VirtIO PCI Modern Driver Implementation Guide

Date: January 9, 2026
Status: Implementation Required
Priority: CRITICAL (Blocks all network testing)

Table of Contents

Problem Statement

The Issue

The current VirtIO driver implementation uses **VirtIO MMIO Transport** register layout, but QEMU's `virtio-net-pci` device uses **VirtIO PCI Transport**. These are fundamentally different:

Transport	Register Access	Discovery	Layout
VirtIO MMIO	Direct memory-mapped at base address	Device Tree / ACPI	Fixed offsets (0x000, 0x004, etc.)
VirtIO PCI Legacy	I/O ports	PCI config space	Fixed offsets in I/O BAR
VirtIO PCI Modern	PCI Capabilities → MMIO regions	PCI capabilities	Multiple structures via caps

Current Symptom

```
[INIT] NIC MMIO base: 0x000000000000c000
[INIT] Initializing VirtIO-net driver...
[FAIL] VirtIO init error: feature negotiation failed
```

The driver reads from offset `0x000` expecting VirtIO MMIO magic (`0x74726976`), but:

- For Legacy PCI (device 0x1000): BAR0 is I/O space, needs `in`/`out` instructions
- For Modern PCI (device 0x1041): BAR0 points to MSI-X, actual config is via PCI capabilities

Root Cause

```
Current ASM (init.s):
    mov     eax, [rcx + VIRTIO_MMIO_MAGIC]  ; Reads memory at mmio_base+0
    cmp     eax, VIRTIO_MAGIC               ; Expects 0x74726976

Reality with VirtIO PCI:
    - Device 0x1000 (Legacy): BAR0 is I/O port (0xc001 → port 0xc000)
    - Device 0x1041 (Modern): BAR0 is MMIO but contains MSI-X tables, not
VirtIO regs
```

VirtIO PCI Modern Architecture

PCI Capabilities Chain

VirtIO PCI Modern devices expose configuration via **PCI Capabilities** (capability ID = 0x09 = `PCI_CAP_ID_VNDR`):

```
PCI Config Space Header (offset 0x00-0x3F)
    ├── Vendor ID: 0x1AF4 (Red Hat)
    ├── Device ID: 0x1041 (Modern VirtIO-net)
    ├── Command, Status, etc.
    └── Capabilities Pointer (offset 0x34) → First capability

Capability Chain:
    Cap 0: [VIRTIO_PCI_CAP_COMMON_CFG]  → Common configuration structure
    Cap 1: [VIRTIO_PCI_CAP_NOTIFY_CFG]  → Notification structure
    Cap 2: [VIRTIO_PCI_CAP_ISR_CFG]     → ISR status
    Cap 3: [VIRTIO_PCI_CAP_DEVICE_CFG]  → Device-specific config (MAC
address)
    Cap 4: [VIRTIO_PCI_CAP_PCI_CFG]     → Alternative config access
(optional)
```

Capability Structure (8+ bytes each)

```
struct virtio_pci_cap {
    u8 cap_vndr;     // 0x09 (PCI_CAP_ID_VNDR)
    u8 cap_next;     // Offset to next capability (0 = end)
    u8 cap_len;      // Length of this capability
    u8 cfg_type;     // Which structure: 1=common, 2=notify, 3=isr,
4=device, 5=pci
    u8 bar;          // BAR index (0-5)
    u8 id;           // Multiple of same type (usually 0)
    u8 padding[2];
    u32 offset;      // Offset within the BAR
    u32 length;      // Length of the structure
};
```

```
// Notify capability has extra field:
struct virtio_pci_notify_cap {
    struct virtio_pci_cap cap;
    u32 notify_off_multiplier;  // Queue notify offset = queue_index * multiplier
};
```

