Lecture 07 – Physical and virtual devices (e1000 NIC and virtio-net)

PCI discovery, BARs, descriptor rings and the packet path on real hardware, then how virtio-net keeps the rings and removes the register traffic that makes an emulated card slow. Running example: an Intel 82540EM (e1000) and a QEMU/KVM virtio-net device.

Contents

01 The big picture

02 Finding devices on PCI

03 BARs: sizing and placement

04 The driver and its registers

05 Rings and the packet path

06 Virtualizing a NIC

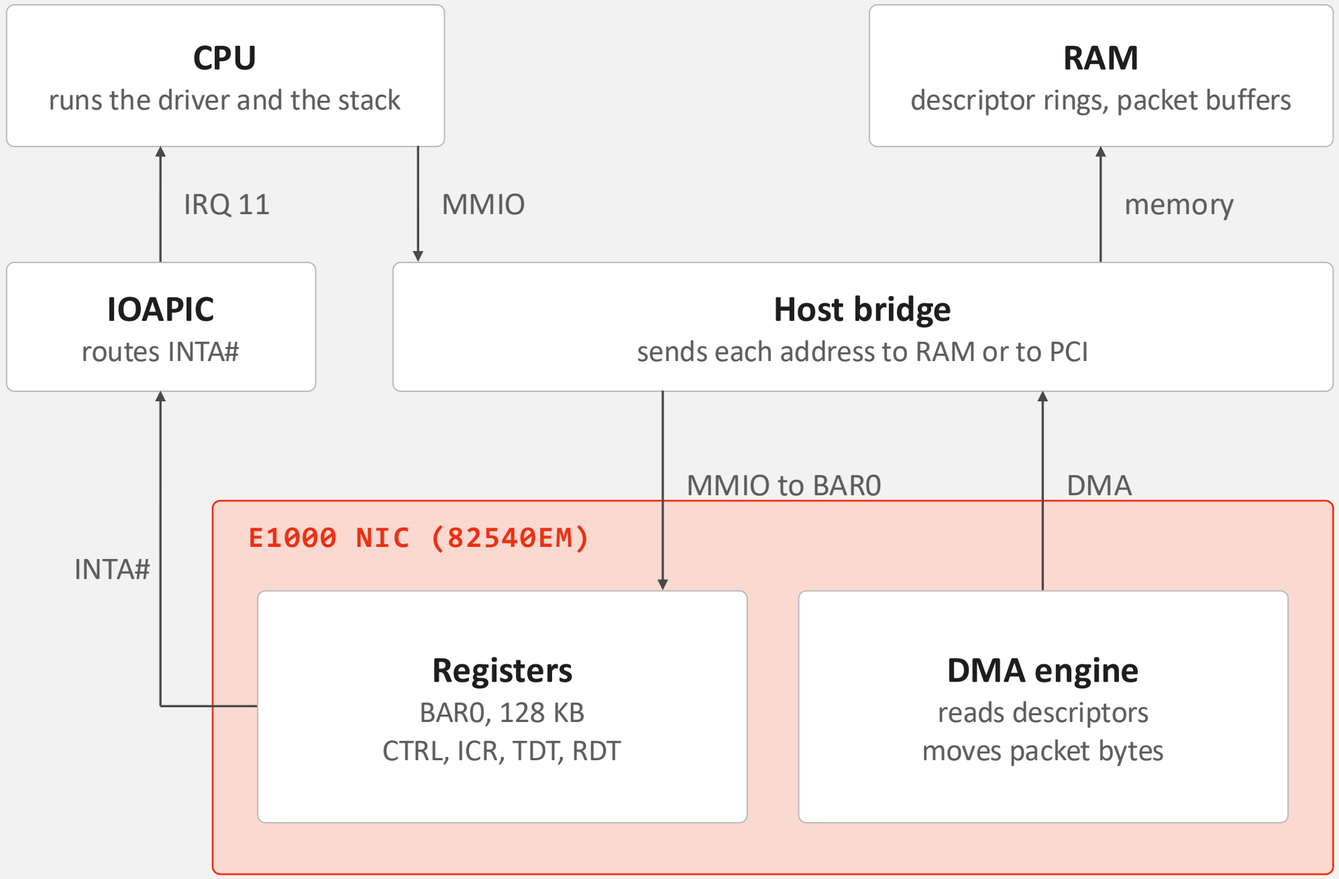
07 virtio-net

08 Takeaways

01

The big picture

Three ways a CPU and a device talk



MMIO · CPU TO DEVICE

The card's registers sit at physical addresses; a load or store there goes over the bus to the card.

DMA · DEVICE TO RAM

The card reads and writes main memory itself. All packet data moves this way.

INTERRUPTS · DEVICE TO CPU

The card raises a signal to say that something happened.

CONFIG SPACE · BOOT ONLY

A standard header per device, used to find it and set up the other three.

02

Finding devices on PCI

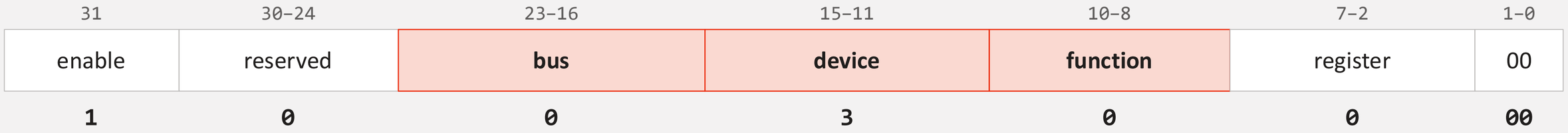
The config space header

Every PCI device starts with a standard header that says who it is and what it needs. Software reads it slot by slot.

OFFSET	FIELD	E1000 VALUE	MEANING
0x00	Vendor ID	0x8086	Intel; 0xFFFF means no device
0x02	Device ID	0x100E	82540EM
0x04	Command	memory and I/O decoding on	which windows the card answers; bus master is set by the driver
0x09–0x0B	Class code	0x020000	network controller, Ethernet
0x0E	Header type	0x00	ordinary device; 0x01 = bridge
0x10–0x24	BAR0–BAR5	BAR0 = 0xFEBC0000, BAR1 = 0xC000	where its register windows live
0x3C	Interrupt Line	11	the IRQ the firmware routed the pin to
0x3D	Interrupt Pin	1	the card uses INTA#

Reaching config space: ports and ECAM

Port 0xCF8 selects bus, device, function and register; port 0xCFC carries the data.



Reading the vendor and device ID of 00:03.0, offset 0x00

LEGACY PCI · PORTS

```
outl(0x80001800, 0xCF8);
inl(0xCFC);   /* -> 0x100E8086 */
```

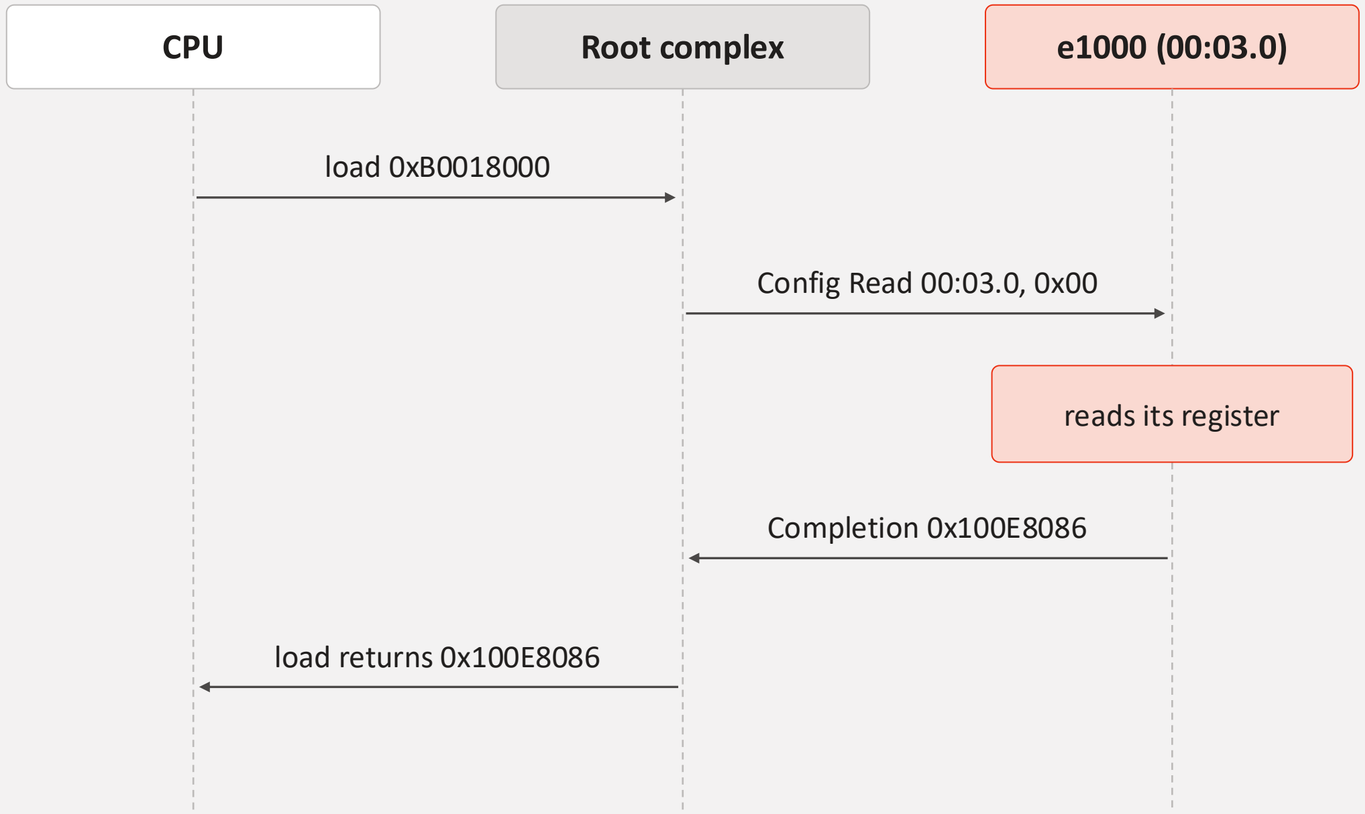
Device ID 0x100E comes back in the upper half, vendor ID 0x8086 in the lower.

PCI EXPRESS · ECAM

```
address = base + (bus << 20)
        + (dev << 15) + (fn << 12) + off
```

Each function gets a 4 KB page; the ACPI MCFG table gives the base. The header contents are the same.

Config space lives on the device, not in RAM



EMPTY SLOTS READ ALL ONES

No device sends a completion, so the root complex returns all ones: vendor ID 0xFFFF means an empty slot.

A RESET WIPES IT

BAR addresses and command bits are lost, so Linux saves config space to RAM before suspend or reset, then restores it.

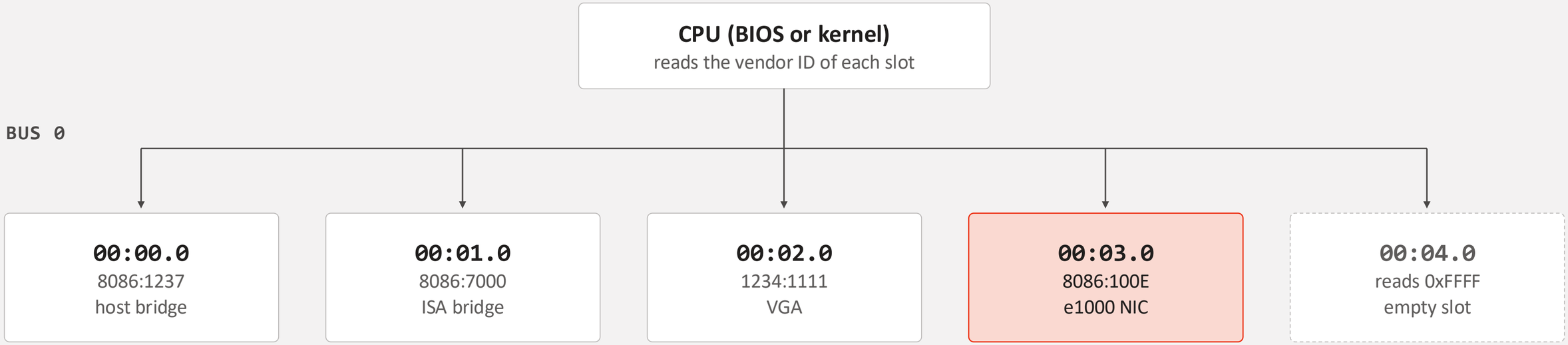
IT IS SLOW

Each read is a round trip the CPU waits for, so the kernel reads it at setup and keeps what it needs in struct pci_dev.

- In a VM there is no device to answer: the access traps, and the emulator replies from a byte array it keeps for each device.

The scan walks every slot, depth-first

For each bus, try devices 0–31 and read the vendor ID. A bridge (header type 1) gets a bus number, and the bus behind it is scanned before the next slot.



- Slots 0–2 are the chipset and graphics; slot 4 reads all ones because nothing is there.

Who decides each part of an address

The ID, 8086:100E, is burned into the chip. The address, bus:device.function, is the device's position in the tree.

00
bus number

03
device number

0
function number

SET BY SOFTWARE

During the scan, the firmware numbers buses as it finds bridges and writes the numbers into them.

SET BY WIRING

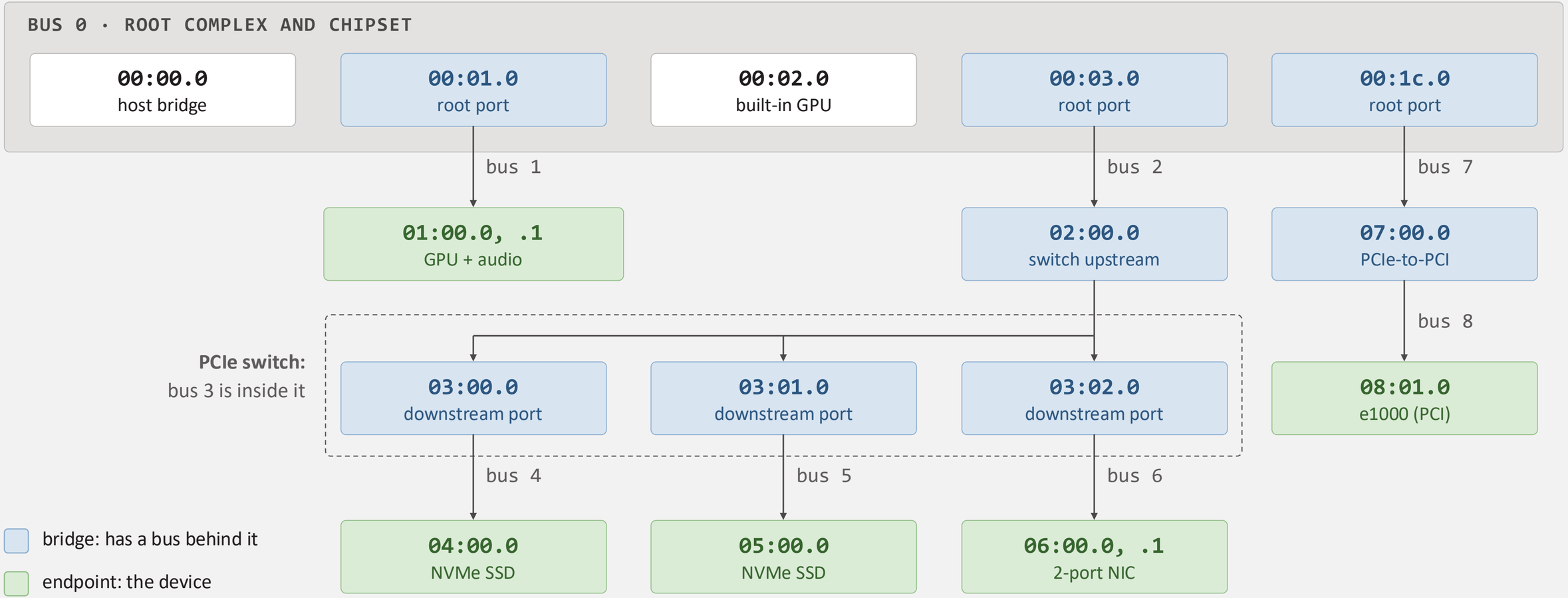
On PCI each slot has its own select line. On PCIe each link is point-to-point, so a card is device 0 on its bus.

