MorpheusX Network Stack Design Audit

Version: 1.1
Date: 2026-01-09
Status: Authoritative Design Specification (Expert Reviewed)
Scope: Post-ExitBootServices Bare-Metal Network Stack

Part 1: Critical Review — Implicit Assumptions & Hidden Dependencies

This section identifies every implicit assumption, underspecified invariant, and hidden dependency in the original design document (NETWORK_STACK_REDESIGN.md). Each is categorized by severity and domain.

1.1 Memory Model Assumptions

ASSUMPTION M1: Identity Mapping Completeness

- **Claim**: "Physical == Virtual addresses"
- **Problem**: The document assumes identity mapping exists and is complete for all DMA regions, but never specifies:
  - WHO establishes the identity mapping (UEFI? Custom page tables?)
  - WHEN identity mapping is validated
  - WHAT happens if UEFI's memory map contains non-identity-mapped regions
  - WHETHER the identity mapping survives ExitBootServices
- **Severity**: CRITICAL
- **Reality**: UEFI does NOT guarantee identity mapping. The firmware may use arbitrary virtual addresses. Post-EBS, if we don't set up our own page tables, we inherit whatever CR3 UEFI left behind—which may or may not be identity-mapped.

ASSUMPTION M2: DMA Region Visibility

- **Claim**: "DMA region must be allocated BEFORE ExitBootServices, then managed by ASM"
- **Problem**: The document assumes allocated memory is immediately DMA-visible without:
  - Cache coherency considerations
  - IOMMU/VT-d implications
  - Memory type (UC vs WB vs WC) specification
- **Severity**: CRITICAL
- **Reality**: On real hardware with IOMMU enabled, DMA addresses may differ from CPU physical addresses. On systems with enabled VT-d, the DMA region must be mapped in the IOMMU or marked as pass-through.

ASSUMPTION M3: Cache Coherency Model

- **Claim**: Uses `mfence` for memory barriers
- **Problem**: Document conflates three distinct concerns:
  1. CPU memory ordering (handled by `mfence`)

    2. Cache coherency with DMA (requires cache flushes or uncached mapping)
    3. Compiler reordering (requires volatile + compiler barriers)
- **Severity**: CRITICAL
- **Reality**: `mfence` does NOT ensure DMA coherency. On x86 with write-back caching, the CPU may have written to cache but not to memory. The device sees stale data. VirtIO on QEMU hides this because QEMU's emulation is cache-coherent by construction.

ASSUMPTION M4: Static Pool Sufficiency

- **Claim**: "2MB DMA region" is sufficient
- **Problem**: No analysis of actual memory requirements:
    - 256 RX buffers × 2KB = 512KB
    - 256 TX buffers × 2KB = 512KB
    - Virtqueue structures: ~1KB per queue
    - But: smoltcp also allocates internally (reassembly buffers, socket buffers)
- **Severity**: HIGH
- **Hidden Dependency**: smoltcp's internal memory usage is not accounted for

---

1.2 Timing & TSC Assumptions

ASSUMPTION T1: TSC Frequency Stability

- **Claim**: "assuming ~2.5GHz CPU"
- **Problem**: TSC frequency:
    - Varies between CPU models (1GHz to 5GHz+)
    - May change with power states on older CPUs (pre-invariant TSC)
    - Is NOT guaranteed to be tied to wall-clock time
- **Severity**: HIGH
- **Reality**: Must detect TSC frequency at boot, not assume it

ASSUMPTION T2: TSC Monotonicity

- **Claim**: Uses `wrapping_sub` for timeout checks
- **Problem**: Document assumes TSC never goes backward
- **Reality**: On SMP systems, TSC can be unsynchronized between cores. On single-core (our case), this is safe, but should be explicitly stated as a constraint.

ASSUMPTION T3: TSC Availability

- **Claim**: Uses `rdtsc` unconditionally
- **Problem**: Does not verify:
    - CPUID feature bit for TSC
    - Invariant TSC feature (CPUID.80000007H:EDX[8])
- **Severity**: MEDIUM
- **Reality**: All x86-64 CPUs have TSC, but invariant TSC should be verified

ASSUMPTION T4: Calibration Reference

- **Claim**: "Use UEFI Stall() as reference" for calibration
- **Problem**: UEFI Stall() accuracy varies by firmware implementation. Some firmwares have ±10% error.
- **Severity**: MEDIUM
- **Better Approach**: Use PIT or HPET for calibration, or accept imprecision

---

1.3 VirtIO Assumptions

ASSUMPTION V1: VirtIO Version

- **Claim**: Uses "modern" VirtIO with MMIO or PCI transport
- **Problem**: Does not specify:
  - Minimum VirtIO spec version (1.0? 1.1? 1.2?)
  - Required vs optional features
  - Legacy vs modern device handling
- **Severity**: HIGH
- **Reality**: QEMU supports both legacy (0.9.5) and modern (1.0+) devices. The driver must negotiate features explicitly.

ASSUMPTION V2: Feature Negotiation

- **Claim**: Implicit that features are negotiated
- **Problem**: Document never specifies which features are:
  - REQUIRED (must reject device if absent)
  - OPTIONAL (use if available)
  - FORBIDDEN (must not negotiate)
- **Severity**: HIGH
- **Critical Missing Features**:
  - `VIRTIO_NET_F_MAC` - needed to get MAC address
  - `VIRTIO_NET_F_STATUS` - link status
  - `VIRTIO_F_VERSION_1` - modern device indicator
  - `VIRTIO_NET_F_MRG_RXBUF` - affects header size!

