member of the `SampleDevice` state structure with a default value of 0.

After adding the above code and recompiling the sample device, the property can be set from the command line when creating the device. The device can then be inspected via the QEMU monitor. With the `id` property set, the device is stored under the machine peripherals container by default.

```
$ ./qemu-system-i386 -device sample,id=foo alpine.img
```

```
(qemu) info qom-tree /machine/peripheral
```

Note that although the property macros are convenient, there are limitations. Namely, when using these macros the properties cannot be changed after the device is realized.

```
(qemu) qom-set /machine/peripheral/foo count 10
Error: Attempt to set property 'count' on device 'foo' (type 'sample') after it was
realized
```

For unknown reasons, this is currently a limitation using qdev. In order to allow dynamic properties, the QOM primitive functions must be called directly. The following code highlights simple usage of the QOM property API:

sample-device.c - Dynamic Properties

```
#include "qapi/visitor.h" ①
#include "qapi/error.h"

static void sample_device_get_count(Object *obj, Visitor *v, const char *name, void
*opaque, Error **errp) ②
{
    SampleDevice *sample = SAMPLE_DEVICE(obj);
    visit_type_uint32(v, name, &sample->count, errp);
}

static void sample_device_set_count(Object *obj, Visitor *v, const char *name, void
*opaque, Error **errp) ③
{
    SampleDevice *sample = SAMPLE_DEVICE(obj);
    uint32_t count;
    Error *local_err = NULL;
    visit_type_uint32(v, name, &count, &local_err);
    if (local_err)
    {
        error_propagate(errp, local_err);
    }
    sample->count = count;
    printf("sample count: %d\n", sample->count);
}

static void sample_device_class_init(ObjectClass *klass, void *data) {
    DeviceClass *dc = DEVICE_CLASS(klass);
    dc->desc = "ITC Sample Device";
    dc->realize = sample_device_realize;
    dc->reset = sample_device_reset;
    dc->user_creatable = true;
    object_class_property_add(klass, "count", "uint32", sample_device_get_count,
sample_device_set_count, NULL, NULL, NULL); ④
}
```

① qapi visitor and error headers

② Property getter function

③ Property setter function

④ Adding object class property in device instance init

Using QOM, the property can now be set either via the command line when creating the device or via the QEMU monitor. Because QOM is a completely separate property system than qdev, note however, that the property will **NOT** show up in various monitor commands such as `info qtree`. The ITC team hopes these two parallel paths are combined in future releases.

```
$ ./qemu-system-i386 -device sample,id=foo,count=10 alpine.img
```

```
(qemu) qom-set /machine/peripheral/foo count 10
sample count: 10
```

Note that the full QOM path does not have to be used if the object name is unique. For example, the following can also be used to set the device count property:

```
(qemu) qom-set peripheral/foo count 10
sample count: 10
(qemu) qom-set foo count 12
sample count: 12
```

TODO: add more detailed info, including in-depth link info

Memory Mapped IO (MMIO)

For a detailed overview of the QEMU memory model, see the `docs/devel/memory.rst` file in the QEMU source tree. The main memory related objects in QEMU are `MemoryRegion` and `AddressSpace` objects. In a nutshell, address spaces define memory as seen from the CPU or a device perspective. The `include/exec/address-spaces.h` file contains the global memory and IO address spaces, along with the `get_system_memory` function to get the top-level `MemoryRegion` object. The QEMU memory API models all memory region types using a single `MemoryRegion` type. QEMU memory is modeled as an acyclic graph of `MemoryRegion` objects. Memory regions can be nested for convenience, each memory region is mapped to a specific offset into the parent region. The `include/exec/memory.h` header file includes the memory region API with detailed function comments. The following lists some of the common memory region types:

- RAM (`memory_region_init_ram`)
- MMIO (`memory_region_init_io`)
- ROM (`memory_region_init_rom`)
- IOMMU (`memory_region_init_iommu`)

Memory regions can be mapped into an address space or other memory region using `memory_region_add_subregion`. The memory region is attached to an object and destruction is automatic upon owner object destruction.

Memory regions can be overlapped and assigned a priority to fill in **holes** using `memory_region_add_subregion_overlap`. Priority values are signed and the highest priority wins. All regions default to priority zero.

NOTE	Memory region priorities are useful for *shadow* memory or for filling in don't care regions of unmapped memory.

As will be illustrated further in the device model section, memory mapped IO (MMIO) regions allow read/write callbacks to be handled in user device code. Each device can also apply constraints, such as min/max access size, alignment, endianness, etc. All of the operation parameters are defined in a

`MemoryRegionOps` structure.