SET BY THE DESIGNER

A single NIC is function 0; a dual-port NIC is often functions 0 and 1 of one device.

- On PCIe the bus number is what tells cards apart, because every card is device 0 behind its own link.

A typical PCIe tree



How the scan finds what is on bus 0

THE SCAN, SIMPLIFIED

```
scan_bus(bus):
  for dev in 0 .. 31:
    if read(bus, dev, 0, VENDOR) == 0xFFFF:
      continue                # empty slot
    fns = 8 if hdr_type & 0x80 else 1
    for fn in 0 .. fns-1:
      read class, header type -> pci_dev
  for each bridge found, in slot order:
    bridge.secondary = ++last_bus
    bridge.subordinate = 0xFF
    scan_bus(bridge.secondary)  # depth-first
    bridge.subordinate = last_bus
```

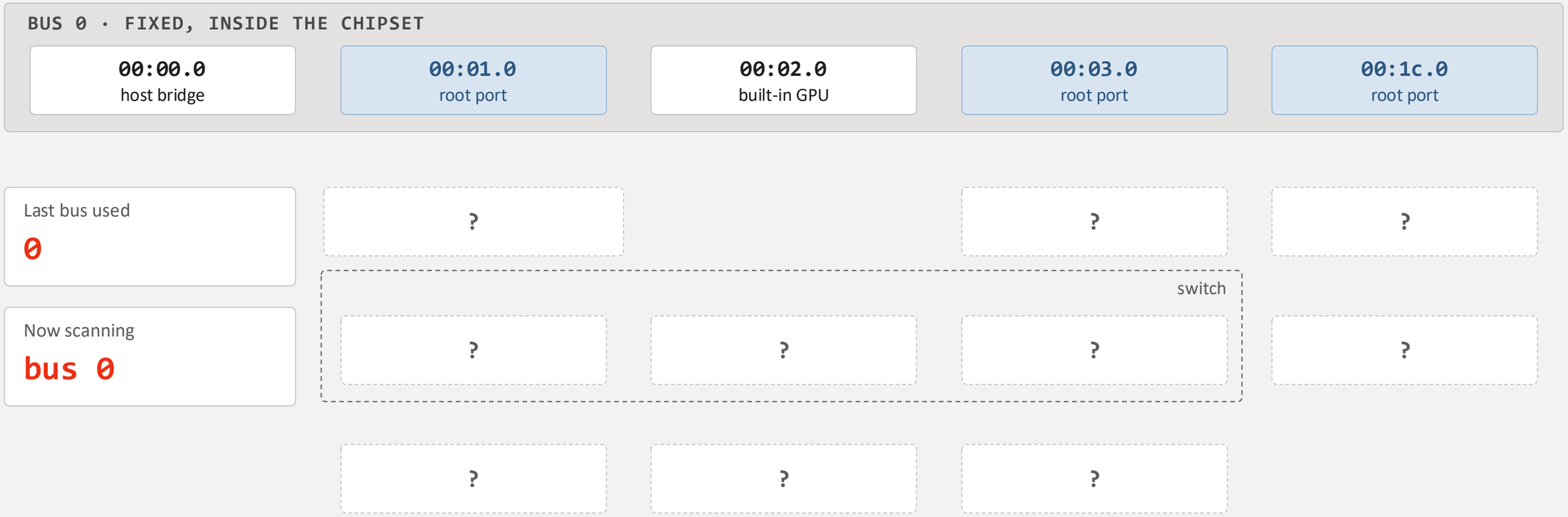
WHAT BUS 0 ANSWERS

SLOT	VENDOR	TYPE	CLASS	WHAT IT IS
00.0	valid	0	0x0600	host bridge: the CPU's side
01.0	valid	1	0x0604	root port: a bus behind it
02.0	valid	0	0x0300	built-in GPU
03.0	valid	1	0x0604	root port
04.0–1b.0	0xFFFF	–	–	empty
1c.0	valid	1	0x0604	root port (chipset)
1d.0–1f.0	0xFFFF	–	–	empty

- Bus 0 is a given: ACPI says the root bus is 0 and where ECAM is. A root port is a PCI-to-PCI bridge (header type 1) whose PCIe capability says root port.

Numbering the buses, depth-first

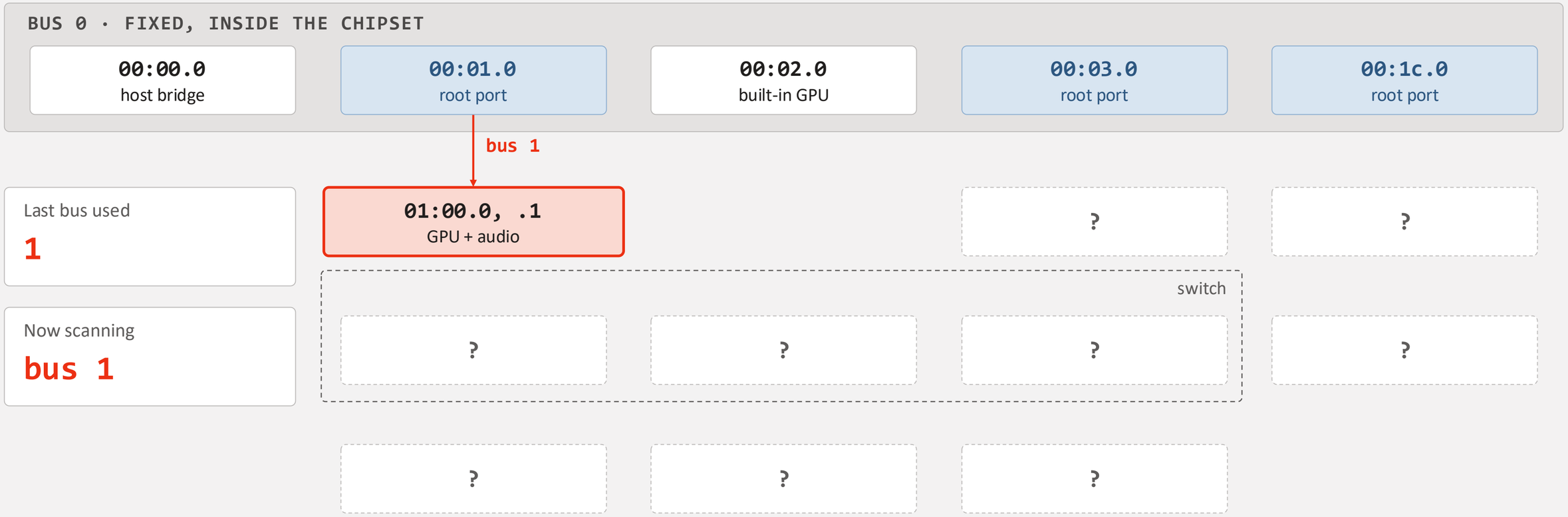
Step 1 of 9



Start: reading slots 0–31 of bus 0 finds five devices at numbers the chipset fixed: 00:00.0, 00:01.0, 00:02.0, 00:03.0, 00:1c.0. Three are bridges. They are now handled one by one, in slot order, each fully before the next.

Numbering the buses, depth-first

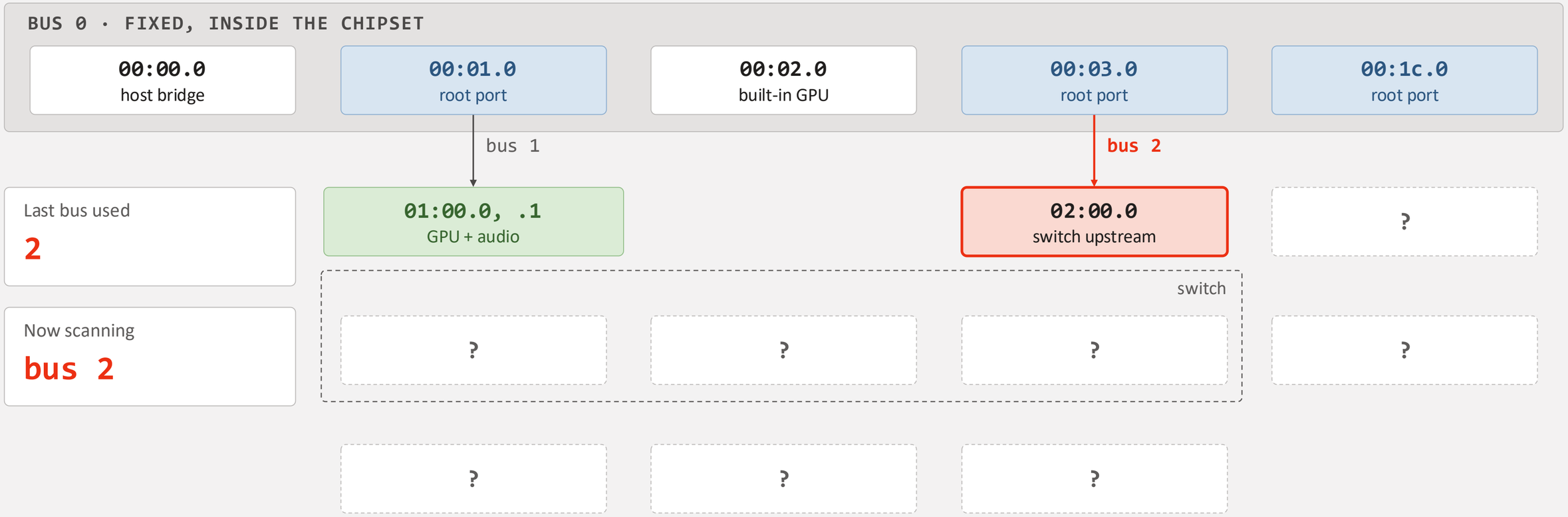
Step 2 of 9



First bridge, 00:01.0, gets the next free bus number, 1, as its secondary bus, and bus 1 is scanned: device 0 is the GPU, 01:00.0, plus its audio function 01:00.1. No bridges below, so the root port's range is 1–1.

Numbering the buses, depth-first

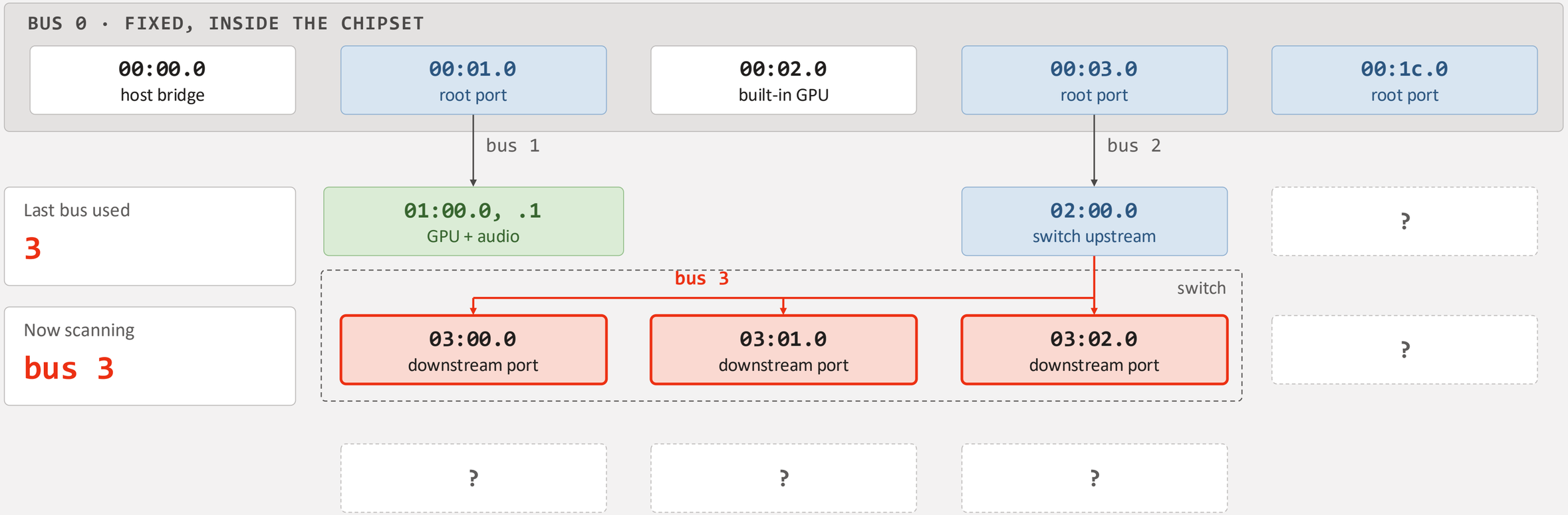
Step 3 of 9



00:02.0 is a GPU, not a bridge. The next bridge, 00:03.0, gets bus 2. Scanning bus 2 finds the switch's upstream port as device 0: 02:00.0. Its 02 is the bus it sits on, not anything copied from 00:03.0.

Numbering the buses, depth-first

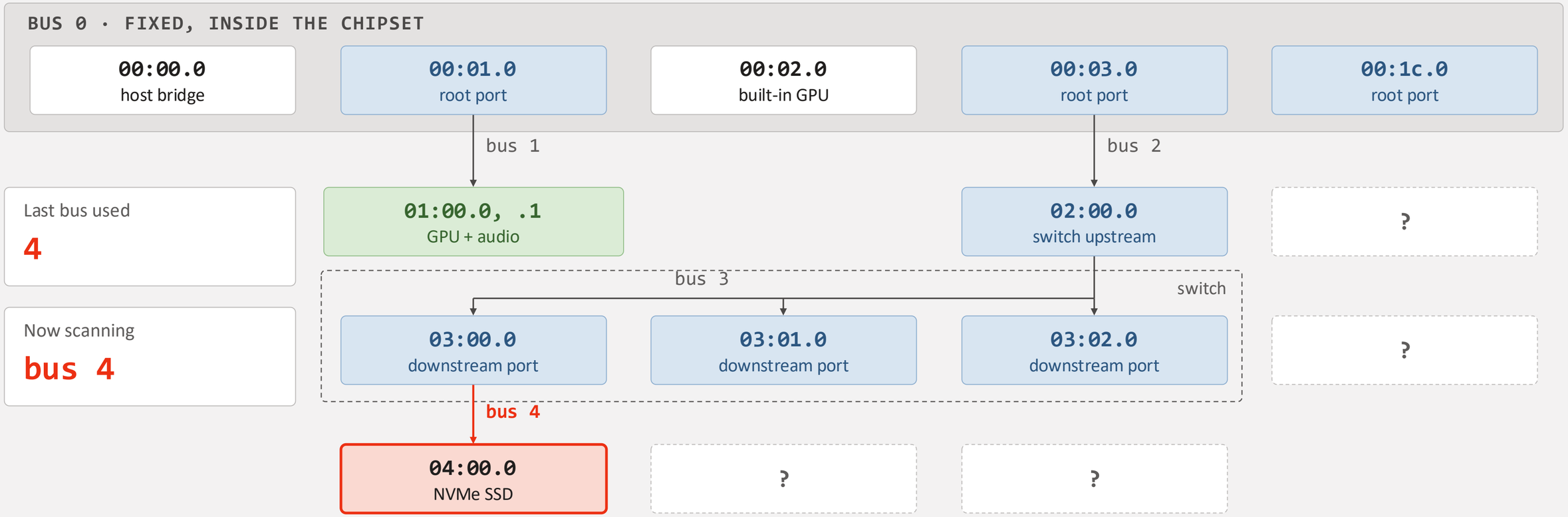
Step 4 of 9



02:00.0 is also a bridge, so it gets bus 3: the bus inside the switch. On bus 3 the downstream ports answer as devices 0, 1 and 2, numbers fixed by the switch's design: 03:00.0, 03:01.0, 03:02.0.