ASSUMPTION V3: Queue Size

- **Claim**: "16 entries" for virtqueues
- **Problem**:
  - VirtIO spec says queue size is negotiated with device
  - Device may support different max sizes
  - 16 is very small for high-throughput scenarios
- **Severity**: MEDIUM

ASSUMPTION V4: Notification Suppression

- **Claim**: Checks `VIRTQ_USED_F_NO_NOTIFY` before notifying
- **Problem**: Document doesn't handle the symmetric case:
  - `VIRTQ_AVAIL_F_NO_INTERRUPT` (driver → device: don't interrupt me)
  - Since we're polling, we MUST set this flag to suppress interrupts

- **Severity**: MEDIUM

ASSUMPTION V5: VirtIO Header Size

- **Claim**: "VirtIO header (12B)"
- **Problem**: Header size depends on negotiated features:
  - Without `VIRTIO_NET_F_MRG_RXBUF`: 10 bytes (legacy) or 12 bytes (modern)
  - With `VIRTIO_NET_F_MRG_RXBUF`: 12 bytes
  - With `VIRTIO_NET_F_HASH_REPORT`: larger
- **Severity**: HIGH
- **Reality**: Must dynamically determine header size based on features

---

1.4 smoltcp Assumptions

ASSUMPTION S1: smoltcp Thread Safety

- **Claim**: Implicit single-threaded use
- **Problem**: Document doesn't verify that smoltcp's `Interface` and sockets are safe to use without synchronization
- **Reality**: smoltcp is NOT thread-safe by design. This is fine for single-core, but must be documented.

ASSUMPTION S2: smoltcp Memory Model

- **Claim**: smoltcp just works with our device
- **Problem**: smoltcp expects the `Device` trait to:
  - Return tokens that are valid until consumed
  - Not have packet data modified between `receive()` returning and token consumption
- **Hidden Dependency**: RX buffer ownership during token lifetime

ASSUMPTION S3: smoltcp Timestamp Resolution

- **Claim**: Converts TSC to milliseconds for smoltcp
- **Problem**: smoltcp's `Instant` is in milliseconds, but:
  - Timeout precision matters for TCP retransmission (typically 200ms RTO min)
  - Clock wrap-around handling in smoltcp
- **Severity**: LOW (likely fine, but unverified)

ASSUMPTION S4: smoltcp Buffer Sizes

- **Claim**: Uses default socket buffers
- **Problem**: Document doesn't specify:
  - TCP socket RX/TX buffer sizes
  - Maximum number of sockets
  - Reassembly buffer limits
- **Hidden Dependency**: These affect both performance and memory usage

ASSUMPTION S5: smoltcp Poll Semantics

- **Claim**: "smoltcp's poll() must never block"
- **Problem**: This is stated but not verified against smoltcp source
- **Reality**: smoltcp's `poll()` is indeed non-blocking, but it may:
  - Call device `transmit()` multiple times in one poll
  - Hold references to device during poll
  - Expect `transmit()` to make progress or return `None`

1.5 UEFI & ExitBootServices Assumptions

ASSUMPTION U1: ExitBootServices Semantics

- **Claim**: After EBS, only Runtime Services available
- **Problem**: Document understates restrictions:
  - Memory map may have changed
  - Firmware may have reclaimed memory
  - Some Runtime Services require virtual address setup
- **Severity**: HIGH

ASSUMPTION U2: Memory Map Stability

- **Claim**: Pre-allocated DMA region survives EBS
- **Problem**: Must allocate with correct memory type:
  - `EfiLoaderData` - may be reclaimed by OS later
  - `EfiRuntimeServicesData` - survives but for firmware use
  - `EfiBootServicesData` - reclaimed at EBS!
- **Severity**: CRITICAL
- **Correct Type**: `EfiLoaderData` or custom type marked in memory map

ASSUMPTION U3: PCI State After EBS

- **Claim**: "PCI config access may fail after EBS"
- **Problem**: This is vague. Reality:
  - ECAM MMIO access works (if region is mapped)
  - Legacy I/O port access works (I/O ports are unaffected by EBS)
  - But: UEFI may have configured PCI bridges; that config is static
- **Severity**: MEDIUM

ASSUMPTION U4: Device Ownership

- **Claim**: "NIC driver initialization MUST occur AFTER ExitBootServices()"
- **Problem**: Partially wrong. What matters is:
  - Not conflicting with UEFI's device access
  - UEFI SNP/NII protocols must be stopped first
  - Device reset is safe before or after EBS
- **Severity**: MEDIUM
- **Reality**: Can probe PCI and note device locations before EBS; full init after EBS

1.6 ASM Layer Assumptions

ASSUMPTION A1: ABI Correctness

- **Claim**: Uses "Microsoft x64 ABI"
- **Problem**: UEFI uses MS ABI, but:
  - After EBS, we control the ABI
  - If mixing Rust (System V by default on Linux targets) with ASM (MS ABI), calling conventions mismatch
- **Severity**: HIGH
- **Reality**: UEFI target uses MS ABI; must be consistent throughout

ASSUMPTION A2: Register Preservation

- **Claim**: ASM functions preserve non-volatile registers
- **Problem**: Document shows ASM snippets that:
  - Push/pop RBX, R12, etc.
  - But doesn't show complete preservation in all functions
- **Severity**: MEDIUM

ASSUMPTION A3: Atomic Operations

- **Claim**: ASM handles virtqueue atomicity
- **Problem**: Virtqueue available index update shown as:

  ```
  inc word [rdi + 2]
  ```

  This is NOT atomic. If interrupted (even by NMI), corruption possible.
- **Severity**: LOW (single-core, interrupts disabled, but still a latent bug)

