



QEMU: Emulate Everything

From Custom CPU Instructions to Peripherals

Jan Kaisrlik
Linuxdays 2026

About Me

Jan Kaisrlik

- Engineer @Intel (Research & POC) & Contractor
- System and Embedded Linux specialist/enthusiast
- Currently running custom Linux on Synology RS815RP+

What is QEMU?

Quick EMUlator - open-source emulator and virtualizer

- Emulates complete systems (CPU, memory, peripherals)
- *Virtualize same hardware architecture*
- Supports many architectures (ARM, x86, RISC-V, ...)

```
qemu-system-x86_64 -name nixos -machine q35,accel=kvm \  
-cpu host -smp 2 -m 4G,slots=0,maxmem=6G \  
--bios OVMF.fd \  
-drive file=nixos.qcow2,id=drive1,if=none,index=1 \  
-device virtio-blk-pci,bootindex=1,drive=drive1 \  
-nographic \  
-net nic,model=virtio -net user,hostfwd=tcp::2222-:22
```

```
BdsDxe: loading Boot0002 "UEFI Misc Device" from PciRoot(0x0)/Pci(0x3,0x0)
BdsDxe: starting Boot0002 "UEFI Misc Device" from PciRoot(0x0)/Pci(0x3,0x0)
EFI stub: Loaded initrd from LINUX_EFI_INITRD_MEDIA_GUID device path
[ 0.000000] Linux version 6.12.31 (nixbld@localhost) (gcc (GCC) 14.2.1 20250322, GNU ld (GNU Binutils) 2.44) #1-NixOS SMP PREEMPT_DYNAMIC Thu May 29 09:03:27 UTC 2025
[ 0.000000] Command line: BOOT_IMAGE=(hd0,gpt1)//kernels/80rcjqc4ch0f7hc5d3rdips69vx3fswv-linux-6.12.31-bzImage init=/nix/store/zäcjp6309amlx4brk71h9jy2nwq7901a-nixos-system-nixos-25.11.20
250603.7ab4906/init console=ttyS0 console=ttyS0 memory_hotplug.memmap_on_memory=force iomem=relaxed selinux=0 audit=0 efi=nosoftreserve loglevel=7 lsm=landlock,yama,bpf
[ 0.000000] BIOS-provided physical RAM map:
[ 0.000000] BIOS-e820: [mem 0x0000000000000000-0x0000000000009ffff] usable
[ 0.000000] BIOS-e820: [mem 0x0000000000100000-0x000000000007ffff] usable
[ 0.000000] BIOS-e820: [mem 0x0000000000800000-0x0000000000807fff] ACPI NVS
[ 0.000000] BIOS-e820: [mem 0x0000000000808000-0x000000000080afff] usable
[ 0.000000] BIOS-e820: [mem 0x000000000080b000-0x000000000080bfff] ACPI NVS
[ 0.000000] BIOS-e820: [mem 0x000000000080c000-0x0000000000810fff] usable
[ 0.000000] BIOS-e820: [mem 0x0000000000811000-0x000000000081ffff] ACPI NVS
[ 0.000000] BIOS-e820: [mem 0x0000000000900000-0x000000000090ffff] usable
[ 0.000000] BIOS-e820: [mem 0x0000000000f8ed000-0x0000000000fb6cfff] reserved
[ 0.000000] BIOS-e820: [mem 0x0000000000fb6d000-0x0000000000fb7efff] ACPI data
[ 0.000000] BIOS-e820: [mem 0x0000000000fb7f000-0x0000000000fbfefff] ACPI NVS
[ 0.000000] BIOS-e820: [mem 0x0000000000fbff000-0x0000000000feaefff] usable
[ 0.000000] BIOS-e820: [mem 0x0000000000feaf000-0x0000000000feb2fff] reserved
[ 0.000000] BIOS-e820: [mem 0x0000000000feb3000-0x0000000000feb4fff] ACPI NVS
[ 0.000000] BIOS-e820: [mem 0x0000000000feb5000-0x0000000000feebfff] usable
[ 0.000000] BIOS-e820: [mem 0x0000000000feec000-0x0000000000fff6fff] reserved
[ 0.000000] BIOS-e820: [mem 0x0000000000fff7000-0x0000000000ffffff] ACPI NVS
[ 0.000000] BIOS-e820: [mem 0x00000000e0000000-0x00000000e0ffffff] reserved
[ 0.000000] BIOS-e820: [mem 0x00000000effc000-0x00000000efffffff] reserved
[ 0.000000] BIOS-e820: [mem 0x0000000100000000-0x000000017ffffff] usable
[ 0.000000] NX (Execute Disable) protection: active
[ 0.000000] APIC: Static calls initialized
[ 0.000000] extended physical RAM map:
```