Common Configuration Structure (at BAR + offset)

```
struct virtio_pci_common_cfg {
    // Device info
    u32 device_feature_select;   // 0x00: Select feature bits (0=low, 1=high)
    u32 device_feature;          // 0x04: Read device features
    u32 driver_feature_select;   // 0x08: Select driver features
    u32 driver_feature;          // 0x0C: Write driver features
    u16 config_msix_vector;      // 0x10: MSI-X vector for config changes
    u16 num_queues;              // 0x12: Number of queues
    u8  device_status;           // 0x14: Device status
    u8  config_generation;       // 0x15: Config generation count

    // Queue configuration
    u16 queue_select;            // 0x16: Select queue
    u16 queue_size;              // 0x18: Queue size (max)
    u16 queue_msix_vector;       // 0x1A: MSI-X vector for queue
    u16 queue_enable;            // 0x1C: Enable queue
    u16 queue_notify_off;        // 0x1E: Notification offset
    u64 queue_desc;              // 0x20: Descriptor table addr (64-bit)
    u64 queue_driver;            // 0x28: Available ring addr
    u64 queue_device;            // 0x30: Used ring addr
    u16 queue_notify_data;       // 0x38: (optional) notify data
    u16 queue_reset;             // 0x3A: (optional) reset queue
};
```

Device-Specific Config (for virtio-net)

At BAR + device_cfg_offset:

```
struct virtio_net_config {
    u8  mac[6];                // 0x00: MAC address (if VIRTIO_NET_F_MAC)
    u16 status;                // 0x06: Link status (if VIRTIO_NET_F_STATUS)
    u16 max_virtqueue_pairs; // 0x08: (if VIRTIO_NET_F_MQ)
    u16 mtu;                   // 0x0A: MTU (if VIRTIO_NET_F_MTU)
    // ... more optional fields
};
```

Notification Mechanism

For Modern VirtIO, notification is:

```
notify_addr = BAR[notify_cap.bar] + notify_cap.offset
              + queue_notify_off * notify_off_multiplier

// Write queue index to notify_addr
mov     [notify_addr], queue_index
```

Implementation Plan

Phase 1: PCI Capability Parsing (New ASM + Rust)

Files to create:

- `asm/pci/capability.s` - Parse PCI capability chain
- `src/device/virtio_pci.rs` - VirtIO PCI capability discovery

ASM Functions:

```
; Parse capability chain, find VirtIO caps
asm_pci_read_cap_ptr         ; Read capabilities pointer from config
asm_pci_find_cap             ; Walk chain to find capability by type
asm_pci_read_cap_data        ; Read capability structure data

; VirtIO-specific capability parsing
asm_virtio_pci_find_common   ; Find COMMON_CFG capability
asm_virtio_pci_find_notify   ; Find NOTIFY_CFG capability
asm_virtio_pci_find_device   ; Find DEVICE_CFG capability
```

Rust Structures:

```rust
/// Parsed VirtIO PCI capabilities
pub struct VirtioPciCaps {
    /// BAR index and offset for common config
    pub common_cfg: Option<(u8, u32, u32)>,  // (bar, offset, length)
    /// BAR index, offset, and notify multiplier
    pub notify_cfg: Option<(u8, u32, u32, u32)>,
    /// BAR index and offset for ISR
    pub isr_cfg: Option<(u8, u32)>,
    /// BAR index and offset for device config
    pub device_cfg: Option<(u8, u32, u32)>,
}
```

Phase 2: Refactor ASM for VirtIO PCI Common Config

Files to modify:

- `asm/drivers/virtio/init.s` - Dual-mode (MMIO vs PCI)
- `asm/drivers/virtio/queue.s` - Dual-mode queue setup
- `asm/drivers/virtio/notify.s` - PCI notification style

Strategy: Create parallel functions for PCI access:

```
; Current (MMIO):
asm_virtio_get_status       ; Reads from mmio_base + 0x70

; New (PCI Modern):
asm_virtio_pci_get_status   ; Reads from common_cfg_base + 0x14

; Device operations need:
;   - common_cfg_base (from parsing)
;   - notify_base + multiplier (for notifications)
;   - device_cfg_base (for MAC address)
```

Phase 3: Rust Driver Integration

Files to modify:

- `src/driver/virtio/init.rs` - Detect MMIO vs PCI, use correct functions
- `src/driver/virtio/driver.rs` - Store PCI capability info
- `src/mainloop/bare_metal.rs` - Pass transport type info