1.7 Hardware Assumptions

ASSUMPTION H1: NIC Presence

- **Claim**: Assumes NIC exists and is VirtIO
- **Problem**: No fallback if:
  - No NIC found
  - NIC is non-VirtIO
  - NIC fails to initialize
- **Severity**: HIGH (for production use)

ASSUMPTION H2: Link State

- **Claim**: Assumes link is up after init
- **Problem**: Link auto-negotiation takes time (up to 5 seconds for GbE)
- **Reality**: VirtIO in QEMU has instant link; real NICs do not

ASSUMPTION H3: MTU

- **Claim**: "MTU: 1514"
- **Problem**: 1514 is Ethernet frame size (1500 payload + 14 header). But:
  - VirtIO can negotiate larger MTU
  - Jumbo frames exist (9000+ bytes)
  - Document mixes MTU (IP level) with frame size (L2)
- **Severity**: LOW (conservative choice is fine)

1.8 Concurrency & Reentrancy Assumptions

ASSUMPTION C1: No Interrupts

- **Claim**: "Polling-only, no interrupt handlers installed"
- **Problem**: Even without installing handlers:
  - NMI can fire (hardware error, watchdog)
  - SMM can preempt (invisible to OS)
  - These can corrupt in-progress operations
- **Severity**: LOW (unlikely but possible)

ASSUMPTION C2: No Reentrancy

- **Claim**: Single main loop, no nesting
- **Problem**: Document doesn't analyze:
  - What if smoltcp's poll() calls device methods that call back into smoltcp?
  - What if error handler triggers network access?
- **Severity**: MEDIUM

1.9 Error Handling Assumptions

ASSUMPTION E1: Failure Semantics

- **Claim**: Table mentions "retry vs fatal" but doesn't define
- **Problem**: For each failure mode, need to specify:
  - Is it recoverable?
  - How many retries?
  - What state to reset to?
- **Severity**: HIGH

ASSUMPTION E2: Partial Failure

- **Claim**: State machines handle errors
- **Problem**: No analysis of partial failure:
  - TX submitted but completion lost
  - RX buffer leaked
  - Socket half-open
- **Severity**: HIGH

1.10 Summary of Critical Assumptions Requiring Resolution

ID	Domain	Assumption	Severity	Resolution Required
M1	Memory	Identity mapping exists	CRITICAL	Define who creates it
M2	Memory	DMA region visible	CRITICAL	Specify IOMMU handling
M3	Memory	mfence = cache coherent	CRITICAL	Specify memory type
V2	VirtIO	Features negotiated	HIGH	List required features
V5	VirtIO	Header size is 12B	HIGH	Make dynamic
U2	UEFI	DMA survives EBS	CRITICAL	Specify memory type
A1	ASM	ABI is MS x64	HIGH	Verify target config
S5	smoltcp	poll() never blocks	HIGH	Verify in source
E1	Error	Failure semantics defined	HIGH	Document each failure

Part 2: Invariant Extraction & Formalization

This section formalizes invariants as **contracts** — preconditions, postconditions, and invariants that must hold for correct operation. These are written in a semi-formal notation suitable for implementation verification.

2.1 DMA Buffer Ownership Contracts

```
INVARIANT DMA-OWN-1: Buffer Exclusive Ownership
  ∀ buffer B in DMA_REGION:
    EXACTLY ONE of the following holds at any time:
      - B is FREE (in allocator free list)
      - B is DRIVER-OWNED (Rust code may read/write)
      - B is DEVICE-OWNED (submitted to hardware)

  TRANSITION: FREE → DRIVER-OWNED
    PRECONDITION: allocator.free_count > 0
    ACTION: allocator.alloc() returns B
    POSTCONDITION: B not in free list, driver may access B

  TRANSITION: DRIVER-OWNED → DEVICE-OWNED
    PRECONDITION: B is fully initialized with valid data
    PRECONDITION: cache flushed OR memory is uncached
    ACTION: submit B to virtqueue available ring
    POSTCONDITION: driver MUST NOT read/write B until returned

  TRANSITION: DEVICE-OWNED → DRIVER-OWNED
    PRECONDITION: B appears in virtqueue used ring
    ACTION: driver retrieves B from used ring
    POSTCONDITION: driver may read B (RX) or reclaim B (TX)
```

```
  TRANSITION: DRIVER-OWNED → FREE
    PRECONDITION: B is not referenced anywhere
    ACTION: allocator.free(B)
    POSTCONDITION: B in free list
```

```
INVARIANT DMA-OWN-2: RX Buffer Lifecycle
  ∀ RX buffer B:
    LIFECYCLE:
      1. ALLOC: B = alloc_rx_buffer()
      2. SUBMIT: submit_rx_buffer(B) → B becomes DEVICE-OWNED
      3. RECEIVE: device writes packet → marks B in used ring
      4. RETRIEVE: poll_used_ring() returns B → B becomes DRIVER-OWNED
      5. CONSUME: smoltcp RxToken consumes B
      6. RECYCLE: return B to step 2 (resubmit) or free

    CONSTRAINT: Between steps 2-4, driver MUST NOT access B.data[]
    CONSTRAINT: B must remain valid memory for entire lifecycle
```

```
INVARIANT DMA-OWN-3: TX Buffer Lifecycle
  ∀ TX buffer B:
    LIFECYCLE:
      1. ALLOC: B = alloc_tx_buffer()
      2. FILL: driver writes packet to B.data[]
      3. SUBMIT: submit_tx_buffer(B) → B becomes DEVICE-OWNED
      4. COMPLETE: device transmits → marks B in used ring
      5. RECLAIM: poll_used_ring() returns B → B becomes DRIVER-OWNED
      6. FREE: free_tx_buffer(B) OR reuse for next packet

    CONSTRAINT: Between steps 3-5, driver MUST NOT access B.data[]
    CONSTRAINT: TX completion may be delayed arbitrarily by device
```