Numbering the buses, depth-first

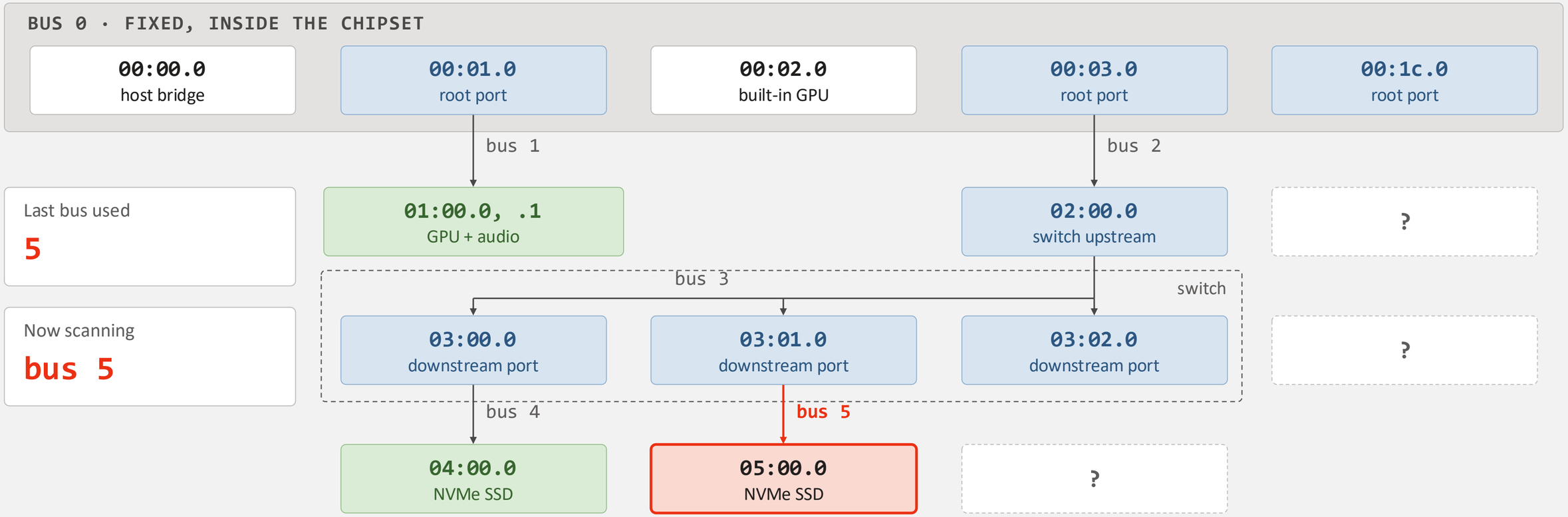
Step 5 of 9



Depth-first: before looking at 03:01.0, the scan goes under 03:00.0. That bridge gets bus 4, and the NVMe drive behind it is 04:00.0. Nothing below it, so its range is 4–4.

Numbering the buses, depth-first

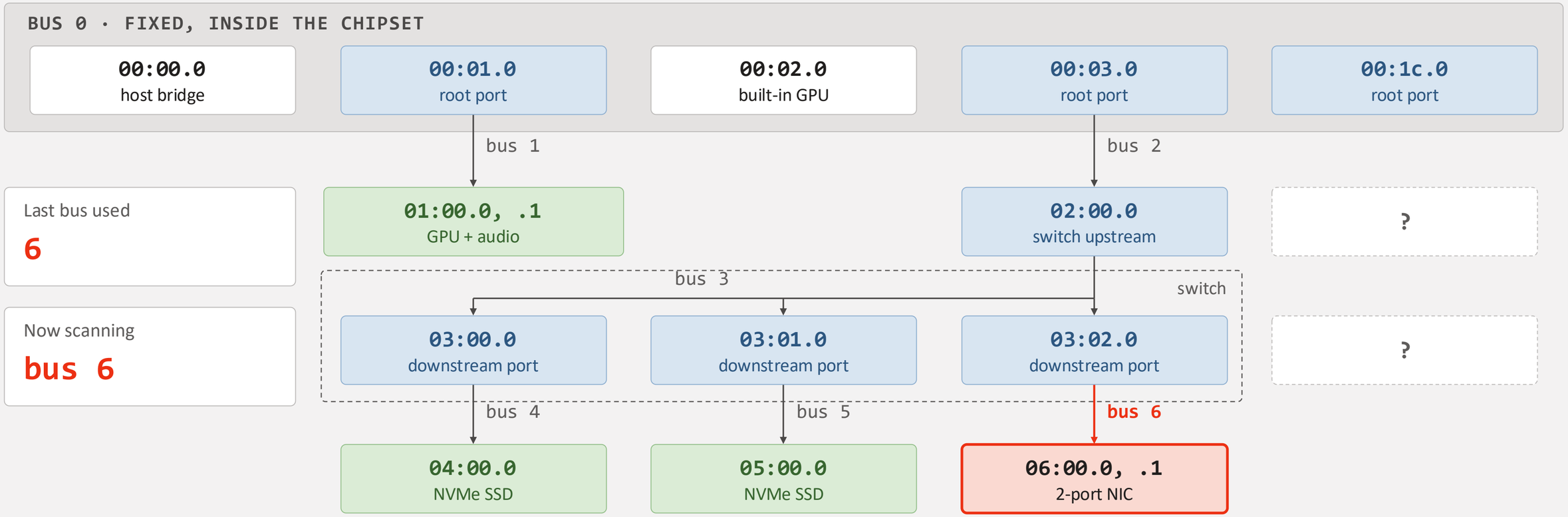
Step 6 of 9



Next downstream port, 03:01.0: it gets bus 5, and its NVMe drive is 05:00.0.

Numbering the buses, depth-first

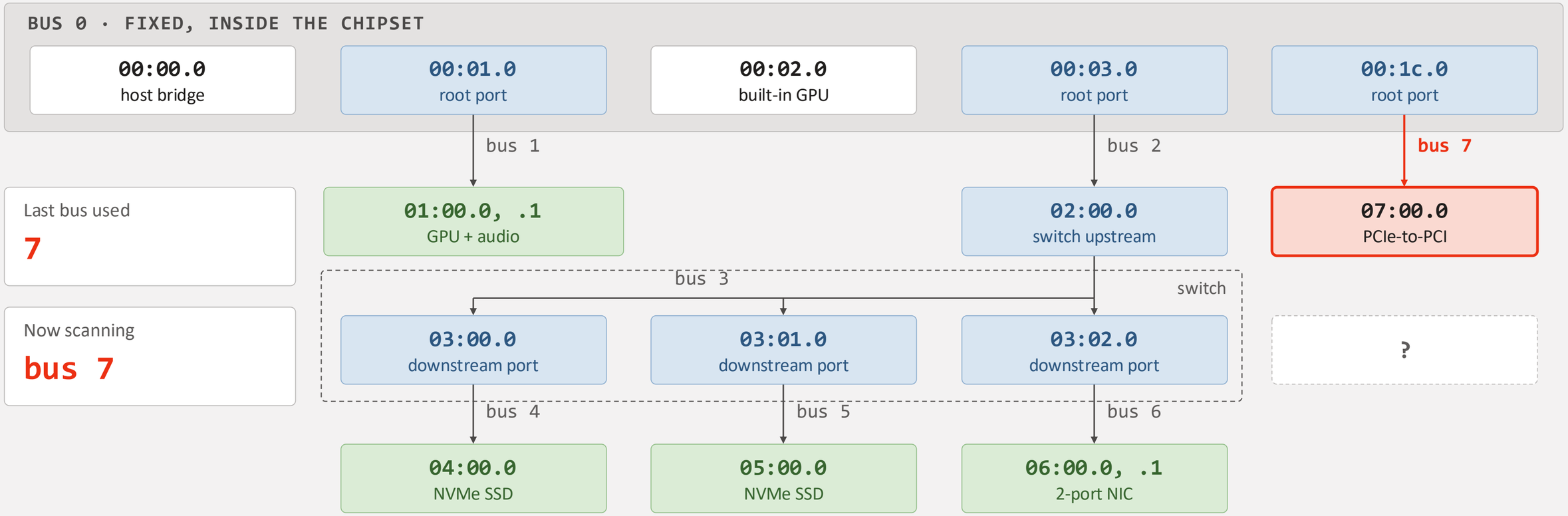
Step 7 of 9



03:02.0 gets bus 6; the NIC behind it is 06:00.0 and 06:00.1. The switch is finished: 02:00.0 records range 3–6, and root port 00:03.0 records 2–6.

Numbering the buses, depth-first

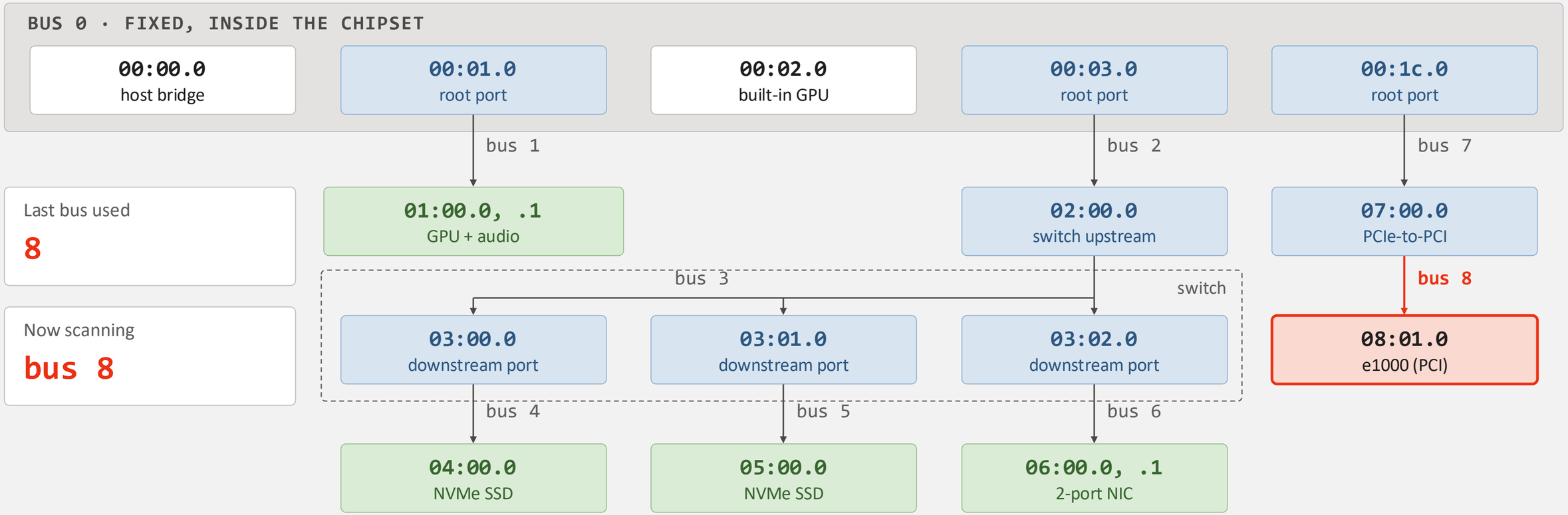
Step 8 of 9



Back on bus 0, the last bridge 00:1c.0 gets the next free number, 7. Device 0 on bus 7 is a PCIe-to-PCI bridge: 07:00.0.

Numbering the buses, depth-first

Step 9 of 9



07:00.0 gets bus 8, a conventional PCI bus where slot wiring sets device numbers: the e1000 is in slot 1, so 08:01.0. The scan is done; buses 0–8 are in use.

Every bus has room for 32 devices

No register says how many devices a bus has. The address format fixes the maximum, and the scan tries every address it allows.

Bus	Device	Function
0–255	**0–31**	**0–7**
8 bits	5 bits	3 bits

ECAM: 4 KB per function × 8 functions × 32 devices = 1 MB per bus.

- Nobody answering reads back as all ones; a device still starting up answers retry, and Linux waits.

WHICH BUSES EXIST

ACPI gives the host bridge's bus range, usually 0–255, and MCFG the ECAM window that covers it.

ONE DEVICE PER PCIE LINK

Behind a root port or downstream port only device 0 can exist, so Linux probes only device 0 there.

FUNCTIONS ONLY WHEN ADVERTISED

Functions 1–7 are tried only if the multi-function bit is set; ARI devices name their next function.

HOT-PLUG SLOTS REPORT PRESENCE

Presence-detect and link-active bits say when a card is there to scan.

What the scan probes on each bus

BUS	WHAT IS ABOVE IT	SLOTS PROBED	FOUND
`0`	root complex	all 32	00.0, 01.0, 02.0, 03.0, 1c.0
`1`	root port 00:01.0	device 0 only	GPU, functions 0 and 1
`2`	root port 00:03.0	device 0 only	switch upstream port
`3`	switch upstream port	all 32	downstream ports 00.0, 01.0, 02.0
`4, 5, 6`	each downstream port	device 0 only	NVMe, NVMe, NIC
`7`	root port 00:1c.0	device 0 only	PCIe-to-PCI bridge
`8`	PCIe-to-PCI bridge	all 32	e1000 at 01.0

PASS 1 · PROBE THE SLOTS

Read function 0's vendor ID in each slot to probe; keep everything that answers.

PASS 2 · FOLLOW THE BRIDGES

In slot order: give each bridge the next bus number and scan behind it fully before the next.

The same tree in lspci -tv

```
$ lspci -tv
-[0000:00]-+-00.0  Host bridge
           +-01.0-[01]--+-00.0  GPU
           |            \-00.1  GPU audio
           +-02.0  Integrated graphics
           +-03.0-[02-06]----00.0-[03-06]--+-00.0-[04]----00.0  NVMe SSD
           |                               +-01.0-[05]----00.0  NVMe SSD
           |                               \-02.0-[06]--+-00.0  NIC port 0
           |                                            \-00.1  NIC port 1
           \-1c.0-[07-08]----00.0-[08]----01.0  e1000
```

- Add -n for numeric vendor:device IDs, or -nn for names and IDs together.

[0000:00]

PCI domain 0000, bus 00: the root of the tree.

-03.0-

device 03, function 0 on the bus above it.

[02-06]

this bridge's bus range: everything on buses 2 to 6 lies behind it.

[08]

a bridge with a single bus behind it; 01.0 there is the e1000.

Giving a bridge a bus number

OFFSET	REGISTER	MEANING
0x18	Primary	the bus the bridge sits on
0x19	Secondary	the bus directly behind it
0x1A	Subordinate	the highest bus below it

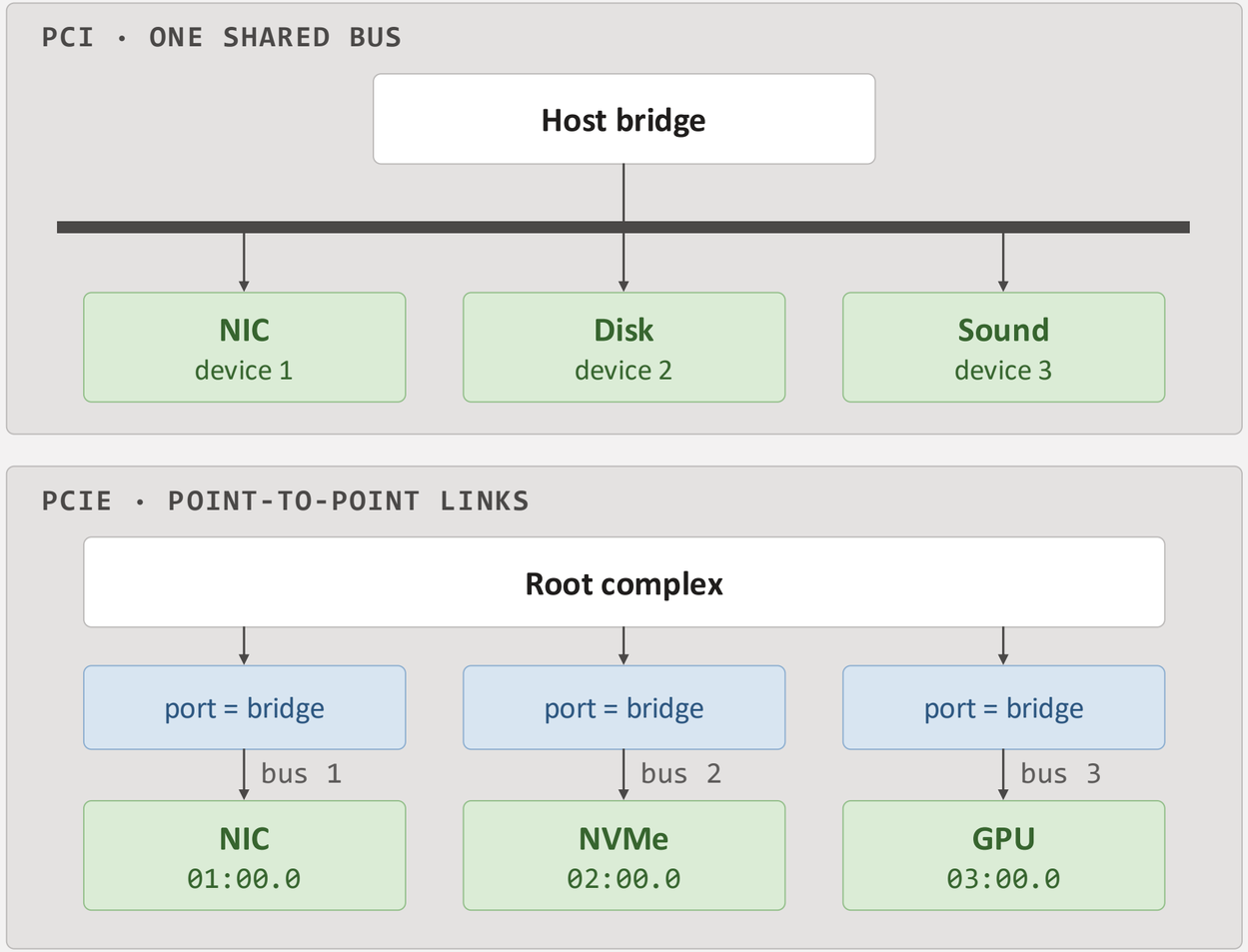
1. **Primary** = the bus being scanned.
2. **Secondary** = the next free number: the new bus.
3. **Subordinate** = 0xFF for now, so nothing below is cut off.
4. **Scan** the new bus; it may use more numbers.
5. **Close** subordinate = the highest bus actually used.