New Flow:

```rust
pub unsafe fn virtio_net_init(
    device_info: &VirtioDeviceInfo, // Contains transport type
    config: &VirtioConfig,
) -> Result<...> {
    match device_info.transport {
        VirtioTransport::Mmio { base } => {
            init_mmio(base, config)
        }
        VirtioTransport::PciModern { caps, bars } => {
            init_pci_modern(&caps, &bars, config)
        }
    }
}
```

Phase 4: Bootloader Integration

Files to modify:

- `bootloader/src/tui/distro_downloader/commit_download.rs`
    - Parse PCI capabilities before EBS
    - Store capability offsets in BootHandoff

- Map required BARs

BootHandoff additions:

```
pub struct BootHandoff {
    // ... existing fields ...

    /// VirtIO transport type
    pub nic_transport: u8,  // 0=none, 1=mmio, 2=pci_modern

    /// For PCI Modern: common config location
    pub nic_common_bar: u8,
    pub nic_common_offset: u32,

    /// For PCI Modern: notify location
    pub nic_notify_bar: u8,
    pub nic_notify_offset: u32,
    pub nic_notify_multiplier: u32,

    /// For PCI Modern: device config location
    pub nic_device_bar: u8,
    pub nic_device_offset: u32,

    /// BAR base addresses (already mapped by UEFI)
    pub nic_bar_bases: [u64; 6],
}
```

Detailed File Changes

New Files

File	Lines	Purpose
`asm/pci/capability.s`	~250	PCI capability chain walking
`asm/drivers/virtio/pci_init.s`	~350	VirtIO PCI Modern init
`asm/drivers/virtio/pci_queue.s`	~300	PCI Modern queue setup
`src/device/virtio_pci.rs`	~200	PCI capability discovery (Rust)
`src/driver/virtio/pci_modern.rs`	~250	PCI Modern driver orchestration

Modified Files

File	Changes
`asm/drivers/virtio/notify.s`	Add `asm_vq_notify_pci` for PCI-style notify
`src/asm/drivers/virtio.rs`	Add PCI function bindings

File	Changes
`src/driver/virtio/init.rs`	Transport detection, routing
`src/driver/virtio/driver.rs`	Store transport info
`src/boot/handoff.rs`	Add PCI capability fields
`bootloader/.../commit_download.rs`	PCI capability parsing pre-EBS

ASM Function Specifications

capability.s

```
;
═══════════════════════════════════════════════════════════════════════════
; asm_pci_get_cap_ptr
;
═══════════════════════════════════════════════════════════════════════════
; Get capabilities pointer from PCI config space.
;
; Parameters:
;   RCX = PCI config base (ECAM address or use legacy I/O)
; Returns:
;   AL = Capabilities pointer (offset in config space), 0 if none
;
═══════════════════════════════════════════════════════════════════════════

;
═══════════════════════════════════════════════════════════════════════════
; asm_pci_find_virtio_cap
;
═══════════════════════════════════════════════════════════════════════════
; Walk capability chain to find VirtIO capability by type.
;
; Parameters:
;   RCX = PCI config base
;   DL  = VirtIO capability type to find (1=common, 2=notify, 4=device)
; Returns:
;   EAX = Capability offset (0 if not found)
;
═══════════════════════════════════════════════════════════════════════════

;
═══════════════════════════════════════════════════════════════════════════
; asm_pci_read_virtio_cap
;
═══════════════════════════════════════════════════════════════════════════
; Read VirtIO capability structure.
;
; Parameters:
;   RCX = PCI config base
;   DL  = Capability offset
;   R8  = Output buffer (16 bytes for virtio_pci_cap)
; Returns:
;   EAX = 0 success, 1 invalid
;
═══════════════════════════════════════════════════════════════════════════
```