2.2 Virtqueue State Invariants

```
INVARIANT VQ-1: Descriptor Table Consistency
  ∀ virtqueue VQ:
    LET desc_count = VQ.num_entries

    INVARIANT: desc_count is power of 2
    INVARIANT: desc_count >= 2 AND desc_count <= 32768
    INVARIANT: ∀ i in 0..desc_count:
      VQ.desc[i].addr is page-aligned OR within valid DMA buffer
      VQ.desc[i].len <= MAX_BUFFER_SIZE
      VQ.desc[i].flags in {0, NEXT, WRITE, NEXT|WRITE}
      VQ.desc[i].next < desc_count (if NEXT flag set)
```

```
INVARIANT VQ-2: Available Ring Consistency
  ∀ virtqueue VQ:
    LET avail = VQ.available_ring
    LET mask = VQ.num_entries - 1

    INVARIANT: avail.idx is monotonically increasing (mod 2^16)
    INVARIANT: ∀ i in [last_avail_idx, avail.idx):
      avail.ring[i & mask] < VQ.num_entries
      avail.ring[i & mask] points to valid descriptor chain

    ORDERING CONSTRAINT:
      descriptor write → memory_barrier → avail.idx write → memory_barrier
→ notify
```

```
INVARIANT VQ-3: Used Ring Consistency
  ∀ virtqueue VQ:
    LET used = VQ.used_ring
    LET mask = VQ.num_entries - 1

    INVARIANT: used.idx is modified ONLY by device
    INVARIANT: driver reads used.idx with acquire semantics
    INVARIANT: ∀ i in [last_used_idx, used.idx):
      used.ring[i & mask].id < VQ.num_entries
      used.ring[i & mask].len <= original buffer length
```

```
INVARIANT VQ-4: Free Descriptor Tracking
  ∀ virtqueue VQ:
    LET free_list = internal free descriptor chain
    LET in_flight = descriptors between available and used

    INVARIANT: |free_list| + |in_flight| == VQ.num_entries
    INVARIANT: free_list ∩ in_flight == ∅
    INVARIANT: each descriptor is in exactly one of {free_list, in_flight}
```

2.3 Memory Ordering Contracts

```
CONTRACT MEM-ORD-1: Virtqueue Submission Ordering
  OPERATION: Submit buffer to available ring

  SEQUENCE (must execute in this order):
    1. WRITE: descriptor.addr, descriptor.len, descriptor.flags
    2. BARRIER: sfence (store fence) - ensures (1) visible before (3)
    3. WRITE: available.ring[idx] = descriptor_index
    4. BARRIER: sfence - ensures (3) visible before (5)
    5. WRITE: available.idx += 1
```

```
    6. BARRIER: mfence - ensures (5) visible before (7)
    7. CONDITIONAL: if notification needed, WRITE to notify register

  RATIONALE:
    - Device may read descriptors as soon as avail.idx changes
    - Without barriers, device may see stale descriptor data
    - mfence before notify ensures all writes complete
```

```
CONTRACT MEM-ORD-2: Virtqueue Completion Polling
  OPERATION: Poll used ring for completions

  SEQUENCE:
    1. READ: used.idx (with acquire semantics)
    2. COMPARE: if used.idx == last_used_idx, return None
    3. BARRIER: lfence (load fence) - ensures (1) before (4)
    4. READ: used.ring[last_used_idx & mask]
    5. READ: buffer contents (for RX)
    6. UPDATE: last_used_idx += 1

  RATIONALE:
    - Acquire semantics on idx read synchronizes with device's release
    - lfence ensures we don't speculatively read ring entry
```

```
CONTRACT MEM-ORD-3: MMIO Access Ordering
  ∀ MMIO register access:
    CONSTRAINT: All MMIO accesses are naturally ordered (x86 strong model)
    CONSTRAINT: Use volatile read/write to prevent compiler reordering
    CONSTRAINT: mfence between MMIO and DMA memory if both involved

  NOTE: x86 provides strong ordering for MMIO (UC memory type).
        DMA buffers may be WB (write-back), requiring explicit barriers.
```

2.4 Cache Coherency Contracts

```
CONTRACT CACHE-1: DMA Region Memory Type
  PRECONDITION: DMA region allocated before ExitBootServices

  REQUIREMENT: DMA region MUST be one of:
    a) Uncached (UC) - simplest, no cache management needed
    b) Write-Combining (WC) - for TX buffers, better performance
    c) Write-Back (WB) with explicit cache management

  IF memory type is WB:
    BEFORE device access (TX submit):
      clflush/clflushopt each cache line in buffer
      sfence after flush sequence
```

```
    AFTER device access (RX complete):
      clflush/clflushopt OR just read (speculative execution safe on x86)

  PREFERRED: Use UC or WC for DMA region to avoid cache management
```

```
CONTRACT CACHE-2: Virtqueue Structure Memory Type
  REQUIREMENT: Virtqueue rings and descriptors SHOULD be uncached

  RATIONALE:
    - Device reads descriptors immediately after avail.idx update
    - WB memory may have stale data in cache
    - UC ensures device always sees latest writes

  ALTERNATIVE: Use clflush before each avail.idx update
               (complex, error-prone, not recommended)
```