Figure 1: Simple runtime using arguments from previous slide

The Problem: Waiting for Hardware

- HW development takes **months or years**
 - SW teams are blocked
 - SW feedback could be crucial in design phase
- Integration comes late in the cycle
- Expensive prototypes

Before HW production:

- SW development without physical device
- Testing different configurations
- Prototyping peripherals

After/Before HW production:

- CI/CD without physical HW
- Automated testing
- Debugging
- Tracing invalid memory accesses

What We'll Cover Today

A few practical scenarios for QEMU in development:

1. **Controlling QEMU basics** - Simple way to control and obtain information
2. **Creating Peripherals** - Adding new devices to your system
3. **Custom CPU Instructions** - Extending processor capabilities with PicoRV32 example
4. **Integration with External Tools** - Connecting QEMU with other simulators/apps

QEMU Basics: Running and Monitoring

Basic QEMU invocation:

```
qemu-system-x86_64 \  
# Board/platform type, acceleration mode  
-machine q35,accel=kvm \  
# CPU model  
-cpu host \  
# RAM size  
-m 2G \  
# Kernel image  
-kernel bzImage \  
# Network virtio device  
-net nic,model=virtio \  
# Network forwarding of ssh  
-net user,hostfwd=tcp::2222-:22 \  
# No GUI, console only  
-nographic  
# Run qemu on daemonize mode  
-daemonize
```

The QEMU Monitor:

Interactive console to control QEMU:

```
# Enable monitor  
qemu-system-x86_64 ... \  
-monitor stdio
```

Useful monitor commands:

(qemu) info registers	# CPU state
(qemu) info mtree	# Memory map
(qemu) x /16xw 0x1000	# Examine virt addr
(qemu) xp /16xw 0x1000	# Examine phys addr
(qemu) stop	# Pause VM
(qemu) cont	# Resume VM
(qemu) quit	# Exit QEMU

Connecting Existing Peripherals in QEMU

Command-line arguments:

```
# Use machine's I2C bus
qemu-system-arm -machine virt \
    -device ds1338,bus=i2c.0,address=0x68

# Add I2C controller that creates bus
qemu-system-arm -machine virt \
    -device imx-i2c,id=i2c0 \
    -device ds1338,bus=i2c0,address=0x68
```

Device options:

- -device - Add peripheral
- bus= - Specify bus connection

Query system information:

```
# List available devices
qemu-system-x86_64 -device help
# Inspect available buses/devices
qemu-system-x86_64 -machine virt \
    -monitor stdio -S
(qemu) info qtree
(qemu) info qom-tree
```

```
(qemu) info network
hub 0
\ hub0port1: #net122: index=0,type=user,net=10.0.2.0,restrict=off
\ hub0port0: e1000e.0: index=0,type=nic,model=e1000e,macaddr=52:54:00:12:34:56
```

Simple PCIe Peripheral

The Simplest PCIe Peripheral - full example

Basic structure:

```
#define TYPE_SIMPLE_DEV "simple-dev" \
    DECLARE_INSTANCE_CHECKER(SimpleDev, \
        SIMPLE_DEV, TYPE_SIMPLE_DEV)

typedef struct {
    PCIDevice parent_obj;
    MemoryRegion mmio;
    uint32_t status_reg;
} SimpleDev;

static const TypeInfo simple_dev_info = {
    .name = TYPE_SIMPLE_DEV,
    .parent = TYPE_PCI_DEVICE,
    .instance_size = sizeof(SimpleDev),
    .class_init = simple_dev_class_init,
};

type_init(simple_dev_info);
```

```
static void simple_dev_class_init(
    ObjectClass *klass, void *data) {
    PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);
    k->realize = simple_dev_realize;
    k->vendor_id = 0x1234;
    k->device_id = 0x5678;
    k->class_id = PCI_CLASS_OTHERS;
}

static void simple_dev_realize(PCIDevice
    *pci_dev, Error **errp) {
    SimpleDev *d = SIMPLE_DEV(pci_dev);
    memory_region_init_io(&d->mmio, OBJECT(d),
        &mmio_ops, d, "simple-mmio", 4096);
    pci_register_bar(pci_dev, 0,
        PCI_BASE_ADDRESS_SPACE_MEMORY, &d->mmio);
}
```

The Simplest PCIe Peripheral

- Device design is “similar” to Linux kernel drivers

MMIO structure:

In previous slide we registered callback for memory interface

```
// simple_dev_realize()  
memory_region_init_io(&d->mmio, OBJECT(d),  
    &mmio_ops, d, "simple-mmio", 4096);
```

```
static const MemoryRegionOps mmio_ops = {  
    .read = simple_mmio_read,  
    .write = simple_mmio_write,  
    .endianness = DEVICE_NATIVE_ENDIAN,  
};
```

```
static uint64_t simple_mmio_read(  
    void *opaque, hwaddr addr, unsigned size)  
{  
    SimpleDev *d = opaque;  
    return d->status_reg;  
}
```

```
static void simple_mmio_write(void *opaque,  
    hwaddr addr, uint64_t val, unsigned size)  
{  
    SimpleDev *d = opaque;  
    d->status_reg = val;  
}
```

Connecting PCIe Device to System

Dynamic Device

User specifies device and bus at runtime:

```
# Connect to PCIe root port
qemu-system-x86_64 -machine q35 \
    -device simple-dev,bus=pcie.0

# With custom properties
qemu-system-x86_64 -machine q35 \
    -device simple-dev,bus=pcie.0,addr=0x4
```

Static Device

Device created in machine init code:

- Fixed board configuration
- Matches real hardware

```
// In board initialization (hw/xxx/board.c)
static void board_init(MachineState
*machine) {
    PCIBus *pci_bus = ...; // Get PCIe bus
    // Create and realize device
    PCIDevice *dev = pci_new(PCI_DEVFN(4, 0),
                             TYPE_SIMPLE_DEV);
    pci_realize_and_unref(dev, pci_bus, &err);
}
```

Building QEMU

1. Get QEMU source:

```
git clone https://gitlab.com/qemu-project/qemu.git
```

2. Configure build:

```
./configure --target-list=x86_64-softmmu \
  --disable-docs \
  --enable-debug
```

3. Build:

```
make -j$(nproc)
```

Add your device files:

```
hw/misc/simple-dev.c
```

Register in build system:

```
# hw/misc/meson.build
system_ss.add(when: 'CONFIG_SIMPLE_DEV',
  if_true: files('simple-dev.c'))
```

```
# hw/misc/Kconfig
config SIMPLE_DEV
  bool
```

Run your build:

```
./build/qemu-system-x86_64 -device simple-dev,bus=pcie.0 ...
```

Tracing

Tracing

Define trace events:

```
# trace-events in device directory
mydev_read(uint64_t addr, uint64_t val) \
    "READ - addr: %#016" PRIx64 ", val: \
    %#016" PRIx64
```

Use in your code:

```
trace_mydev_read(addr, val);
```

Enable traces:

```
# Enable specific events
qemu-system-x86_64 \
    --trace mydev_read,mydev_write ...

# Output to file (default: stderr)
qemu-system-x86_64 \
    --trace mydev_* \
    -trace file=/tmp/trace.log
```

Custom CPU instructions

Custom CPU Instructions

Scenario: You're developing a custom processor with new instructions

Process:

1. Define new instruction in TCG (Tiny Code Generator)
2. Implement decoding
3. Implement behavior (e.g., setting registers, triggering IRQs)

Warning

Custom instructions require TCG (software emulation).

They **cannot** run with KVM/hardware acceleration, as the physical CPU doesn't know about your custom instructions.

Real Example: PicoRV32 in QEMU

PicoRV32 - Compact RISC-V CPU with small footprint: <https://github.com/YosysHQ/picorv32>

Added non-ISA compliant instructions:

- `retirq` - Return from interrupt (custom handling)
- `waitirq` - Wait for interrupt (pause execution)
- `maskirq` - Mask/unmask interrupts
- `setq / getq` - Access special interrupt Q-registers (q0-q3)
- `timer` - Custom timer control

PicoRV32: Instruction Decoding

maskirq

The “IRQ Mask” register contains a bitmask of masked (disabled) interrupts. This instruction writes a new value to the irq mask register and reads the old value.

```
# target/riscv/insn32.decode - Define instruction format
# %rd          7:5
maskirq  0000011 00000 ..... 110 ..... 0001011 %rs1 %rd
```

QEMU automatically generates:

- Decoder function from instruction format
- `arg_maskirq` structure with `rs1`, `rd` fields
- Requires implementing `trans_maskirq()` function

Translation function signature:

```
static bool trans_maskirq(DisasContext *ctx, arg_maskirq *a);
```

PicoRV32: maskirq implementation

maskirq

The “IRQ Mask” register contains a bitmask of masked (disabled) interrupts. This instruction writes a new value to the irq mask register and reads the old value.