Primary / secondary / subordinate along the e1000's branch

STEP	ROOT PORT	BRIDGE 07:00.0
Root port found: bus 7	0 / 7 / 255	not reachable
Bus 7: a bridge, bus 8	0 / 7 / 255	7 / 8 / 255
Bus 8: e1000 at 08:01.0	0 / 7 / 255	7 / 8 / 8
Back on bus 0	0 / 7 / 8	7 / 8 / 8

```
/* reading 08:01.0 through ECAM */
0xB0000000 + (8 << 20) + (1 << 15)
root port 1c.0: range 7-8 -> forward
bridge 07:00.0: bus 8 is its own
              -> type 0 read to dev 1
```

Why PCIe is built this way



DEVICES DESCRIBE THEMSELVES

PCI (1992) replaced ISA's jumpers: firmware reads config space and hands out addresses that cannot clash.

LINKS REPLACED THE SHARED BUS

A shared parallel bus topped out near 133 MB/s for everyone; each serial link gives one device its own bandwidth.

LINKS STILL LOOK LIKE BUSES

Each port wears a PCI bridge header, so the enumeration code of 2003 worked unchanged.

NUMBERING IS DEPTH-FIRST

Every subtree gets a contiguous bus range, so a bridge routes with two registers and no tables.

PCIe pretends to be PCI: every link counts as a bus and every port as a bridge, so the old enumeration code never had to change.

03

BARs: sizing and placement

A BAR reports its size by refusing to store bits

1. **Turn decoding off.** Clear the memory and I/O enable bits in Command (0x04).
2. **Write all ones.** Save BAR0, then write 0xFFFFFFFF to it.
3. **Read it back.** The e1000 returns 0xFFFE0000: bits 16 to 0 stayed zero.
4. **Decode and restore.** Mask the 4 flag bits, invert, add 1: 0x20000 = 128 KB.

After writing 0xFFFFFFFF: reads 0xFFFE0000

31 … 17	16 … 4	3	2–1	0
1 1 1 … 1	0 0 0 … 0	0	00	0
writable address bits	wired to 0: the size	prefetch	32-bit	mem

```
size = NOT(0xFFFE0000 AND NOT 0xF) + 1 = 0x20000
```

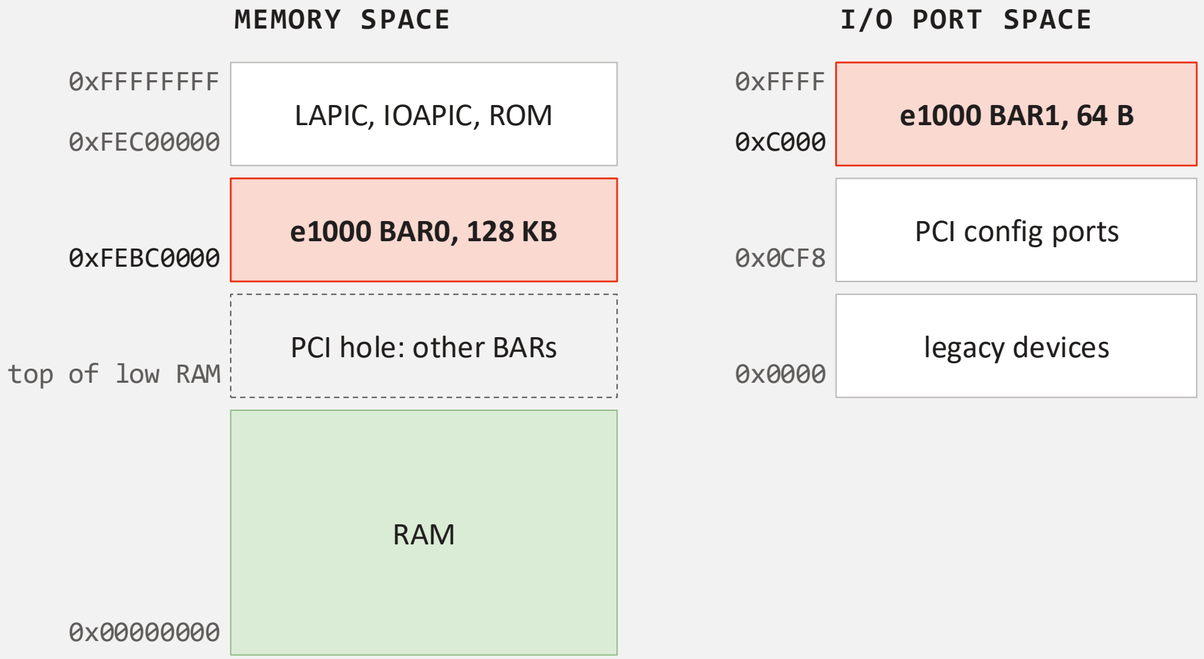
After the BIOS writes 0xFEBC0000

31 … 17	16 … 4	3	2–1	0
top 15 bits of 0xFEBC0000	0 0 0 … 0	0	00	0

The wired-zero bits force a 128 KB boundary: 0xFEBC0000 = 0x7F5E × 0x20000.

Where the BIOS places the BARs

1. **Sort** the memory requests largest first, so alignment wastes little space.
2. **Place memory BARs** in the PCI hole, below the interrupt controllers under 4 GB.
3. **Place I/O BARs** in the 64 KB port space; BAR1 lands at 0xC000.
4. **Program bridge windows** so each bridge forwards the ranges behind it.
5. **Write the BARs** and turn decoding back on in Command.



Not to scale. Labels mark where each region starts.

- The BIOS hands over no device list, only the programmed BARs and ACPI tables. Linux scans the bus again itself.

04

The driver and its registers

Linux rescans the bus and matches a driver

1. **Scan.** The PCI core reads every slot's header again.
2. **Measure.** It sizes each BAR itself and records a struct resource.
3. **Validate.** Each range must fit its bridge window; it goes into /proc/iomem.
4. **Publish.** pci_device_add() creates the sysfs entry and a uevent.
5. **Match.** The driver core compares IDs with every driver's table and calls probe().

THE DRIVER'S ID TABLE

```
static const struct pci_device_id e1000_pci_tbl[] = {
    INTEL_E1000_ETHERNET_DEVICE(0x1000),
    /* ... */
    INTEL_E1000_ETHERNET_DEVICE(0x100E), /* 82540EM */
    {0,}
};
```

WHAT THE BOOT LOG SHOWS

```
pci 0000:00:03.0: [8086:100e] type 00 class 0x020000
pci 0000:00:03.0: BAR 0 [mem 0xfebc0000-0xfebdffff]
pci 0000:00:03.0: BAR 1 [io  0xc000-0xc03f]
e1000 0000:00:03.0 eth0: (PCI:33MHz:32-bit)
    52:54:00:12:34:56
```

lspci shows what the scan found

```
$ lspci -nn -v -s 00:03.0
00:03.0 Ethernet controller [0200]: Intel Corporation 82540EM Gigabit Ethernet Controller [8086:100e] (rev 03)
        Subsystem: Red Hat, Inc. QEMU Virtual Machine [1af4:1100]
        Flags: bus master, fast devsel, latency 0, IRQ 11
        Memory at febc0000 (32-bit, non-prefetchable) [size=128K]
        I/O ports at c000 [size=64]
        Kernel driver in use: e1000
        Kernel modules: e1000
```

`[0200] [8086:100e]`

class and vendor:device IDs from the config header

`Memory at febc0000`

BAR0: 128 KB of registers, placed by the BIOS

`I/O ports at c000`

BAR1: the small I/O-port window

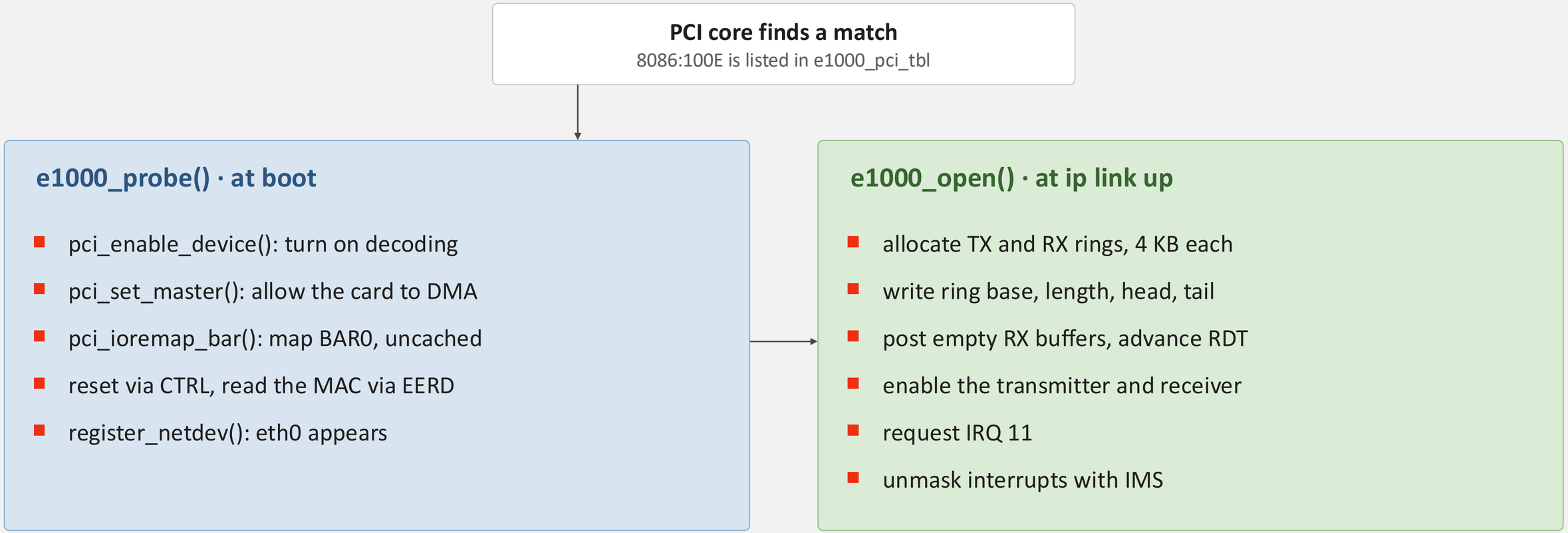
`bus master, IRQ 11`

Command bits the driver set, and the routed interrupt

`Kernel driver in use`

the PCI driver whose ID table matched

Probe makes the card reachable; open builds the data path



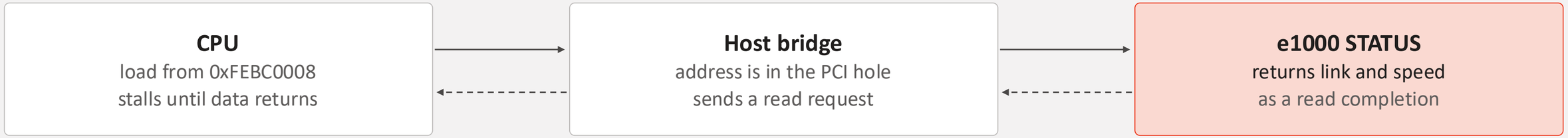
All control lives in one 128 KB register window

The driver maps BAR0 once with pci_ioremap_bar(), then uses readl() and writel() on offsets into it.

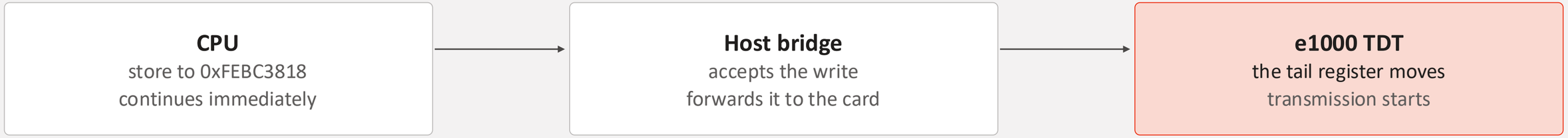
OFFSET	ADDRESS	REGISTERS	PURPOSE
0x0000	0xFEBC0000	CTRL, STATUS (0x0008)	reset, link setup; link up, speed
0x0014	0xFEBC0014	EERD	read a word from the EEPROM (the MAC address)
0x00C0	0xFEBC00C0	ICR	why the card interrupted; reading it clears it
0x00D0	0xFEBC00D0	IMS, IMC (0x00D8)	unmask and mask interrupt causes
0x0100	0xFEBC0100	RCTL	receiver on, receive buffer size
0x0400	0xFEBC0400	TCTL	transmitter on
0x2800	**0xFEBC2800**	**RDBAL, RDBAH, RDLEN, RDH, RDT**	**receive ring: base, length, head, tail**
0x3800	**0xFEBC3800**	**TDBAL, TDBAH, TDLEN, TDH, TDT**	**transmit ring: base, length, head, tail**
0x5400	0xFEBC5400	RAL0, RAH0	MAC address filter

An MMIO read stalls the CPU; a write is posted

READ · readl(STATUS)



WRITE · writel(TDT)

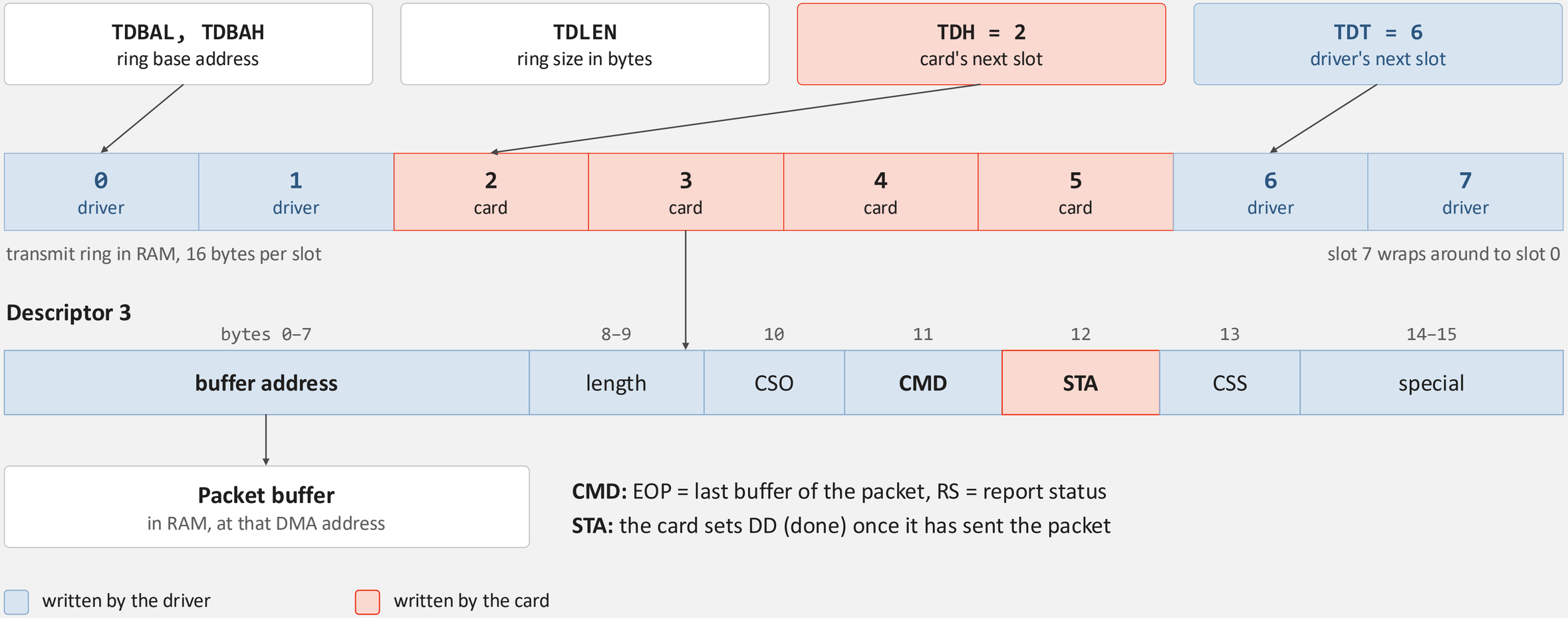


- Reads cost roughly 0.5–1 µs even on real hardware, so fast-path code writes registers but avoids reading them, and uses wmb() so data in RAM is visible before the doorbell.