2.5 Time Source Contracts

```
CONTRACT TIME-1: TSC Properties
  PRECONDITION: CPU supports invariant TSC (CPUID.80000007H:EDX[8] == 1)

  GUARANTEES:
    - TSC increments at constant rate regardless of power state
    - TSC is monotonically increasing (no backward jumps)
    - TSC is per-core but we run single-core (no sync issues)

  NON-GUARANTEES:
    - TSC frequency is NOT known a priori
    - TSC does NOT survive system reset
    - TSC MAY wrap (at 2^64 cycles, ~200 years at 3GHz)
```

```
CONTRACT TIME-2: TSC Calibration
  OPERATION: Determine TSC frequency at boot

  METHOD:
    1. PRE-EBS: Use UEFI Stall(1000000) as 1-second reference
       OR: Program PIT channel 2, measure TSC delta
    2. Record: TSC_FREQ = delta_tsc / reference_seconds
    3. Store: TSC_FREQ in boot handoff structure

  ACCURACY: ±5% acceptable for network timeouts
  PRECISION: At least 1µs resolution required (any modern CPU satisfies)
```

```
CONTRACT TIME-3: Timeout Calculation
  ∀ timeout operation:
    LET deadline_tsc = start_tsc + (timeout_ms * TSC_FREQ / 1000)
    LET now_tsc = rdtsc()

    TIMEOUT_EXPIRED iff:
      (now_tsc - start_tsc) >= (timeout_ms * TSC_FREQ / 1000)

    NOTE: Use wrapping subtraction to handle TSC wrap
    NOTE: Timeout precision is ±1 TSC read latency (~20-100 cycles)
```

2.6 VirtIO Feature Negotiation Contract

```
CONTRACT VIRTIO-FEAT-1: Required Features
  PRECONDITION: Device offers features via feature bits

  DRIVER MUST negotiate (refuse device if absent):
    - VIRTIO_F_VERSION_1 (bit 32): Modern device indicator
    - VIRTIO_NET_F_MAC (bit 5): MAC address available

  DRIVER SHOULD negotiate (use if available):
    - VIRTIO_NET_F_STATUS (bit 16): Link status available
    - VIRTIO_NET_F_MRG_RXBUF (bit 15): Mergeable RX buffers
    - VIRTIO_F_RING_EVENT_IDX (bit 29): Efficient notifications

  DRIVER MUST NOT negotiate:
    - VIRTIO_NET_F_GUEST_TSO4/6: We don't support TSO
    - VIRTIO_NET_F_GUEST_UFO: We don't support UFO
    - VIRTIO_NET_F_CTRL_VQ: Control virtqueue (complexity)

  POSTCONDITION:
    Header size = 10 bytes (legacy) or 12 bytes (modern with VERSION_1)
    If MRG_RXBUF negotiated: header includes num_buffers field
```

```
CONTRACT VIRTIO-FEAT-2: Device Status Protocol
  SEQUENCE (must execute in order):
    1. Reset device: write 0 to status register
    2. Set ACKNOWLEDGE (bit 0): driver recognizes device
    3. Set DRIVER (bit 1): driver can drive device
    4. Read features, select features to negotiate
    5. Write negotiated features
    6. Set FEATURES_OK (bit 3): features accepted
    7. Read status, verify FEATURES_OK still set (device agreed)
    8. Perform device-specific setup (queues, etc.)
    9. Set DRIVER_OK (bit 2): driver ready
```

```
  ON ERROR at any step:
    Set FAILED (bit 7) and abort
```

2.7 State Machine Invariants

```
INVARIANT SM-1: State Machine Progress
  ∀ state machine SM:
    PROPERTY: Each call to SM.step() either:
      a) Transitions to a new state, OR
      b) Returns without state change (waiting for external event)

    LIVENESS: Given sufficient external progress (packets arrive, time
passes),
              SM eventually reaches a terminal state (Done or Failed)

    BOUNDED EXECUTION: SM.step() completes in O(1) time
                       (no unbounded loops within step)
```

```
INVARIANT SM-2: State Machine Cleanup
  ∀ state machine SM:
    ON transition to Failed state:
      - All owned resources released (sockets closed, buffers freed)
      - No dangling references to state machine data

    ON transition to Done state:
      - Result data valid and owned by caller
      - Intermediate resources released
```

```
INVARIANT SM-3: TCP Socket State Consistency
  ∀ TCP socket S managed by state machine:
    IF SM.state == Connecting:
      S.state ∈ {SynSent, Established} (smoltcp TCP states)
    IF SM.state == Established:
      S.state == Established
    IF SM.state == Closing:
      S.state ∈ {FinWait1, FinWait2, Closing, TimeWait, LastAck}

    INVARIANT: SM state transitions only on observed smoltcp state changes
```

2.8 Main Loop Invariants

```

```

```
INVARIANT LOOP-1: Bounded Iteration Time
  ∀ iteration of main loop:
    LET t_start = rdtsc() at loop start
    LET t_end = rdtsc() at loop end

    GUARANTEE: (t_end - t_start) < MAX_LOOP_CYCLES
    WHERE: MAX_LOOP_CYCLES = 5 * TSC_FREQ / 1000  (5ms)

    MECHANISM:
      - RX_BUDGET bounds RX processing
      - TX_BUDGET bounds TX processing
      - smoltcp.poll() is O(sockets + packets)
      - App.step() is O(1) per state machine step
```

```
INVARIANT LOOP-2: No Nested Blocking
  ∀ function F called within main loop:
    F MUST NOT:
      - Loop waiting for external condition
      - Call any function that loops waiting
      - Sleep or delay (except bounded spin for HW timing)

    F MAY:
      - Loop bounded by input size (e.g., process N packets)
      - Loop bounded by constant (e.g., retry 3 times)
      - Return early if condition not met
```

```
INVARIANT LOOP-3: Progress Guarantee
  GIVEN:
    - Network is functional (packets can be sent/received)
    - Time advances (TSC increments)

  THEN:
    - DHCP completes or times out within DHCP_TIMEOUT
    - TCP connections complete or time out within TCP_TIMEOUT
    - HTTP requests complete or time out within HTTP_TIMEOUT

  MECHANISM: Each state machine checks timeout per step()
```