pci_init.s

```
;
═══════════════════════════════════════════════════════════════════════════
; VirtIO PCI Modern Device Initialization
;
═══════════════════════════════════════════════════════════════════════════
;
; Unlike MMIO transport, PCI Modern accesses device via:
;   - Common Config structure (in a BAR, at parsed offset)
;   - Notification via different BAR/offset
;   - Device config (MAC) via yet another location
;
; Parameters for most functions:
;   RCX = common_cfg_base (BAR base + common offset)
;
═══════════════════════════════════════════════════════════════════════════

; Status register at common_cfg + 0x14
VIRTIO_PCI_STATUS           equ 0x14

; Feature select/read at common_cfg + 0x00/0x04
VIRTIO_PCI_DEVICE_FEATURE_SEL equ 0x00
VIRTIO_PCI_DEVICE_FEATURE     equ 0x04
VIRTIO_PCI_DRIVER_FEATURE_SEL equ 0x08
VIRTIO_PCI_DRIVER_FEATURE     equ 0x0C

; Queue config at common_cfg + 0x16
VIRTIO_PCI_QUEUE_SELECT     equ 0x16
VIRTIO_PCI_QUEUE_SIZE       equ 0x18
VIRTIO_PCI_QUEUE_ENABLE     equ 0x1C
VIRTIO_PCI_QUEUE_NOTIFY_OFF equ 0x1E
VIRTIO_PCI_QUEUE_DESC       equ 0x20
VIRTIO_PCI_QUEUE_DRIVER     equ 0x28
VIRTIO_PCI_QUEUE_DEVICE     equ 0x30

;
═══════════════════════════════════════════════════════════════════════════
; asm_virtio_pci_get_status
;
═══════════════════════════════════════════════════════════════════════════
; Read device status from PCI common config.
;
; Parameters:
```

```
;   RCX = common_cfg_base
; Returns:
;   AL = status
;
════════════════════════════════════════════════════════════════════════════
asm_virtio_pci_get_status:
    movzx   eax, byte [rcx + VIRTIO_PCI_STATUS]
    ret

;
════════════════════════════════════════════════════════════════════════════
; asm_virtio_pci_set_status
;
════════════════════════════════════════════════════════════════════════════
; Write device status to PCI common config.
;
; Parameters:
;   RCX = common_cfg_base
;   DL  = status
; Returns: None
;
════════════════════════════════════════════════════════════════════════════
asm_virtio_pci_set_status:
    mov     [rcx + VIRTIO_PCI_STATUS], dl
    mfence
    ret

;
════════════════════════════════════════════════════════════════════════════
; asm_virtio_pci_read_features
;
════════════════════════════════════════════════════════════════════════════
; Read 64-bit device features via feature select.
;
; Parameters:
;   RCX = common_cfg_base
; Returns:
;   RAX = 64-bit features
;
════════════════════════════════════════════════════════════════════════════
asm_virtio_pci_read_features:
    ; Read low 32 bits (feature_sel = 0)
    xor     eax, eax
    mov     [rcx + VIRTIO_PCI_DEVICE_FEATURE_SEL], eax
    mfence
    mov     r8d, [rcx + VIRTIO_PCI_DEVICE_FEATURE]

    ; Read high 32 bits (feature_sel = 1)
    mov     eax, 1
    mov     [rcx + VIRTIO_PCI_DEVICE_FEATURE_SEL], eax
    mfence
    mov     eax, [rcx + VIRTIO_PCI_DEVICE_FEATURE]

    ; Combine: RAX = (high << 32) | low
    shl     rax, 32
    or      rax, r8
    ret

; ... more functions
```

Testing Strategy

Step 1: QEMU Configuration

Force VirtIO Modern by adding to QEMU command:

```
-device virtio-net-pci,netdev=net0,...,disable-legacy=on
```

Device should now appear as 0x1041 with MMIO BARs and capabilities.