Continuing with the sample device, the following code snippets can be added to enable a memory region. It should also be reiterated that the machine/board code is typically responsible for mapping the memory to the overall system and everything at the device level is relative. Also, note that the `TYPE_DEVICE` had been changed to `TYPE_SYS_BUS_DEVICE`. As previously mentioned, this document lumps the sysbus device API with the qdev API, as sysbus devices are built on top of the basic qdev `TYPE_DEVICE`, adding additional functionality such as mapping device memory to system memory, connecting device GPIO/IRQ to system level IRQs, etc.

sample-device.c - Memory Mapped IO (MMIO)

```
typedef struct {
    SysBusDevice parent;
    MemoryRegion mmio; ①
    uint32_t count;
} SampleDevice;

static uint64_t sample_device_read(void *opaque, hwaddr addr, unsigned size) ②
{
    printf("**READ**\n");
    return 0;
}

static void sample_device_write(void *opaque, hwaddr addr, uint64_t value, unsigned
size) ③
{
    printf("**WRITE**\n");
}

static const MemoryRegionOps sample_device_ops = { ④
    .read = sample_device_read,
    .write = sample_device_write,
    .endianness = DEVICE_LITTLE_ENDIAN,
    .valid = {
        .min_access_size = 4,
        .max_access_size = 4,
    },
    .impl = {
        .min_access_size = 4,
        .max_access_size = 4,
    },
};

static void sample_device_realize(DeviceState *dev, Error **errp) {
    SampleDevice *sample = SAMPLE_DEVICE(dev);

    SysBusDevice *sbd = SYS_BUS_DEVICE(dev); ⑤
    memory_region_init_io(&sample->mmio, OBJECT(dev), &sample_device_ops, sample,
"sample", 0x8);
    sysbus_init_mmio(sbd, &sample->mmio);
    sysbus_mmio_map(sbd, 0, 0x07000000);
}

static const TypeInfo sample_device_info = {
    .name = TYPE_SAMPLE_DEVICE,
    .parent = TYPE_SYS_BUS_DEVICE, ⑥
    .instance_size = sizeof(SampleDevice),
    .class_init = sample_device_class_init,
};
```

① Memory region added to device state

② Memory region read operation callback

③ Memory region write operation callback

④ Memory region operations structure

⑤ Memory region initialization

⑥ Parent type changed from `TYPE_DEVICE` to `TYPE_SYS_BUS_DEVICE`

Registers

Unfortunately, QEMU does not contain a single unified method of modeling registers. QEMU devices use the following three methods of modeling registers, ordered from the most widely used:

1. Manual / Case Statement

   The most common pattern used throughout QEMU code is to use `#define` for all register related constants (addresses, masks, bit shifts, etc.). The register assigned memory region read/write callbacks then contain case statements for each address and manually handle the call. This is tedious and error prone, especially handling the various bit patterns, such as write-1-clears, read-only, etc., in addition to properly handling reset values. The code can quickly grow complicated and hard to maintain.

2. Register Field Macros

   The second most widely used method of modeling registers appears to be using the convenience macros in `include/hw/registerfields.h`. This simplifies case 1 by providing macros that allow user code to simply specify register names, fields, lengths, etc. and it generates the appropriate enums containing the info that was manually created in case 1. In addition, it has helper macros that allow field deposit and extraction. Although this simplifies case 1, it still doesn't solve the issue of dealing with the various field bit patterns, such as read-only, write-1-clears, etc.

3. Register Object

   Xilinx contributed a register management object (`include/hw/register.h`) to QEMU which allows fields to be named and updated in some common patterns. This management object removes **some** of that boilerplate by building in configuration to handle clearing bits, read-only bits, and reads/writes to fields within registers. The header file contains the following constructs:

   - `RegisterInfo` This struct wraps the data itself for any given register, with a underlying size of 1, 2, 4, or 8 bytes. Also included is a `RegisterAccessInfo` pointer that defines the register characteristics.
   - `RegisterAccessInfo` This struct defines the characteristics of a register, and a set of function handles to intercept reads/writes for modification. Besides the name, the struct has bitfields that denote whether the register's bits are read-only, write-1-to-clear, clear-on-read, reset values, and reserved/no access.
   - `RegisterInfoArray` This struct groups all the registers associated with a device (represented by `RegisterInfo` objects) along with a name and a memory region. As a part of the parent

object's realization, this array is populated by a call to `register_init_block32()`, which binds it to a device, assigns pre-allocated memory for register data, and initializes the underlying `MemoryRegion` for use in a container upstream.