2.9 Error Recovery Invariants

```
INVARIANT ERR-1: Recoverable vs Fatal Errors
  CLASSIFICATION:
    FATAL (halt system):
      - DMA region allocation failed
      - No NIC found
```

```
      - VirtIO feature negotiation failed
      - ExitBootServices failed

    RECOVERABLE (retry or continue):
      - TX queue full → backpressure, retry later
      - RX buffer exhausted → drop packet, refill
      - DHCP timeout → retry with backoff
      - TCP connection refused → report error, continue
      - DNS timeout → try fallback servers
```

```
INVARIANT ERR-2: Resource Leak Prevention
  ON any error path:
    ENSURE: All allocated resources are freed

    CHECKLIST:
      - [ ] TX buffers returned to pool
      - [ ] RX buffers returned to pool
      - [ ] Sockets removed from interface
      - [ ] State machine data dropped
      - [ ] Virtqueue descriptors reclaimed
```

End of Part 2. Continue to Part 3 for Authoritative Source Cross-References.

Part 3: Cross-Reference with Authoritative Sources

This section cites authoritative specifications for each critical invariant, explaining how each source constrains the design.

3.1 VirtIO Specification References

Source: *Virtual I/O Device (VIRTIO) Version 1.2*, OASIS Standard, 2022
URL: https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html

Section 2.1: Device Status Field

> "The driver MUST update device status, setting bits to indicate completed steps of the driver initialization sequence."

Constraint on Design:

- Status register writes MUST follow exact sequence (Reset → Acknowledge → Driver → Features_OK → Driver_OK)
- Design document shows correct sequence in CONTRACT VIRTIO-FEAT-2

Section 2.4.2: The Virtqueue Available Ring

> "The driver SHOULD NOT publish an available descriptor index value more than MAX_QUEUE_SIZE ahead of the used index."

Constraint on Design:

- Cannot submit more buffers than queue size
- Current design uses 16 entries; must track in-flight count

Section 2.7.4: Device Operation - Notification

> "The driver MUST perform a suitable memory barrier before the driver notification, to ensure the device sees the most up-to-date copy."

Constraint on Design:

- Validates CONTRACT MEM-ORD-1 requirement for mfence before notify
- ASM layer MUST include memory barrier in notification path

Section 2.7.7: Legacy Interface

> "Transitional devices and drivers MAY support the legacy interface."

Constraint on Design:

- Legacy devices use different header format (10 bytes vs 12 bytes)
- Design MUST check VIRTIO_F_VERSION_1 to determine mode

Section 5.1.4: Device Configuration Layout (Network Device)

> "mac: 6 bytes. Only present if VIRTIO_NET_F_MAC is set."

Constraint on Design:

- MUST negotiate VIRTIO_NET_F_MAC before reading MAC address
- Current implementation assumes MAC is always available (WRONG)

Section 5.1.6: Device Operation (Network Device)

> "The virtio_net_hdr structure... For VIRTIO_NET_F_VERSION_1 conformant devices, the size is 12 bytes."

Constraint on Design:

- Header size depends on negotiated features
- Design MUST use 12 bytes for modern devices, 10 for legacy
- With VIRTIO_NET_F_MRG_RXBUF, header includes `num_buffers` field

3.2 UEFI Specification References

Source: *UEFI Specification Version 2.10*, UEFI Forum, 2022
URL: https://uefi.org/specifications

Section 7.4: EFI_BOOT_SERVICES.ExitBootServices()

> "On success, the UEFI OS Loader owns all of the available memory in the system. In addition, the UEFI OS Loader can treat all memory in the map as available if it conforms to the corresponding memory type."

Constraint on Design:

- After EBS, memory marked `EfiConventionalMemory` is available
- Memory marked `EfiBootServicesData` is reclaimed — MUST NOT use for DMA
- Memory marked `EfiLoaderData` remains valid — USE THIS for DMA region

Section 7.4 (continued)

> "This call is the point in time where memory map changes are completed."

Constraint on Design:

- Memory map retrieved before EBS may differ from final map
- MUST get fresh memory map immediately before EBS call

Section 2.3.2: Handoff State

> "The firmware boot manager, or operating system boot loader... must preserve the content of the memory map."

Constraint on Design:

- We are the "OS loader" in UEFI terms
- We control memory after EBS; firmware doesn't touch it

Section 8.2: Runtime Services

> "Certain runtime services require that SetVirtualAddressMap() be called..."

Constraint on Design:

- If we never call SetVirtualAddressMap(), runtime services use physical addresses
- This is fine for our use case (identity mapping)

3.3 Intel Architecture References

Source: *Intel 64 and IA-32 Architectures Software Developer's Manual*
Volume 3A: System Programming Guide
URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Section 8.2.2: Memory Ordering in P6 and More Recent Processor Families

> "Reads may be reordered with older writes to different locations... but not with older writes to the same location."

Constraint on Design:

- x86 provides strong ordering for most operations
- BUT: writes may be buffered (store buffer)
- `sfence` flushes store buffer — needed before avail.idx update

Section 8.2.5: Strengthening or Weakening the Memory-Ordering Model

> "MFENCE — Serializing: MFENCE serializes all store and load operations."