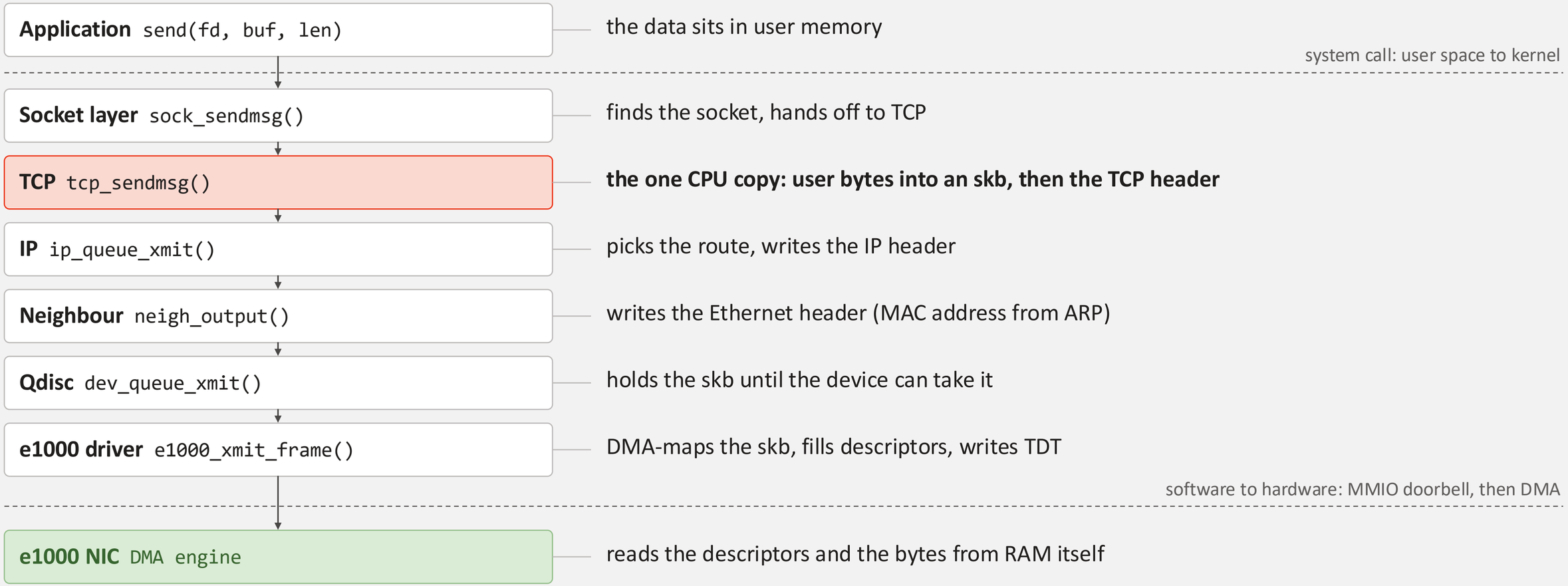
05

Rings and the packet path

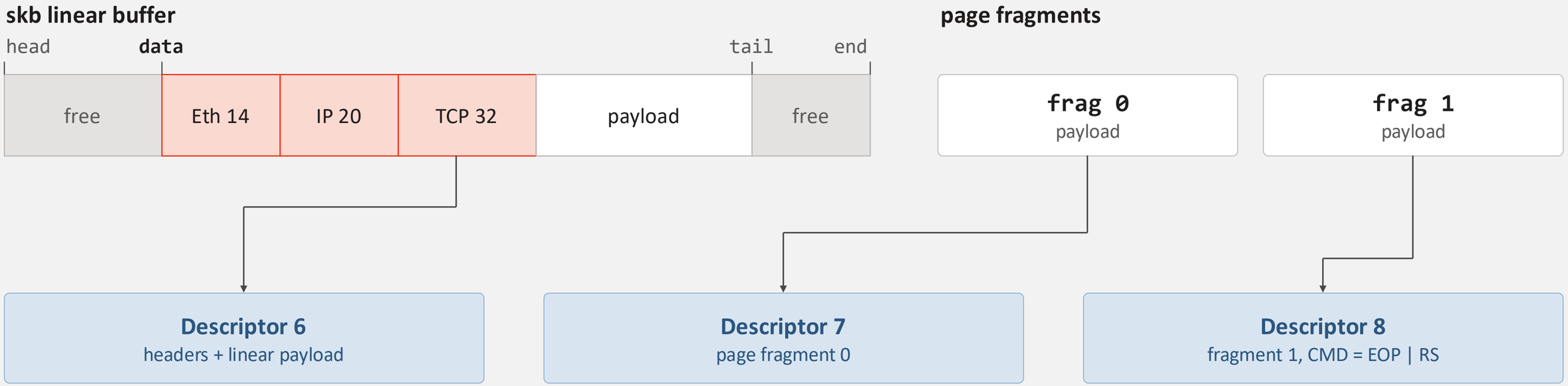
A ring: the card owns slots from head to tail – 1



From send() to the device: one copy, then DMA



Headers go into headroom; each piece becomes a descriptor

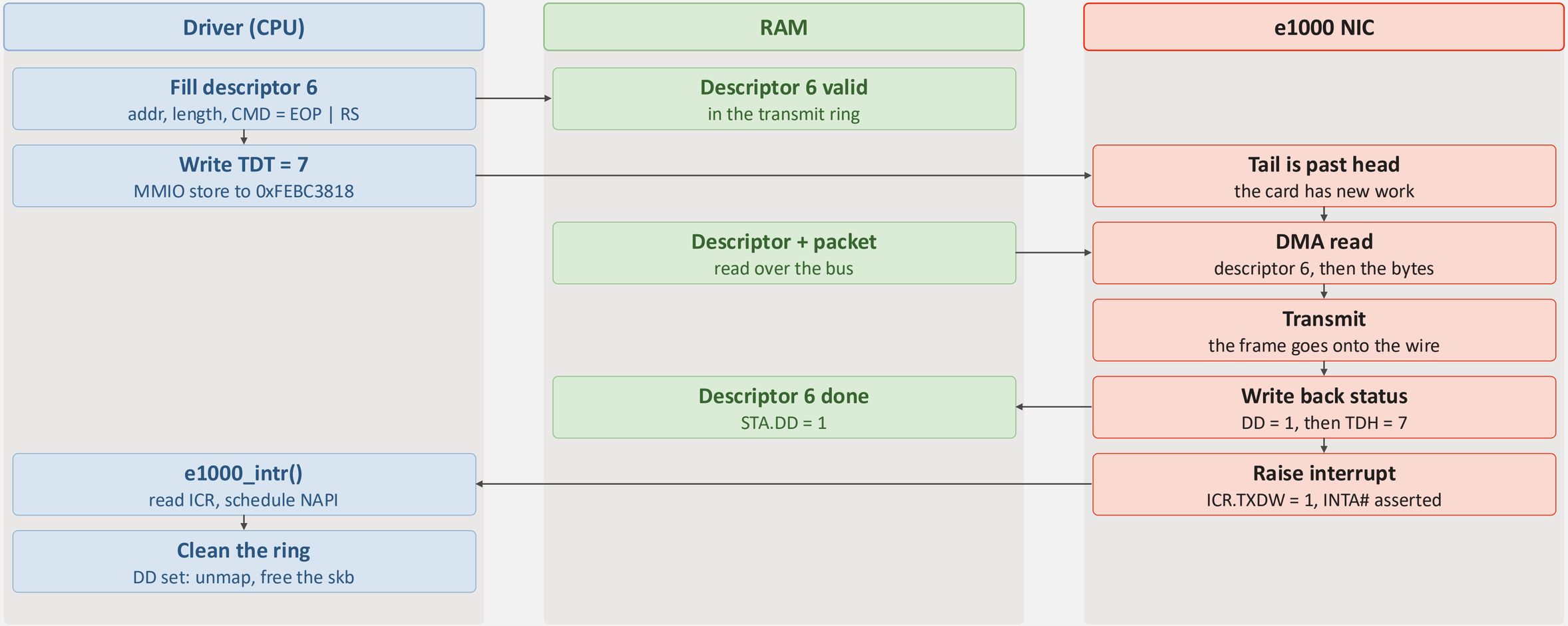


- Each piece is DMA-mapped and gets its own descriptor; only the last one carries EOP | RS.
- The headers were written in place: skb_push() moved the data pointer back, so no bytes were copied.

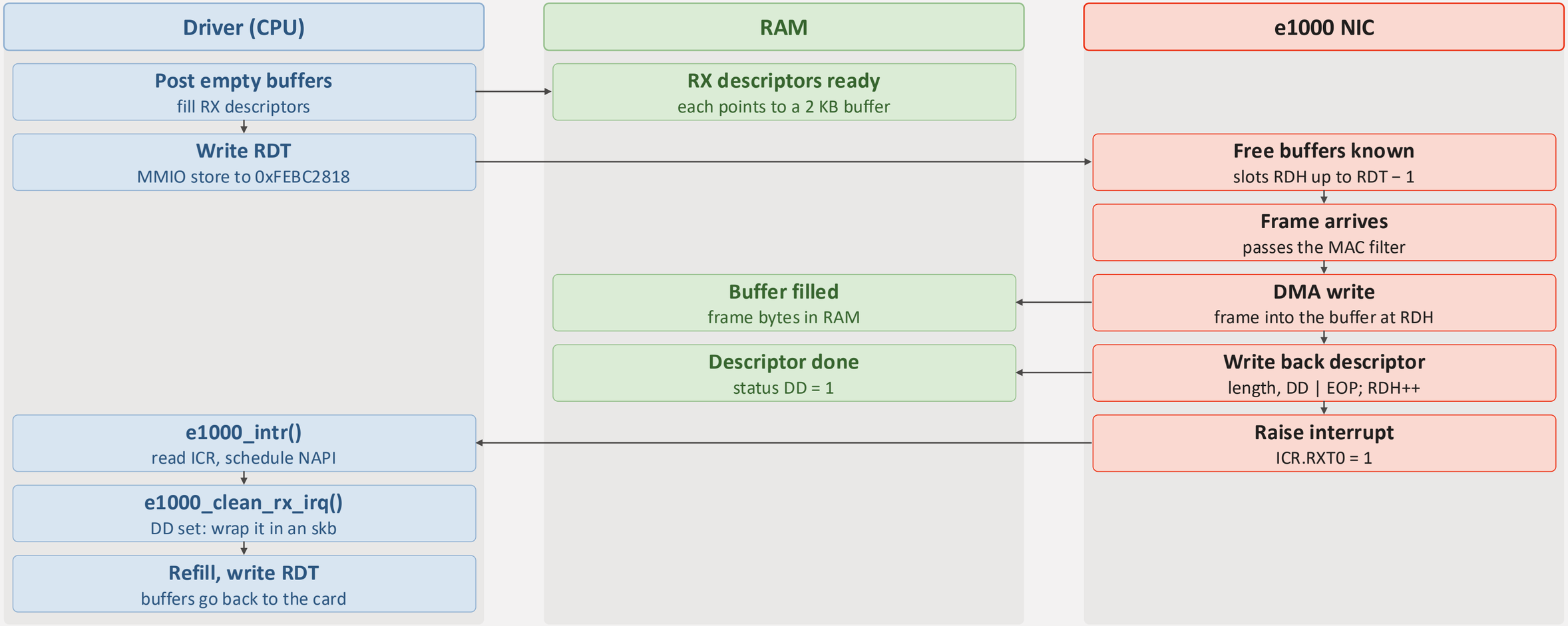
headers written by TCP, IP and Ethernet

unused head and tail room

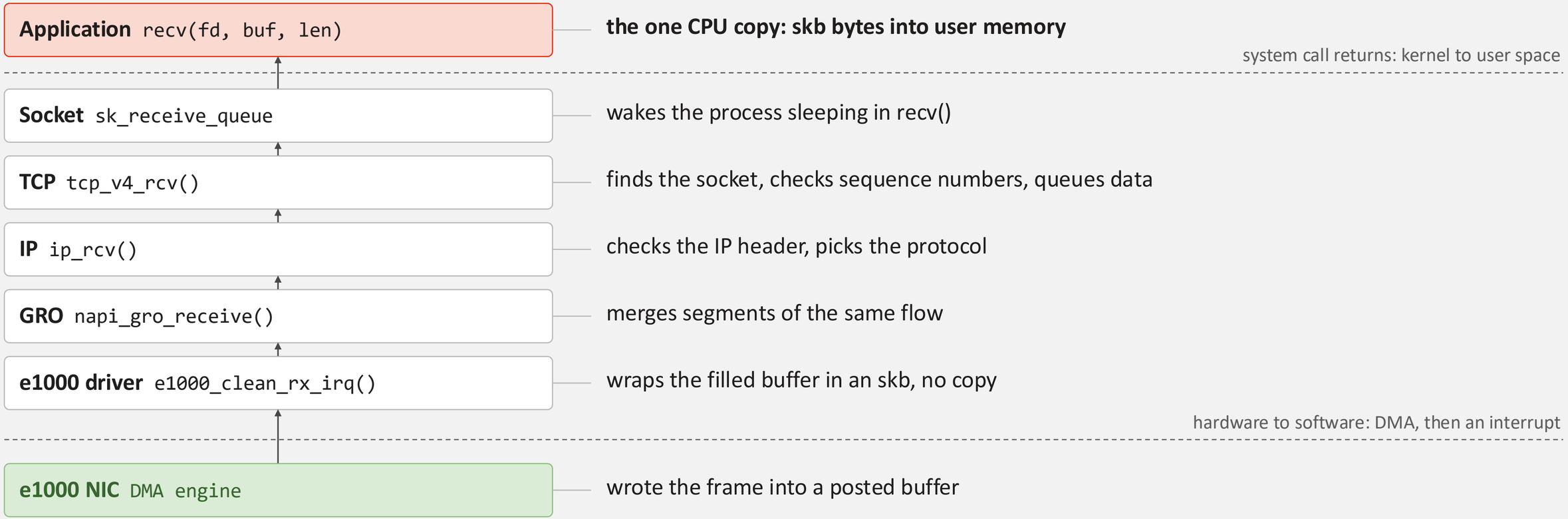
Transmit: one doorbell write, then the card works alone