In a nutshell, this allows you to register common register traits and a handler to intercept data. You still need to manually apply those traits, but the framework will organize it for you. See `hw/rtc/xlnx-zynqmp-rtc.c` and `hw/dma/xlnx-zdma.c` for examples of how to use the library.

General Purpose IO (GPIO)

TODO

PCI Devices

TODO This section under construction!!

The QEMU source includes an *educational* PCI device that does a great job demonstrating PCI modeling in QEMU, including memory mapping and DMA functionality. For reference, see `docs/specs/edu.txt` and `hw/misc/edu.c`. In summary, QEMU includes a `TYPE_PCI_DEVICE` that can be set as the parent device. The QEMU code also includes `PCIDeviceClass` and `PCIDevice` structures to simplify setting up the PCI configuration space, along with definitions of all standard PCI registers. See `include/hw/pci/pci.h` for more information.

TODO: pci bridges, host bridges, etc.

NOTE	The QEMU host bridge assumes a PC/i386 compatible PCI config space access mechanism, using the standard `CONFIG_ADDRESS` and `CONFIG_DATA` registers. If a different access mechanism is used the user has to write custom PCI config access.

References:

- Polard, Tic Le. Emulate a PCI device with Qemu. Jan 10, 2015.

I^2C Devices

TODO

Machines

TODO

CPU

Tiny Code Generator (TCG)

Topics

This chapter descripes various miscellaneous topics related to QEMU of interest to the ITC team.

Custom Commands

This section describes how to implement custom commands in both HMP and QMP. Similar to the ITC developed Simics based simulators, these commands can be added to support GUIs, general debugging, fault scenarios, etc.

Custom QMP Commands

QMP commands are defined in JSON files in the `qapi` directory and implemented in source as the same name as the command with hyphens replaced with underscores and prefixed with `qmp_`. For example, this guide will illustrate a custom QMP command to get the current sim time. The command is `itc-sim-time` and the command function is `qmp_itc_sim_time`.

The first step to defining a custom QMP command is to define the command in a JSON file. The following example file illustrates a few concepts, such as defining the command, defining custom enumeration types, and returning a custom structure.

NOTE	For some reason QMP appears to limit the return type of commands and POD types, such as int, cannot be used and instead must be *wrapped* in a custom struct.

itc.json - QMP Command Definition

```
##
# @ClockType:
#
# QEMU clock type enumeration (see include/qemu/timer.h).
##
{ 'enum': 'ClockType',
  'data': [ 'realtime', 'virtual', 'host', 'virtual-rt' ] }

##
# @SimTime:
#
# @time_ns: Sim time (ns).
#
##
{ 'struct': 'SimTime', 'data': { 'time_ns': 'int64' } }

##
# @itc-sim-time:
#
# Get current simulation time.
#
##
{ 'command': 'itc-sim-time',
  'data': { 'clock': 'ClockType' },
  'returns': 'SimTime' }
```

The custom JSON file must then be included in the schema by appending the following line to the end of the schema file (`qapi/qapi-schema.json`):

qapi-qapi-schema.json

```
{ 'include': 'itc.json' }
```

Finally, the command must be implemented in a custom source file and added to the build appropriately. The following file illustrates the custom command to get the current simulation time.

commands.c - QMP Command Implementation

```
#include "qemu/osdep.h"
#include "qemu/timer.h"
#include "qapi/error.h"
#include "qapi/qapi-commands-itc.h"

SimTime* qmp_itc_sim_time(ClockType clock_type, Error **errp)
{
    assert(errp && !*errp);

    SimTime *sim_time = NULL;
    QEMUClockType clock = QEMU_CLOCK_MAX;

    switch(clock_type)
    {
    case CLOCK_TYPE_REALTIME:
        clock = QEMU_CLOCK_REALTIME;
        break;
    case CLOCK_TYPE_VIRTUAL:
        clock = QEMU_CLOCK_VIRTUAL;
        break;
    case CLOCK_TYPE_HOST:
        clock = QEMU_CLOCK_HOST;
        break;
    case CLOCK_TYPE_VIRTUAL_RT:
        clock = QEMU_CLOCK_VIRTUAL_RT;
        break;
    default:
        error_setg(errp, "unknown qemu clock type: %s\n", ClockType_str(clock_type));
    }

    if(clock < QEMU_CLOCK_MAX)
    {
        sim_time = g_new0(SimTime, 1);
        sim_time->time_ns = qemu_clock_get_ns(clock);
    }

    return sim_time;
}
```

Debugging

This chapter describes various methods of debugging QEMU, including the device models, along with the running guest OS, or flight software (FSW) in the case of ITC simulations.

Logging/Tracing

GDB