Constraint on Design:

- `mfence` is the correct barrier before device notification
- Cheaper alternatives (`sfence`) may be insufficient for full serialization

Section 11.3: Methods of Caching Available

> "UC (Uncacheable)... System memory locations are not cached."

Constraint on Design:

- UC memory provides strongest ordering guarantees
- DMA regions SHOULD be mapped UC for simplicity
- Alternative: Use WB with explicit cache management (complex)

Section 11.5.3: CLFLUSH Instruction

> "Flushes the cache line that contains the linear address specified..."

Constraint on Design:

- If DMA region is WB, must CLFLUSH before device reads (TX)
- Must CLFLUSH or invalidate after device writes (RX)

3.4 AMD Architecture References

Source: *AMD64 Architecture Programmer's Manual*
Volume 2: System Programming
URL: https://www.amd.com/en/support/tech-docs

Section 7.4.2: Memory Barrier Interactions with Memory Types

> "For cacheable memory types (WB, WT, WP), reads can pass buffered writes..."

Constraint on Design:

- Same as Intel — WB memory requires barriers or cache management
- AMD is consistent with Intel on x86-64 memory model

3.5 PCI/PCIe Specification References

Source: *PCI Express Base Specification Revision 5.0*, PCI-SIG, 2019

Section 2.4: Transaction Ordering

> "Memory Write Requests are posted... Completions are returned for Memory Read Requests."

Constraint on Design:

- PCIe writes are posted (no completion)
- Reads block until data returns
- Device notification (write) may complete before device processes it

Section 6.5: Error Handling

> "Advisory Non-Fatal errors are uncorrected errors that have been corrected by PCI Express protocols."

Constraint on Design:

- PCIe errors typically don't crash system
- But corrupted DMA data is possible (rare)
- Design doesn't handle PCIe errors (acceptable for research)

3.6 smoltcp Source Code Analysis

Source: `smoltcp` crate, version 0.11.x
Repository: https://github.com/smoltcp-rs/smoltcp

File: `src/iface/interface.rs`, `poll()` method

```
pub fn poll<D>(&mut self, timestamp: Instant, device: &mut D, sockets: &mut
SocketSet<'_>)
where D: Device + ?Sized
```

Verified Behavior:

- `poll()` does NOT block internally
- Calls `device.receive()` zero or more times
- Calls `device.transmit()` zero or more times
- Returns immediately when no work to do

Constraint on Design:

- Confirms ASSUMPTION S5 is correct
- Device `receive()` returns `None` when no packets — smoltcp handles this
- Device `transmit()` returns `None` when queue full — smoltcp handles this

File: `src/phy/mod.rs`, `Device` trait

```
pub trait Device {
    type RxToken<'a>: RxToken where Self: 'a;
    type TxToken<'a>: TxToken where Self: 'a;

    fn receive(&mut self, timestamp: Instant) -> Option<(Self::RxToken<'_>,
Self::TxToken<'_>)>;
    fn transmit(&mut self, timestamp: Instant) ->
Option<Self::TxToken<'_>>;
}
```

Verified Behavior:

- `receive()` returns paired RX+TX tokens (for responses)
- `transmit()` returns just TX token
- Tokens are borrowing from device (lifetime tied to `&mut self`)

Constraint on Design:

- RX buffer MUST remain valid while RxToken exists
- Cannot submit RX buffer back to device until token consumed

File: `src/socket/tcp.rs`, TCP socket state machine

Verified Behavior:

- smoltcp handles TCP state machine internally
- Exposes socket state via `tcp::Socket::state()`
- Retransmission handled by smoltcp (timer-based)

Constraint on Design:

- Design's TCP state tracking (INVARIANT SM-3) aligns with smoltcp
- Must provide accurate timestamps for retransmission timers

3.7 virtio-drivers Crate Analysis

Source: `virtio-drivers` crate, used by current implementation
Repository: https://github.com/rcore-os/virtio-drivers

File: `src/device/net.rs`, `VirtIONetRaw`

```
pub fn transmit_begin(&mut self, buf: &[u8]) -> Result<u16> {
    // Submits buffer, returns token
}

pub fn poll_transmit(&mut self) -> Option<u16> {
    // Non-blocking check for TX completion
}
```

Verified Behavior:

- `transmit_begin` is non-blocking (just submits to queue)
- `poll_transmit` is non-blocking (just checks used ring)
- Current implementation wraps these in a BLOCKING LOOP (the bug)

Constraint on Design:

- virtio-drivers provides correct non-blocking primitives
- Bug is in wrapper code, not the library
- Redesign should use begin/poll pattern directly

File: `src/hal.rs`, HAL trait requirements

```
pub trait Hal {
    fn dma_alloc(pages: usize, direction: BufferDirection) -> (PhysAddr,
NonNull<u8>);
    fn dma_dealloc(paddr: PhysAddr, vaddr: NonNull<u8>, pages: usize);
    fn phys_to_virt(paddr: PhysAddr) -> NonNull<u8>;
    // ...
}
```

Constraint on Design:

- HAL assumes identity mapping for simplicity
- Must provide DMA allocator that returns both phys and virt addresses
- Current `dma-pool` crate satisfies this interface

3.8 Summary: Specification Compliance Matrix

Requirement	VirtIO Spec	UEFI Spec	Intel SDM	Design Status
Feature negotiation sequence	§2.1	-	-	⚠ Needs verification
Memory barrier before notify	§2.7.4	-	§8.2.5	✓ In ASM
DMA memory type	-	§7.4	§11.3	✗ Not specified
Header size determination	§5.1.6	-	-	✗ Hardcoded 12
Memory map for DMA	-	§7.4	-	⚠ Type unclear
Cache coherency	-	-	§11.5.3	✗ Not addressed
TSC calibration	-	§7.5	§17.17	⚠ Assumed freq

Legend:

- ✓ Correctly implemented
- ⚠ Partially correct, needs review

- ✗ Missing or incorrect

End of Part 3. Continue to Part 4 for Design Gaps Analysis.