Receive: the card fills buffers posted in advance



Receive: from the card up to recv()



06

Virtualizing a NIC

What an emulator has to reproduce

MECHANISM	ON REAL HARDWARE	WHAT THE EMULATOR MUST DO
Config space	a header in the card's silicon	answer every config read and write, including BAR sizing
Registers (BAR0)	logic with side effects	trap every access and reproduce its effect: a TDT write starts transmission
DMA	the card's DMA engine	read descriptors and packets from guest RAM, write received frames into it
Interrupt	the INTA# wire	inject a virtual interrupt into the guest

REGISTER ACCESSES PER PACKET, EACH A TRAP IN A VM

TDT

write to send

exit to QEMU, several µs

ICR

read on each interrupt

exit to QEMU, several µs

IMC

write to mask

exit to QEMU, several µs

IMS

write to unmask

exit to QEMU, several µs

07

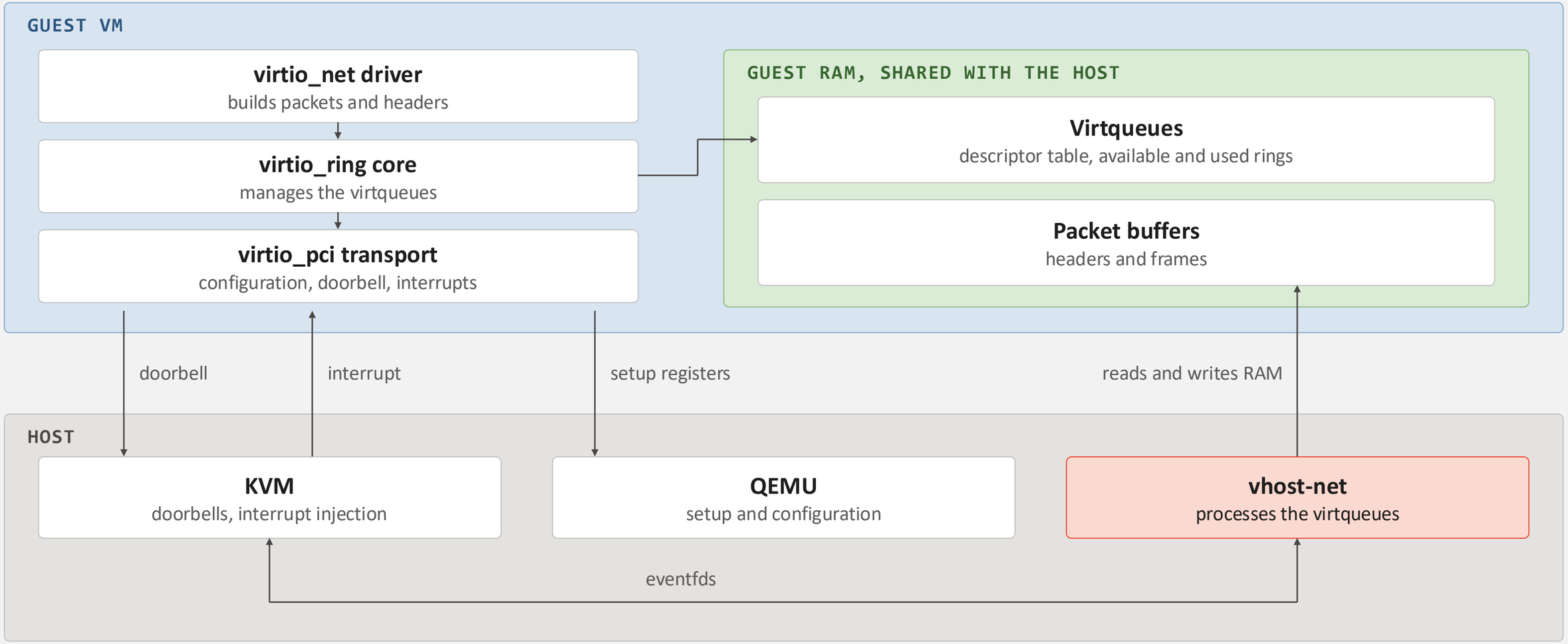
virtio-net

virtio-net keeps the rings and drops the register traffic

It is paravirtualized: the guest knows it is in a VM and uses a driver for a device that exists only in software. Sending a batch costs one doorbell write; receiving, one interrupt.

ON THE DATA PATH	EMULATED E1000	VIRTIO-NET
Announce new packets	write TDT: exit to QEMU	one doorbell write, handled inside KVM, often skipped
Find out why an interrupt came	read ICR: exit to QEMU	nothing to read: completions are in RAM
Mask and unmask interrupts	write IMC and IMS: exits to QEMU	flags and indexes in RAM, no exit
Who processes the rings	QEMU's model of the card	vhost-net, a thread in the host kernel

The layers: guest driver, shared RAM, host



Discovery: the same PCI, plus capabilities

CONFIG SPACE AT 0xB0100000

0x00	**Vendor:Device**	1AF4 : 1041
0x14	**BAR1**	0xFE800000, 4 KB
0x20	**BAR4**	0xFE600000, 16 KB
0x34	**Cap pointer**	first capability
cap	**MSI-X**	BAR1: table, PBA
cap	**virtio common**	BAR4 + 0x0000
cap	**virtio ISR**	BAR4 + 0x1000
cap	**virtio device**	BAR4 + 0x2000
cap	**virtio notify**	BAR4 + 0x3000, × 4

BAR WINDOWS (GUEST-PHYSICAL)

0xFE800000 MSI-X table
vector table and pending bits

0xFE600000 common config
features, status, queue setup

0xFE601000 ISR status
why the interrupt fired

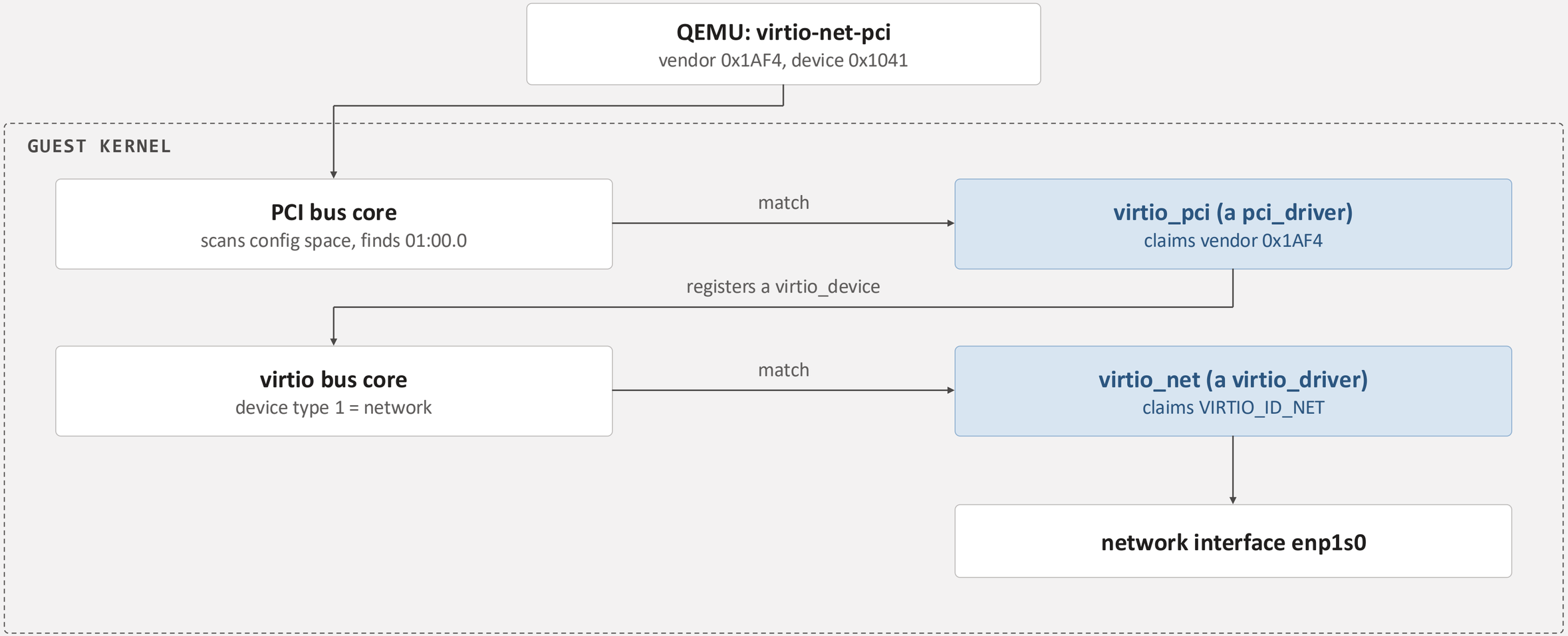
0xFE602000 device config
MAC, link status, queue pairs, MTU

0xFE603000 notification area
+0x0 queue 0 receive · +0x4 queue 1 transmit
+0x8 queue 2 control

access exits to QEMU

doorbell write, handled inside KVM

Two-stage matching: the PCI bus, then the virtio bus



Both stages from a shell

```
$ lspci -nn -v -s 01:00.0
01:00.0 Ethernet controller [0200]: Red Hat, Inc. Virtio 1.0 network device [1af4:1041] (rev 01)
        Memory at fe800000 (32-bit, non-prefetchable) [size=4K]
        Memory at fe600000 (64-bit, prefetchable) [size=16K]
        Capabilities: [98] MSI-X: Enable+ Count=4 Masked-
        Capabilities: [70] Vendor Specific Information: VirtIO: Notify
        Capabilities: [40] Vendor Specific Information: VirtIO: CommonCfg
        Kernel driver in use: virtio-pci
```

```
$ readlink /sys/bus/virtio/devices/virtio0/driver
../../../../bus/virtio/drivers/virtio_net
$ readlink -f /sys/class/net/enp1s0
/sys/devices/pci0000:00/0000:00:02.0/0000:01:00.0/virtio0/net/enp1s0
```

STAGE 1 · PCI

lspci sees only the PCI device: virtio-pci claimed vendor 1af4.

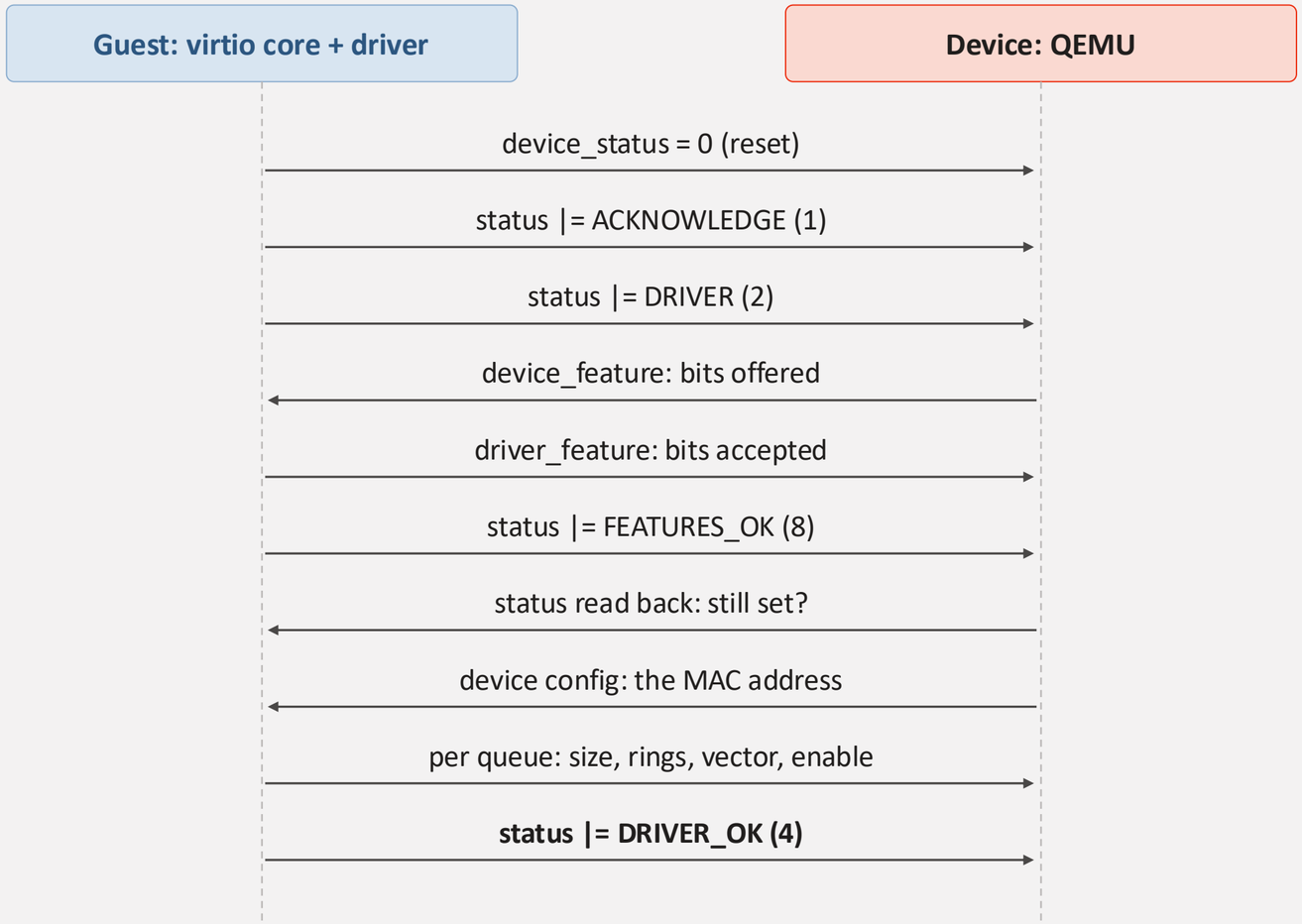
CAPABILITIES

the vendor-specific entries point to the structures in BAR4.

STAGE 2 · VIRTIO

the virtio device virtio0 is bound to virtio_net, with enp1s0 below it.