TCG (Tiny Code Generator) API:

- `get_gpr()` - read from RISC-V general purpose register
- `dest_gpr()` - prepare destination register for writing
- `tcg_temp_new()` - allocate temporary variable
- `tcg_gen_mov_tl()` - generate move operation

```
// target/riscv/translate.c: example maskirq
static bool trans_maskirq(DisasContext *ctx,
arg_maskirq *a)
{
    TCGv src = get_gpr(ctx, a->rs1, EXT_NONE);
    TCGv dest = dest_gpr(ctx, a->rd);
    TCGv temp = tcg_temp_new();

    tcg_gen_mov_tl(temp, cpu_irq_mask);
    tcg_gen_mov_tl(cpu_irq_mask, src);
    tcg_gen_mov_tl(dest, temp);
    return true;
}
```

Custom CPU registers

PicoRV32: New registers

```
--- a/target/riscv/cpu.c
+++ b/target/riscv/cpu.c
+const char * const riscv_qpr_regnames[] = {
+    "q0/qt0", "q1/qt1", "q2/qt2", "q3/qt3"
+};
--- a/target/riscv/translate.c
+++ b/target/riscv/translate.c
@@ -40,6 +40,7 @@
     static TCGv_i64 cpu_fpr[32]; /* assume F and D extensions */
+static TCGv cpu_qpr[4];
     static TCGv load_res;

@@ -1466,6 +1506,9 @@ void riscv_translate_init(void)
+    for (i = 0; i < MAX_QPR_NR; i++)
+        cpu_qpr[i] = tcg_global_mem_new(tcg_env,
+            offsetof(CPURISCVState, qpr[i]),
+            riscv_qpr_regnames[i]);
```

PicoRV32: GDB Debugging Support

Custom registers can be exposed to GDB for debugging:

- Define register layout in XML
- Implement getter/setter functions
- Register with GDB stub

Custom registers visible in GDB:

```
<!-- gdbstub/gdb-xml/riscv-32bit-qpr.xml -->
<feature name="org.gnu.gdb.riscv.qpr">
  <reg name="q0" bitsize="32"
type="ieee_single"/>
  <reg name="tim0" bitsize="32"
type="ieee_single"/>
</feature>
```

```
// target/riscv/gdbstub.c
static int riscv_gdb_get_qpr(CPUState *cs,
GByteArray *buf, int n)
{
    RISCVCPU *cpu = RISCV_CPU(cs);
    CPURISCVState *env = &cpu->env;
    if (n < MAX_QPR_NR)
        return gdb_get_reg32(buf, env->qpr[n]);
    return 0;
}

// void
riscv_cpu_register_gdb_regs_for_features(CPUState
*cs)
gdb_register_coprocessor(cs, riscv_gdb_get_qpr,
riscv_gdb_set_qpr,
gdb_find_static_feature("riscv-32bit-
qpr.xml"));
```

PicoRV32: GDB Debugging Support

Start QEMU with GDB server:

```
qemu-system-riscv32 -machine picorv32 -  
kernel fw.bin -s -S  
# -s: GDB on port 1234  
# -S: pause at startup
```

Simple example to connect to QEMU, automatically:

```
# .gdbinit  
target remote localhost:1234  
set confirm off  
tui l a  
tui l r  
b main  
c
```

Result:

```
Register group: general  
zero          0x0      0  
sp            0x1240c0 0x1240c0  
tp            0x0      0x0  
t1            0x0      0  
fp            0x1240d0 0x1240d0  
B+>0x100010 <irq_vec>      j      0x100070 <irq_v  
      0x100012 <irq_vec+2>  nop  
      0x100014 <irq_vec+4>  nop  
      0x100018 <irq_vec+8>  nop  
      0x10001c <irq_vec+12> nop  
remote Thread 1.1 (asm) In: irq_vec  
(gdb) info registers q0  
q0          1.47596245e-39  (raw 0x00101262)
```

QEMU and other app/sim/emu

Multiple Environment Integration

In some cases you want to be able to connect your work with work of your colleagues:

- Hardware team uses proprietary simulator
- Software team uses QEMU
- Both need to interact

Solution: Bridge via standard interfaces

- Serial/UART connections (to be showed)
- Network sockets
- Shared memory
- File-based communication

Serial-Controlled Device

Use case: Control QEMU/QEMU-device from external source to allow connection with different tools/fuzzers etc.