Step 2: Verify Capability Detection

Add debug output in bootloader:

```
// Parse and print VirtIO capabilities
let caps = parse_virtio_pci_caps(pci_config_base);
screen.put_str_at(9, log_y, &format!(
    "Common: BAR{} @ {:#x}",
    caps.common_cfg.0, caps.common_cfg.1
), ...);
```

Step 3: Incremental Testing

1. Read device status via PCI common config → Should return 0
2. Write ACKNOWLEDGE → Status should be 0x01
3. Read features → Should include standard net features
4. Full init sequence → Driver OK status

Implementation Order

Day	Task	Deliverable
1	PCI capability parsing ASM	`asm/pci/capability.s` working
1	Rust capability structures	`src/device/virtio_pci.rs`
2	PCI init ASM	`asm/drivers/virtio/pci_init.s`
2	PCI queue setup ASM	`asm/drivers/virtio/pci_queue.s`

Day	Task	Deliverable
3	Rust driver integration	Modified `init.rs`, `driver.rs`
3	Bootloader capability parsing	Modified `commit_download.rs`
4	BootHandoff updates	Add PCI fields
4	End-to-end testing	DHCP working in QEMU

References

- VirtIO Specification v1.2: https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html
    - §4.1: PCI Transport
    - §4.1.4: PCI Device Layout
    - §4.1.4.3: Common configuration structure
- QEMU VirtIO Implementation: https://github.com/qemu/qemu/blob/master/hw/virtio/virtio-pci.c
- Linux VirtIO PCI Driver: https://github.com/torvalds/linux/blob/master/drivers/virtio/virtio_pci_modern.c

Quick Reference: Register Offsets

VirtIO MMIO (Current - For Reference)

```
0x000 Magic        0x004 Version     0x008 DeviceID   0x00C VendorID
0x010 DevFeatures  0x014 DevFeatSel  0x020 DrvFeatures 0x024 DrvFeatSel
0x030 QueueSel     0x034 QueueNumMax 0x038 QueueNum    0x044 QueueReady
0x050 QueueNotify  0x060 IntStatus   0x064 IntAck      0x070 Status
0x080 QDescLow     0x084 QDescHigh   0x090 QDrvLow     0x094 QDrvHigh
0x0A0 QDevLow      0x0A4 QDevHigh    0x100+ Config
```

VirtIO PCI Common Config (NEW - Need to Implement)

```
0x00 DeviceFeatureSel  0x04 DeviceFeature
0x08 DriverFeatureSel  0x0C DriverFeature
0x10 ConfigMsixVector  0x12 NumQueues
0x14 DeviceStatus      0x15 ConfigGeneration
0x16 QueueSelect       0x18 QueueSize
0x1A QueueMsixVector   0x1C QueueEnable
0x1E QueueNotifyOff    0x20 QueueDesc (64-bit)
0x28 QueueDriver (64-bit)  0x30 QueueDevice (64-bit)
```

VirtIO Block Device Integration

Current Status

The VirtIO-blk driver is **already implemented** but **not wired up** to the main loop!

Existing Files:

- `asm/drivers/virtio/blk.s` - Full ASM implementation (503 lines)
- `src/driver/virtio_blk.rs` - Rust driver wrapper (604 lines)
- `src/driver/block_traits.rs` - `BlockDriver` trait

What's Missing:

1. VirtIO-blk device discovery and probe in bootloader
2. Block device initialization in `bare_metal_main()`
3. ISO streaming write during HTTP download

Block Device Flow (What Needs to Be Added)

```
┌─────────────────────────────────────────────────────────────────────┐
│                         bare_metal_main()                           │
├─────────────────────────────────────────────────────────────────────┤
│                                                                     │
│  1. Initialize VirtIO-net (ALREADY EXISTS - needs PCI fix)          │
│                                                                     │
│  2. Initialize VirtIO-blk (NEEDS IMPLEMENTATION)                    │
│        ┌──────────────────────────────────────────────────────────┐ │
│        │ a. Read blk_mmio_base from BootHandoff                   │ │
│        │ b. Create VirtioBlkConfig with DMA region                │ │
│        │ c. Call VirtioBlkDriver::new()                           │ │
│        │ d. Verify disk capacity >= ISO size                      │ │
│        └──────────────────────────────────────────────────────────┘ │
│                                                                     │
│  3. Run DHCP (ALREADY EXISTS)                                       │
│                                                                     │
│  4. Stream HTTP Download → Disk (NEEDS IMPLEMENTATION)              │
│        ┌──────────────────────────────────────────────────────────┐ │
│        │ HTTP GET response body is received in chunks             │ │
│        │                                                          │ │
│        │ For each chunk:                                          │ │
│        │   a. Wait for block driver can_submit()                  │ │
│        │   b. Calculate sector from offset (offset / 512)         │ │
│        │   c. submit_write(sector, chunk_phys, num_sectors)       │ │
│        │   d. Call notify()                                       │ │
│        │   e. Poll for completion with poll_completion()          │ │
│        │   f. Update progress                                     │ │
│        └──────────────────────────────────────────────────────────┘ │
│                                                                     │
│  5. Verify (optional SHA256 check)                                  │
│                                                                     │
│  6. Boot from ISO or halt                                           │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```

Block Driver Integration Code (To Add)

In `bare_metal_main()`:

```
// ═══════════════════════════════════════════════════════════════════════
// STEP 2b: INITIALIZE BLOCK DEVICE
// ═══════════════════════════════════════════════════════════════════════
serial_println("[INIT] Checking for VirtIO-blk device...");

let mut blk_driver = if handoff.blk_type == BLK_TYPE_VIRTIO &&
handoff.blk_mmio_base != 0 {
    serial_print("[INIT] Block device MMIO base: ");
    serial_print_hex(handoff.blk_mmio_base);
    serial_println("");

    // Calculate DMA offsets for block driver (separate from network)
    // Network uses first 512KB, block uses next 512KB
    let blk_dma_offset = 512 * 1024;
    let blk_dma_base = handoff.dma_cpu_ptr + blk_dma_offset;

    let blk_config = VirtioBlkConfig {
        queue_size: 32,
        desc_phys: blk_dma_base,
        avail_phys: blk_dma_base + 32 * 16, // After descriptors
        used_phys: blk_dma_base + 32 * 16 + 4 + 32 * 2, // After avail ring
        headers_phys: blk_dma_base + 0x1000, // Page-aligned
        status_phys: blk_dma_base + 0x1000 + 32 * 16, // After headers
        headers_cpu: (blk_dma_base + 0x1000) as u64,
        status_cpu: (blk_dma_base + 0x1000 + 32 * 16) as u64,
        notify_addr: handoff.blk_mmio_base + 0x50, // MMIO notify offset
    };

    match unsafe { VirtioBlkDriver::new(handoff.blk_mmio_base, blk_config)
} {
        Ok(driver) => {
            let info = driver.info();
            serial_print("[OK] Block device: ");
            serial_print_dec(info.total_sectors * info.sector_size as u64 /
(1024*1024));
            serial_println(" MB");
            Some(driver)
        }
        Err(e) => {
            serial_println("[WARN] Block device init failed, writes
disabled");
            None
        }
    }
} else {
    serial_println("[INFO] No block device configured");
    None
};
```

Streaming Write During Download:

```rust
// In the HTTP download state machine, when receiving body chunks:
fn handle_body_chunk(
    chunk: &[u8],
    offset: u64,
    blk_driver: &mut Option<VirtioBlkDriver>,
    write_buffer_phys: u64,  // Pre-allocated DMA buffer
) -> Result<(), WriteError> {
    let Some(driver) = blk_driver.as_mut() else {
        // No block device - just accumulate in memory or skip
        return Ok(());
    };

    // Copy chunk to DMA buffer
    unsafe {
        core::ptr::copy_nonoverlapping(
            chunk.as_ptr(),
            write_buffer_phys as *mut u8,
            chunk.len(),
        );
    }

    // Calculate sector range
    let sector = offset / 512;
    let num_sectors = (chunk.len() + 511) / 512;

    // Submit write
    driver.submit_write(
        sector,
        write_buffer_phys,
        num_sectors as u32,
        (offset / 512) as u32, // Use sector as request ID
    )?;

    // Notify device
    driver.notify();

    // Poll for completion (non-blocking, fire-and-forget style)
    // Completions collected in main loop

    Ok(())
}
```

BootHandoff Block Device Fields (Already Exists!)