The handshake: status, features, queues



COMMON CONFIG AT 0xFE600000

OFFSET	FIELD
0x00–0x0C	feature windows (device, driver)
0x12	num_queues
0x14	device_status
0x16	queue_select
0x18	queue_size
0x1A	queue_msix_vector
0x1C	queue_enable
0x1E	queue_notify_off
0x20–0x30	the three ring addresses

The virtqueue: three parts, one writer each

Descriptor table

#	addr	len	flags	next
0	→ net header	12	NEXT	1
1	→ linear part	66	NEXT	2
2	→ page frag	1448	–	–
3	free			
4	free			

Available ring driver writes

flags 0	idx = 1		
0	–	–	–

ring[0] = first descriptor of the chain

Used ring device writes

flags 0	idx = 1	
id 0, len 0	–	–

on transmit, len is 0: nothing was written back

DESCRIPTOR TABLE

where the buffers are: address, length, flags, next

AVAILABLE RING

what the device should work on next

USED RING

what the device has finished, and how many bytes it wrote

Walking through a virtqueue

Step 1 of 8

DESCRIPTOR TABLE · DRIVER WRITES

#	buffer	len	flags
0	free		
1	free		
2	free		
3	free		

AVAILABLE RING · DRIVER WRITES

idx = 0				

device has read 0 of 0 available entries

USED RING · DEVICE WRITES

idx = 0				

driver has read 0 of 0 used entries

Start: all four descriptors are free, and both rings are empty with idx = 0. Each side has also read 0 entries of the other side's ring.

written by the driver written by the device changed in this step

Walking through a virtqueue

Step 2 of 8

DESCRIPTOR TABLE · DRIVER WRITES

#	buffer	len	flags
0	**packet A**	**1514**	
1	free		
2	free		
3	free		

AVAILABLE RING · DRIVER WRITES

idx = 0				

device has read 0 of 0 available entries

USED RING · DEVICE WRITES

idx = 0				

driver has read 0 of 0 used entries

The driver puts packet A's address and length into free descriptor 0. This is only a write to RAM: the device has no way to know about it yet.

written by the driver written by the device changed in this step

Walking through a virtqueue

Step 3 of 8

DESCRIPTOR TABLE · DRIVER WRITES

#	buffer	len	flags
0	packet A	1514	
1	free		
2	free		
3	free		

AVAILABLE RING · DRIVER WRITES

idx = 1	0			

device has read 0 of 1 available entries

USED RING · DEVICE WRITES

idx = 0				

driver has read 0 of 0 used entries

The driver publishes it: avail.ring[0] = 0 (the chain starts at descriptor 0), then avail.idx = 1. Then it rings the doorbell, or skips it if the device said it doesn't need one.

written by the driver written by the device changed in this step

Walking through a virtqueue

Step 4 of 8

DESCRIPTOR TABLE · DRIVER WRITES

#	buffer	len	flags
0	packet A	1514	
1	**B header**	**12**	**NEXT → 2**
2	**B data**	**1514**	
3	free		

AVAILABLE RING · DRIVER WRITES

idx = 2	0	**1**		

device has read 0 of 2 available entries

USED RING · DEVICE WRITES

idx = 0				

driver has read 0 of 0 used entries

Packet B needs two buffers, so the driver chains descriptor 1 (header) to descriptor 2 (data) with NEXT. It publishes only the first one: avail.ring[1] = 1, avail.idx = 2.

written by the driver · written by the device · changed in this step

Walking through a virtqueue

Step 5 of 8

DESCRIPTOR TABLE · DRIVER WRITES

#	buffer	len	flags
0	packet A	1514	
1	B header	12	NEXT → 2
2	B data	1514	
3	free		

AVAILABLE RING · DRIVER WRITES

idx = 2	0	1		

device has read 1 of 2 available entries

USED RING · DEVICE WRITES

idx = 1	**id 0**			

driver has read 0 of 1 used entries

The device sees avail.idx = 2 but has read 0 entries, so 2 are new. It reads avail.ring[0] = 0, sends packet A, then reports it: used.ring[0] = {id 0, len 0}, used.idx = 1. (len 0: on transmit it wrote nothing back.)

written by the driver written by the device changed in this step

Walking through a virtqueue

Step 6 of 8

DESCRIPTOR TABLE · DRIVER WRITES

#	buffer	len	flags
0	free		
1	B header	12	NEXT → 2
2	B data	1514	
3	free		

AVAILABLE RING · DRIVER WRITES

idx = 2	0	1		

device has read 1 of 2 available entries

USED RING · DEVICE WRITES

idx = 1	id 0			

driver has read 1 of 1 used entries

The driver sees used.idx = 1 but has read 0 used entries. It reads used.ring[0]: id 0, so descriptor 0 is done. It frees packet A and returns descriptor 0 to its free list.

written by the driver · written by the device · changed in this step

Walking through a virtqueue

Step 7 of 8

DESCRIPTOR TABLE · DRIVER WRITES

#	buffer	len	flags
0	free		
1	B header	12	NEXT → 2
2	B data	1514	
3	free		

AVAILABLE RING · DRIVER WRITES

idx = 2	0	1		

device has read 2 of 2 available entries

USED RING · DEVICE WRITES

idx = 2	id 0	**id 1**		

driver has read 1 of 2 used entries

The device takes the next new entry: avail.ring[1] = 1. It follows the chain 1 → 2, sends packet B, and reports the chain by its first descriptor: used.ring[1] = {id 1, len 0}, used.idx = 2.

written by the driver written by the device changed in this step

Walking through a virtqueue

Step 8 of 8

DESCRIPTOR TABLE · DRIVER WRITES

#	buffer	len	flags
0	free		
1	**free**		
2	**free**		
3	free		

AVAILABLE RING · DRIVER WRITES

idx = 2	0	1		

device has read 2 of 2 available entries

USED RING · DEVICE WRITES

idx = 2	id 0	id 1		

driver has read 2 of 2 used entries

The driver reads used.ring[1]: id 1, and frees the whole chain starting there, descriptors 1 and 2. All slots are free again; the idx counters keep counting up and wrap around the ring (position = idx mod 4).

written by the driver · written by the device · changed in this step

Compared with the e1000 ring

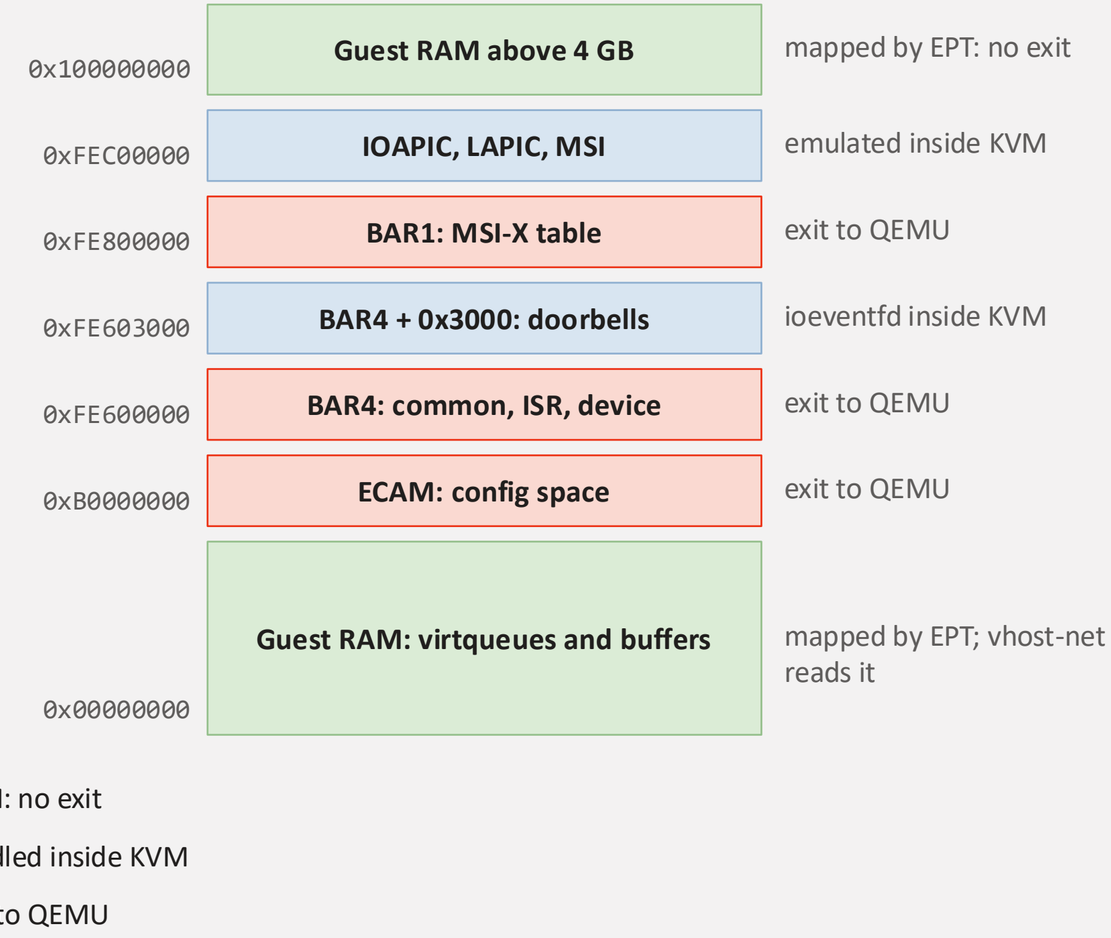
	E1000 RING	VIRTQUEUE
Handing over buffers	write TDT (MMIO)	write avail.idx in RAM, then at most one doorbell
Learning what finished	DD bits, plus an ICR read	used.idx and used entries in RAM
Completion order	in ring order	any order: the used ring names each chain
Multi-buffer packets	consecutive slots up to EOP	a chain linked by next, any free slots

THE 12-BYTE PACKET HEADER

```
struct virtio_net_hdr_v1 {
  u8   flags;       // NEEDS_CSUM
  u8   gso_type;    // NONE, TCPV4 ...
  le16 hdr_len;     // headers' bytes
  le16 gso_size;    // MSS for TSO
  le16 csum_start;
  le16 csum_offset;
  le16 num_buffers; // receive only
};
```

- **EVENT_IDX:** each side says exactly when it wants to be notified, so busy queues see few doorbells and interrupts.
- **Packed virtqueue (1.1):** one ring that the device overwrites as used; the same protocol with fewer memory accesses.

Three kinds of guest address

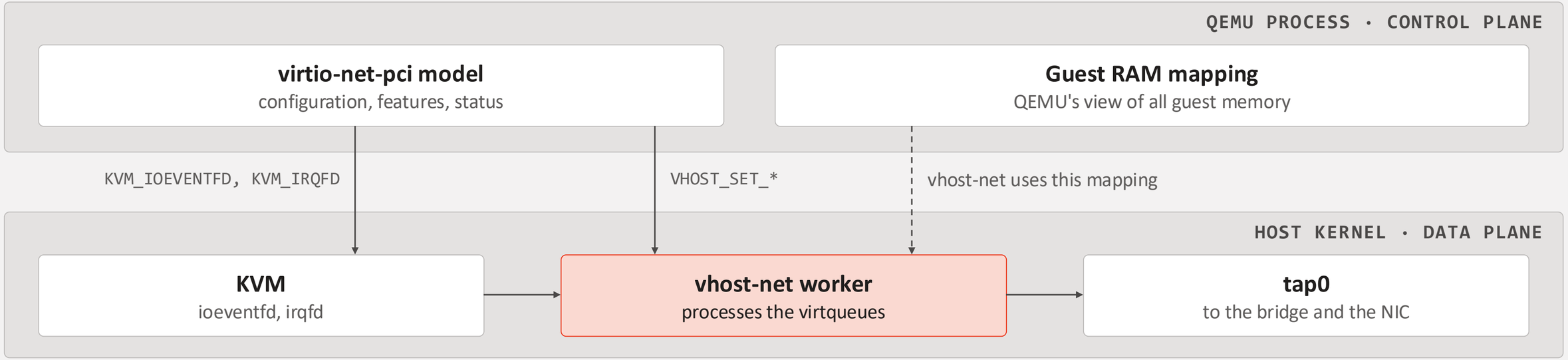


WHAT ONE TRAPPED ACCESS COSTS

1. **EPT fault.** A 1-byte load from 0xFE602000 (the MAC) finds no RAM and exits to KVM.
2. **To QEMU.** Not an ioeventfd, so KVM_RUN returns KVM_EXIT_MMIO.
3. **Emulate.** QEMU finds virtio-net's device config, offset 0: 0x52.
4. **Resume.** KVM completes the load and the guest continues.

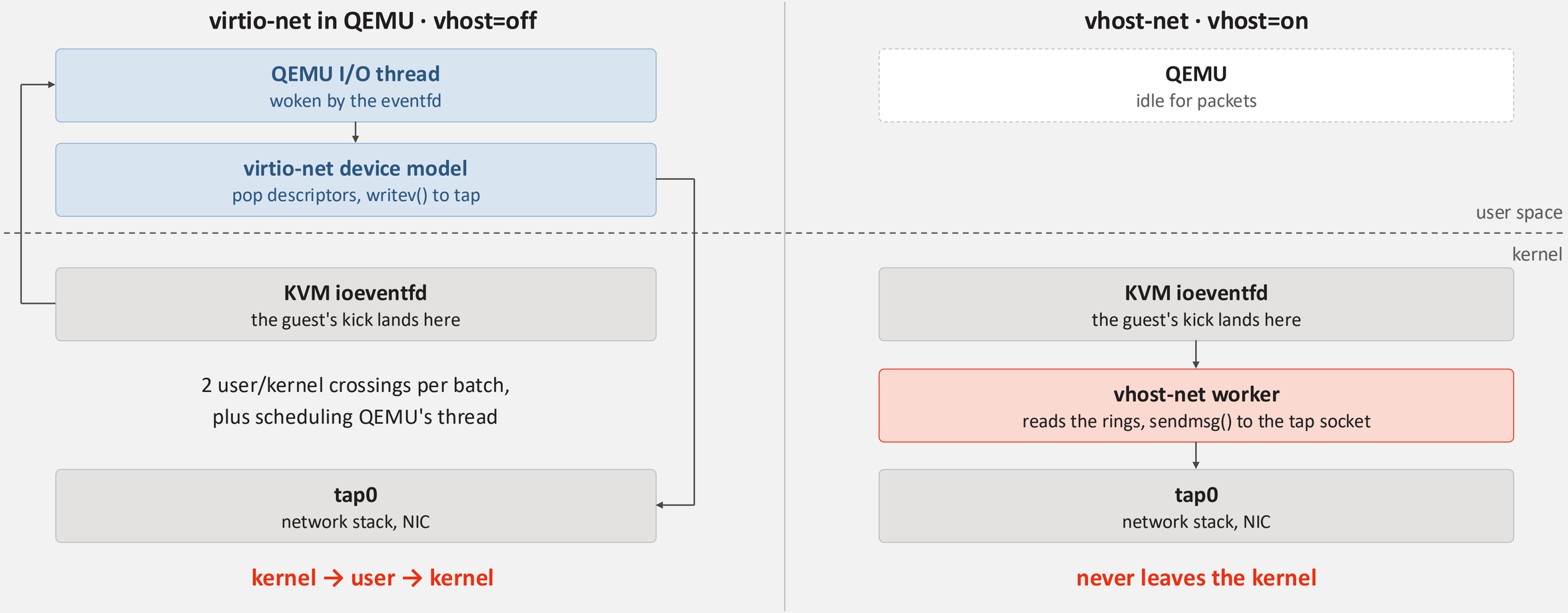
- Several µs per access: fine for setup, far too slow per packet.

QEMU sets things up; vhost-net moves the packets



CALL	WHAT IT HANDS OVER
`KVM_IOEVENTFD`	a 2-byte write to 0xFE603004 means: signal eventfd A
`KVM_IRQFD`	when eventfd B fires, inject the MSI from MSI-X entry 2
`VHOST_SET_MEM_TABLE`	guest-physical ranges to addresses in QEMU's process
`VHOST_SET_VRING_ADDR, _KICK, _CALL`	each queue's rings and its kick and call eventfds
`VHOST_NET_SET_BACKEND`	the tap device's file descriptor

Two backends for the same rings: QEMU or vhost-net



What QEMU still does when vhost-net is on

CONTROL PLANE

PCI config space, the status and feature handshake, MSI-X table writes: every access that traps as MMIO.

STARTING AND STOPPING VHOST

At DRIVER_OK it hands over the memory table, ring addresses and indexes, and eventfds. On reset or VM stop it takes the indexes back (VHOST_GET_VRING_BASE).