Example interface connected to the system:

- Linking host serial with system serial

Example device:

- Accepts text commands over serial (chardev)
- Parses and executes commands
- Sends responses back
- Generates interrupts based on commands

Does not need to be necessary serial, it could be a file, socket, etc.

Device populates USB bus (f.e. -machine=q35)

Create virtual tty device listening on port udp:8888:

```
socat -dd pty,link=/dev/ttyFAKEACM0,raw,echo=0,b115200 udp-listen:8888,reuseaddr,fork
```

You can pass existing device into the system:

```
qemu-system-x86_64 -chardev serial,path=/dev/ttyFAKEACM0,id=usbser \  
-usb -device usb-serial,chardev=usbser
```

Pass message via virtual devices:

```
ncat -u 127.0.0.1 8888
```

Serial Device: Device Setup

```
static void example_realize(DeviceState *dev, Error **errp) {
    ExampleState *s = EXAMPLE(dev);
    // Setup MMIO region (for register access) and irq line
    memory_region_init_io(&s->mmio, OBJECT(s), &ops, s, "example-csr", EXAMPLE_BASE);
    sysbus_init_mmio(SYS_BUS_DEVICE(dev), &s->mmio);
    sysbus_init_irq(SYS_BUS_DEVICE(dev), &s->irq);

    // Setup serial handlers - char device is set via property
    qemu_chr_fe_set_handlers(&s->chr, can_read, receive, NULL, NULL, s, NULL, true);
}

static Property example_properties[] = {
    DEFINE_PROP_CHR("chardev", ExampleState, chr),
};
```

Serial Device: Command Parser

```
// Simplified versions missing parser
static void zam_misccsr_receive(void *opaque, const uint8_t *buf, int size) {
    ExampleState *s = EXAMPLE(opaque);
    process_command(s, s->input_buffer);
}

static void process_command(ExampleState *s, const char *cmd) {
    if (strcmp(cmd, "irq") == 0) {
        qemu_set_irq(s->irq, 1);
        strcpy(response, "IRQ triggered.\n");
    }
    qemu_chr_fe_write_all(&s->chr, response, strlen(response));
}
```

Serial Device: Integration

1. Connect chardev in board init:

```
// hw/riscv/picorv32.c
DeviceState *echo_dev = qdev_new(TYPE_EXAMPLE);
Chardev *char_in = qemu_chr_find("ch0");
// This device does not support dynamic probing
qdev_prop_set_chr(echo_dev, "chardev", char_in);
sysbus_realize_and_unref(SYS_BUS_DEVICE(echo_dev), &error_fatal);
sysbus_mmio_map(SYS_BUS_DEVICE(echo_dev), 0, EXAMPLE_BASE);
```

2. Start QEMU with serial socket:

```
qemu-system-riscv32 -machine picorv32 -kernel fw.bin \
  -chardev socket,id=ch0,host=localhost,port=2233,server=on,wait=off
```

3. Connect and send commands:

```
$ nc localhost 2233
irq
IRQ triggered.
```

Shared device example

Shared Device Scenario

Challenge: Multiple QEMU instances need to access the same device

Single System (Simple):

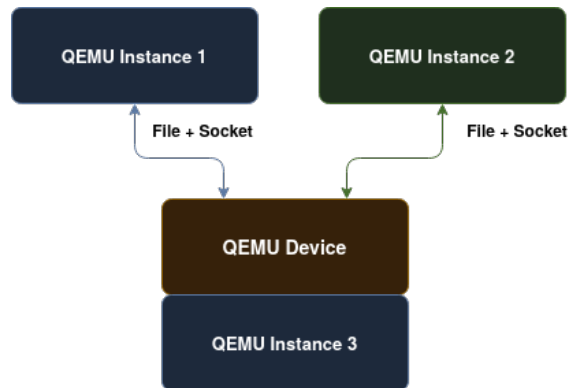
- One QEMU instance
- Direct device access
- Standard device model

Solutions:

- Device proxy over network/socket
- Shared memory regions
- Message passing between instances

Multiple Systems (Complex):

- Multiple QEMU instances
- Shared device access
- Need synchronization



Conclusion

- QEMU **accelerates** embedded systems development
- Enables **parallel** HW and SW development
- **Open-source** and actively developed
- Scalable from simple to complex systems

Stop waiting for hardware - emulate it!

Questions?

Thanks!

Resources

- <https://www.qemu.org/>
- https://github.com/Kaisrlik/hello_simple_qemu_dev_and_driver
- https://github.com/Kaisrlik/qemu/tree/picorv32_support