```rust
/// Block device MMIO base address (from PCI BAR)
pub blk_mmio_base: u64,
/// Block device PCI bus number
pub blk_pci_bus: u8,

/// Block device PCI device number
pub blk_pci_device: u8,

/// Block device PCI function number
pub blk_pci_function: u8,

/// Block device type: 0=None, 1=VirtIO-blk, 2=NVMe, 3=AHCI
pub blk_type: u8,

/// Block device sector size (typically 512)
pub blk_sector_size: u32,

/// Block device total sectors
pub blk_total_sectors: u64,
```

Bootloader Changes Needed

In `commit_download.rs`:

```
// After probing NIC, also probe for VirtIO-blk
fn probe_virtio_blk_with_debug(screen: &mut Screen, log_y: &mut usize) ->
(u64, bool) {
    // Same pattern as probe_virtio_nic_with_debug
    // Look for vendor 0x1AF4, device 0x1001 (legacy) or 0x1042 (modern)
    const VIRTIO_BLK_LEGACY: u16 = 0x1001;
    const VIRTIO_BLK_MODERN: u16 = 0x1042;

    // Scan PCI for block device...
}

// In prepare_handoff():
fn prepare_handoff(..., blk_mmio_base: u64, blk_info: BlockInfo) ->
BootHandoff {
    BootHandoff {
        // ... existing fields ...
        blk_mmio_base,
        blk_type: BLK_TYPE_VIRTIO,
        blk_sector_size: blk_info.sector_size,
        blk_total_sectors: blk_info.total_sectors,
        // ...
    }
}
```

VirtIO-blk Also Needs PCI Modern Support!

Critical: VirtIO-blk has the **same problem** as VirtIO-net:

- Current ASM uses MMIO transport offsets
- QEMU's `virtio-blk-pci` is a PCI device, not MMIO

The **same PCI capability parsing** infrastructure will work for both:

- VirtIO-net (device 0x1041)
- VirtIO-blk (device 0x1042)

They share the **same Common Config structure**, just with different device-specific configs.

DMA Memory Layout (Extended for Block)

```
DMA Region (2MB total):
┌──────────────────────────────────────────────────────────────────────┐
│  Offset      │   Size    │  Content                                  │
├──────────────┼───────────┼───────────────────────────────────────────┤
│  0x00000     │   256KB   │  Network RX/TX queues + buffers           │
│  0x40000     │   256KB   │  Block I/O queue + buffers                │
│  0x80000     │   512KB   │  HTTP receive buffer (for streaming)      │
│  0x100000    │   512KB   │  Block write staging buffer               │
│  0x180000    │   512KB   │  Reserved / scratch                       │
└──────────────────────────────────────────────────────────────────────┘
```

Summary

The fix requires implementing **VirtIO PCI Modern transport** alongside the existing MMIO transport. The key differences are:

1. **Discovery**: Parse PCI capabilities instead of reading fixed MMIO offsets
2. **Configuration**: Use Common Config structure (different offsets than MMIO)
3. **Notification**: Calculate notify address from capability info + multiplier
4. **Device Config**: MAC address at separate location specified by capability
5. **Block Device**: Wire up existing VirtIO-blk driver, same PCI Modern fix needed

Total Implementation Scope

Component	New Lines	Modified Lines
PCI Capability Parsing (ASM)	~300	-
VirtIO PCI Modern Init (ASM)	~400	-
VirtIO PCI Queue Setup (ASM)	~300	-
VirtIO PCI Notify (ASM)	~100	-
Rust PCI Capability Discovery	~200	-
Rust Driver Integration	-	~300
Block Device Wire-up	-	~200

Component	New Lines	Modified Lines
Bootloader Block Probe	-	~150
Total	**~1300**	**~650**

Total estimated effort: **~2000 lines of changes**, **4-5 days**.

This enables the complete flow:

```
VirtIO-net (PCI Modern) → HTTP Download → VirtIO-blk (PCI Modern) → Disk
```