LIVE MIGRATION

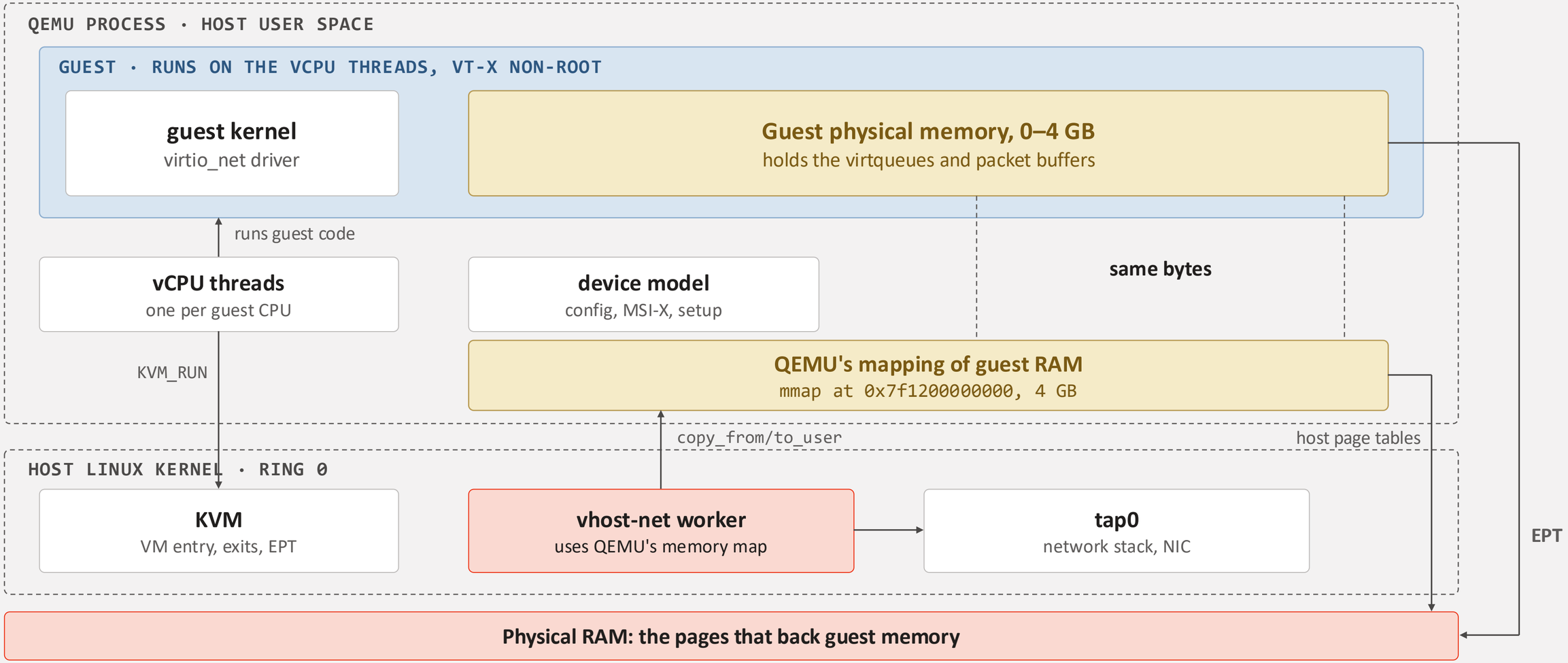
vhost writes guest RAM behind QEMU's back, so QEMU gives it a dirty-page log (VHOST_SET_LOG_BASE) and copies every page it marks.

FALLBACK

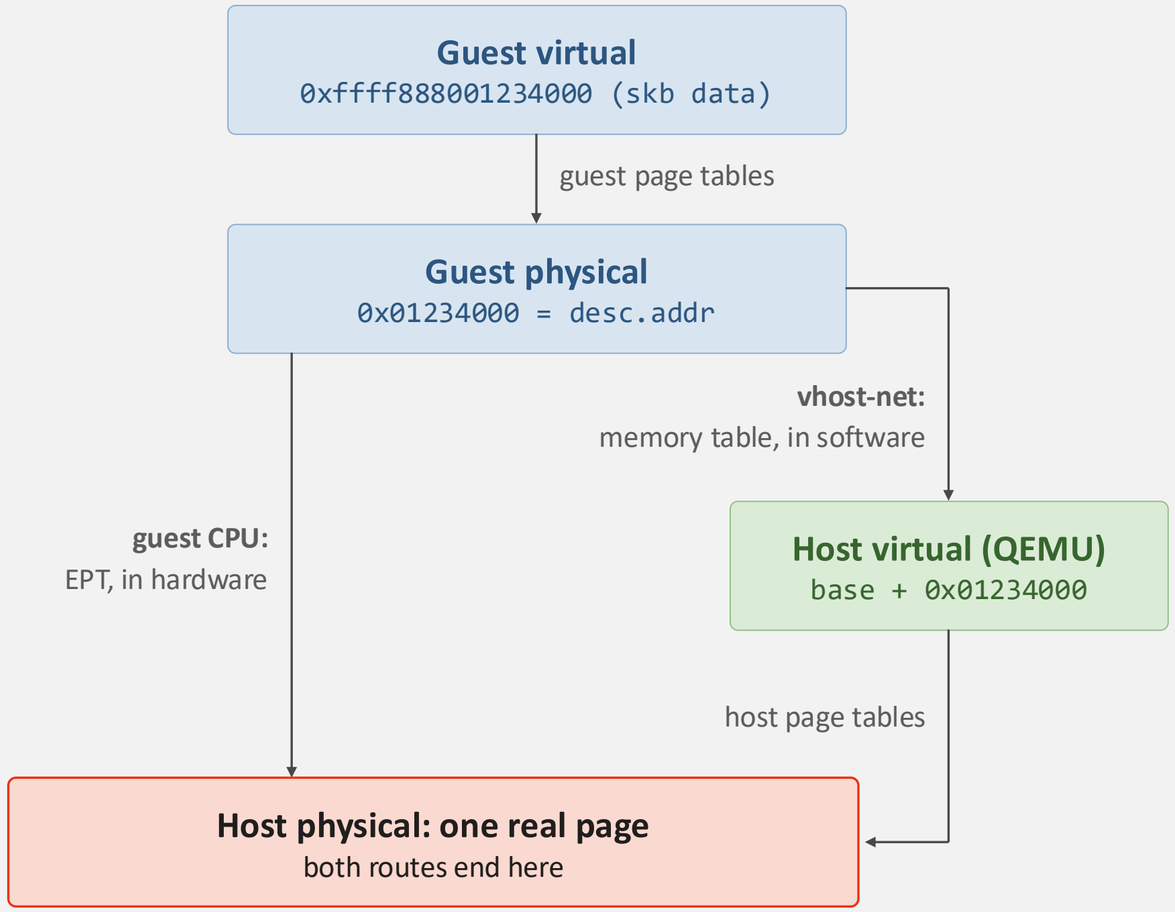
No /dev/vhost-net, or a feature vhost does not support: QEMU runs the device itself, as on the previous slide's left side.

- vhost-user: the same handoff to another user process (DPDK, Open vSwitch) over a Unix socket, with guest RAM shared as file descriptors.

The guest runs inside QEMU; guest RAM is QEMU memory



Two routes from desc.addr to the same page



SET UP ONCE, AT VM START

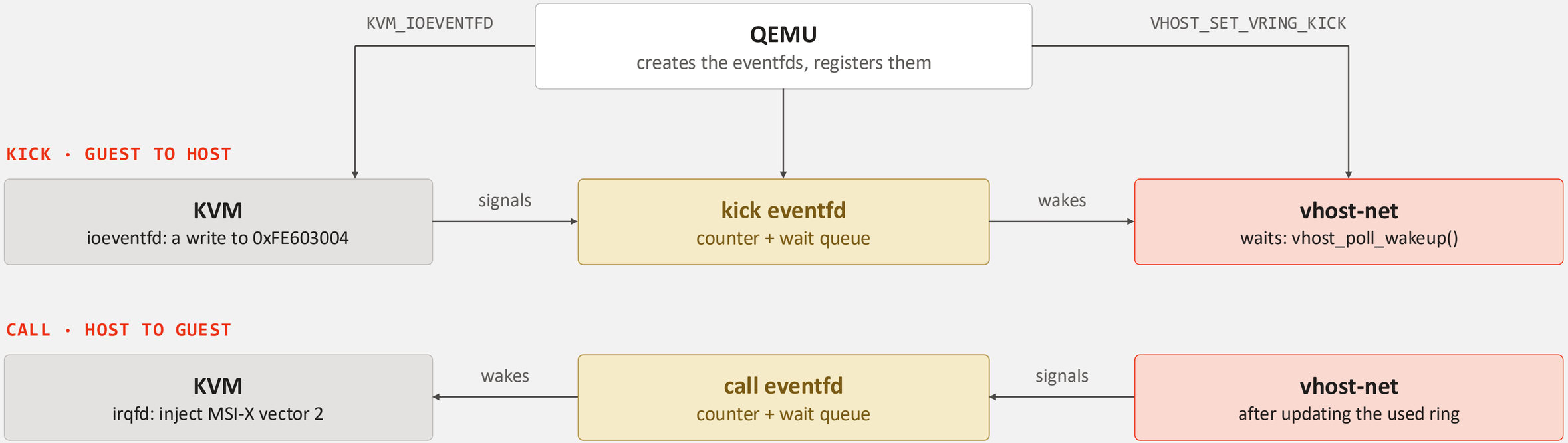
GUEST-PHYSICAL RANGE	QEMU ADDRESS
0x00000000–0x7FFFFFFF	base + offset
0x100000000–0x17FFFFFFF	base + 0x80000000 + offset

PER DESCRIPTOR, IN VHOST-NET

1. **Read** desc.addr: a guest-physical address.
2. **Look up** its region; HVA = region start + offset.
3. **Copy** with copy_from_user / copy_to_user in QEMU's address space.

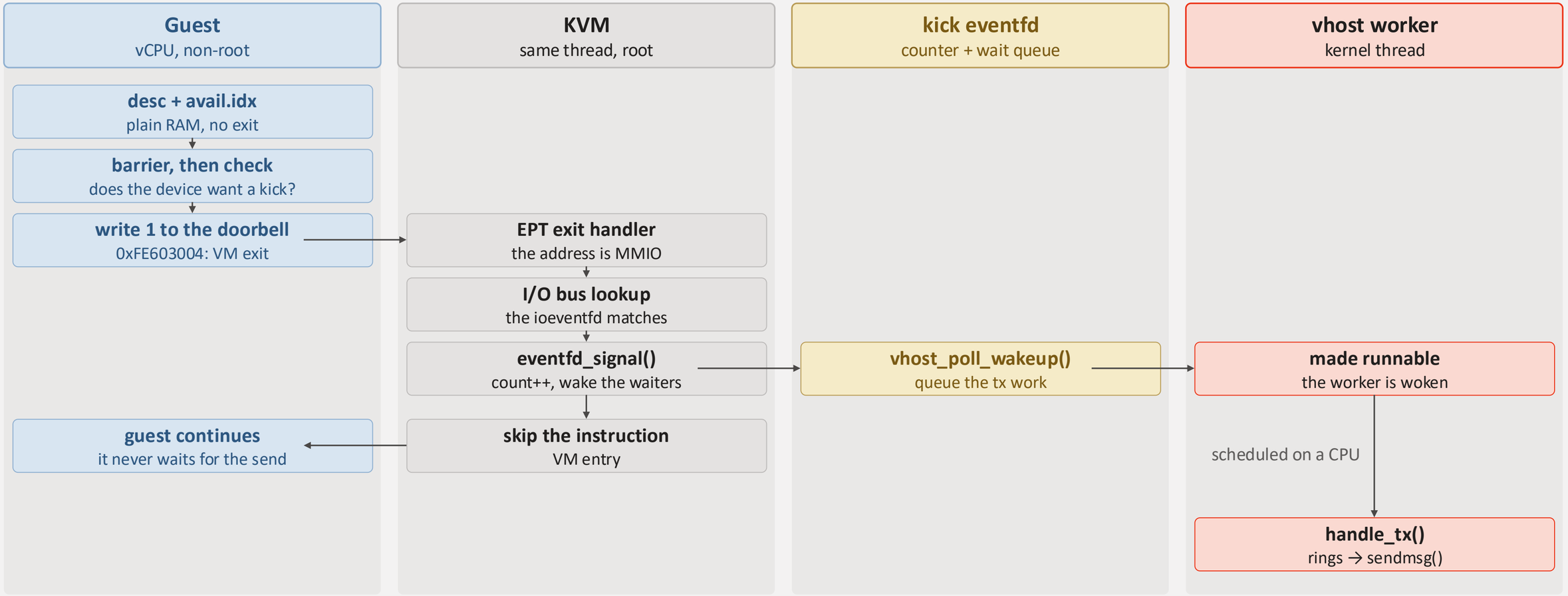
- Rings are shared pages, never copied. Packet bytes are copied once, unless zero-copy transmit pins the guest pages.

Wiring the kick: one eventfd, connected at DRIVER_OK



- QEMU wires the call side the same way, with VHOST_SET_VRING_CALL and KVM_IRQFD. From then on KVM and vhost-net talk through the eventfds, without QEMU.

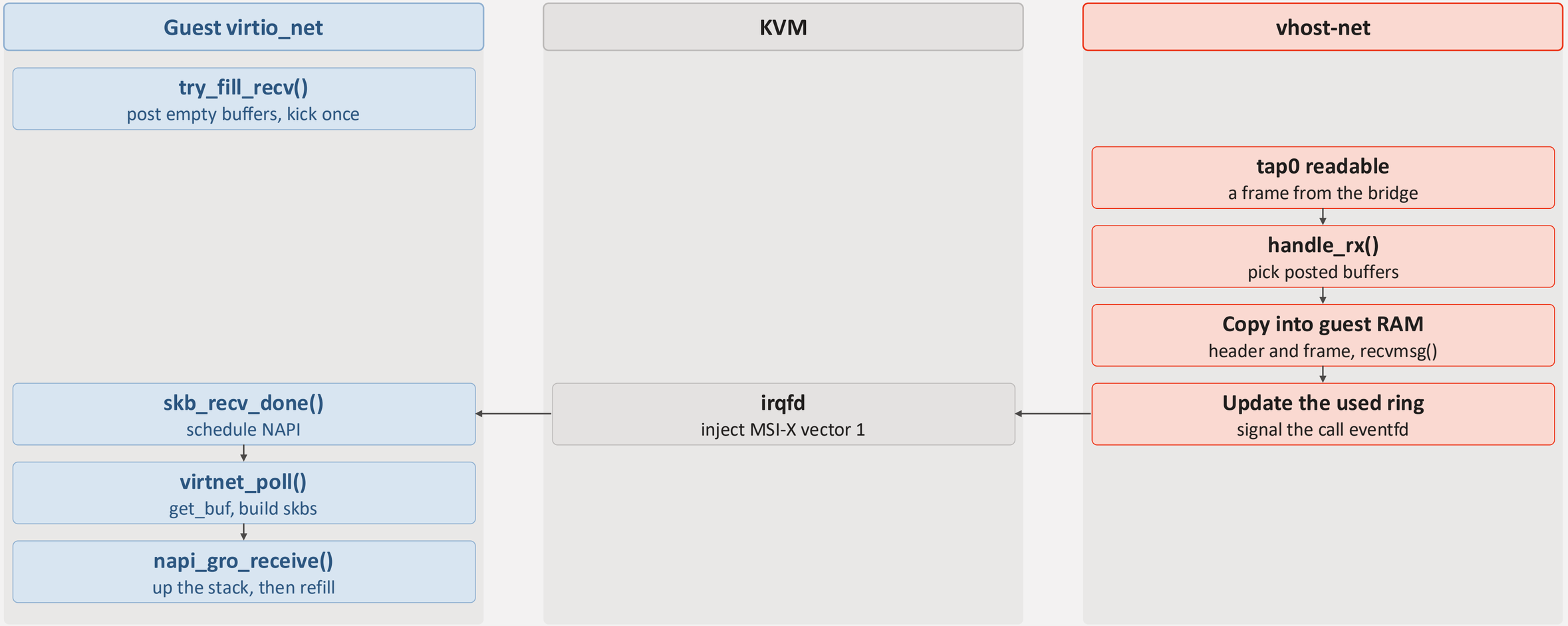
From the doorbell write to the vhost worker



Transmit with virtio-net: QEMU is never involved



Receive with virtio-net: one interrupt starts a batch



08

Takeaways

e1000 and virtio-net side by side

	E1000 (REAL OR EMULATED)	VIRTIO-NET
Discovery	PCI, ID 8086:100E, one driver	PCI, ID 1AF4:1041, then a virtio-bus match
Registers	128 KB in BAR0 at fixed offsets	four structures in BAR4, located by capabilities
Rings	TX and RX descriptor rings in RAM	virtqueues in RAM
Progress counters	head and tail registers on the card	avail.idx and used.idx in RAM
Telling the device	write TDT or RDT	one doorbell write, often skipped
Learning completions	DD bits, plus reading ICR	the used ring in RAM
Interrupt control	IMS and IMC registers	flags and EVENT_IDX in RAM
Data path in a VM	QEMU emulating the card, several exits per batch	vhost-net in the host kernel, about one exit per batch or none

Four things to remember

01

Devices describe themselves

Config space lets firmware and the OS find each device, size its BARs and place them. PCIe keeps the same model.

02

Registers control, RAM carries data

MMIO registers start and stop things; descriptor rings and DMA move the packets.

03

In a VM, register accesses are the cost

Each trapped access is a round trip to the emulator; an emulated e1000 pays several per packet.

04

virtio keeps the rings, drops the traps

Counters live in RAM, KVM handles the doorbell, and the host kernel processes the rings in vhost-net.