Part 4: Design Gaps & Incorrect Assumptions

This section directly calls out assumptions that are **invalid**, behaviors that **work only in QEMU but not on real hardware**, and guarantees that **smoltcp does not actually provide**.

4.1 Assumptions Invalid on Real Hardware

GAP 4.1.1: Cache Coherency is NOT Free

Original Assumption: Using `mfence` ensures device sees correct data
Reality on Real Hardware:

On real x86 hardware with write-back (WB) memory:

1. CPU writes to DMA buffer → data goes to cache, not memory
2. `mfence` ensures all stores complete → they complete TO CACHE
3. Device reads via DMA → reads from MEMORY, sees stale data

QEMU Behavior: QEMU emulates cache-coherent DMA. Device reads see cache contents.
Real Hardware Behavior: Device reads bypass cache, see only what's in RAM.

Required Fix:

```
OPTION A: Map DMA region as Uncached (UC)
  - Page table entry: PCD=1, PWT=1
  - No cache management needed
  - Performance impact: ~10x slower writes

OPTION B: Map DMA region as Write-Back (WB) with explicit flushes
  - Before TX: CLFLUSH each cache line in buffer, then SFENCE
  - After RX: CLFLUSH or rely on speculative invalidation
  - Complex, error-prone

RECOMMENDATION: Use UC for DMA region. Simplicity > performance.
```

GAP 4.1.2: IOMMU May Remap DMA Addresses

Original Assumption: "Physical == Virtual addresses" for DMA
Reality on Real Hardware:

On systems with Intel VT-d or AMD-Vi enabled:

1. DMA addresses from device perspective are IOVA (I/O Virtual Addresses)

2. IOMMU translates IOVA → physical address
3. If IOMMU is active, device-visible address ≠ CPU physical address

QEMU Behavior: VirtIO devices use guest physical addresses directly (no IOMMU by default)
Real Hardware Behavior: Depends on firmware/OS IOMMU configuration

Required Fix:

```
STRATEGY: Disable IOMMU or configure passthrough

UEFI Phase:
  1. Check if IOMMU enabled (ACPI DMAR table presence)
  2. If IOMMU active:
     a) Add identity mapping for DMA region in IOMMU tables, OR
     b) Disable IOMMU for our device, OR
     c) Use IOVA returned by IOMMU allocation

SIMPLEST: Many firmwares leave IOMMU disabled during boot.
          Document this as requirement: "IOMMU must be disabled or
passthrough"
```

GAP 4.1.3: PCIe Device Reset Timing

Original Assumption: Device reset is instantaneous
Reality on Real Hardware:

VirtIO spec §2.1.1:

> "After writing 0 to the device status register, the driver MUST wait for the reset to complete before doing further initialization."

Reset completion time varies:

- QEMU VirtIO: Immediate
- Real Intel NIC: 10-100ms
- Real Realtek NIC: 5-50ms

Required Fix:

```
AFTER device reset:
  1. Write 0 to status register
  2. Poll status register until reads as 0 (or timeout)
  3. Wait additional 100ms for conservative reset completion
  4. THEN proceed with initialization
```

GAP 4.1.4: Link Negotiation Takes Real Time

Original Assumption: Link is up after driver init
Reality on Real Hardware:

Physical layer negotiation (for real NICs):

- 10/100 Mbps auto-negotiate: 2-3 seconds
- 1 Gbps auto-negotiate: 3-5 seconds
- 10 Gbps: 1-2 seconds

VirtIO has no physical layer — link is "always up" in QEMU.

Required Fix:

```
FOR real NIC drivers (Intel e1000, Realtek):
  1. Initialize PHY
  2. Wait for link up (poll PHY status register)
  3. Timeout after 10 seconds if no link
  4. Report link speed to application layer

FOR VirtIO:
  - Check VIRTIO_NET_S_LINK_UP in device status (if F_STATUS negotiated)
  - Treat as always-up if F_STATUS not available
```

4.2 QEMU-Specific Behaviors That Hide Bugs

GAP 4.2.1: QEMU VirtIO is Synchronous

Hidden Behavior: In QEMU, VirtIO TX completion happens in the same VCPU timeslice
Consequence: Code that assumes immediate TX completion works in QEMU but fails on real HW

The original blocking loop:

```
loop {
    if let Some(t) = self.inner.poll_transmit() {
        if t == token { return Ok(()); }
    }
    tsc_delay_us(10);
}
```

Why it works in QEMU: QEMU's VirtIO backend processes TX in virtqueue_push(), which happens during the same execution context. The poll_transmit() sees completion almost immediately.

Why it would fail on real hardware: Real hardware processes TX asynchronously. The loop might spin for milliseconds or longer, violating the no-blocking constraint.

The fix is already in the design document — replace with state machine.

GAP 4.2.2: QEMU Memory Access is Atomic

Hidden Behavior: QEMU emulates memory access atomically per instruction
Consequence: Non-atomic updates to virtqueue structures appear to work

Example from ASM:

```
inc word [rdi + 2]  ; Increment avail.idx
```

This is NOT atomic on x86. It's a read-modify-write sequence:

1. Read [rdi+2] into temp
2. Increment temp
3. Write temp to [rdi+2]

If NMI or SMI fires between steps 1 and 3, corruption possible.

Why it works in QEMU: QEMU's TCG (and even KVM) doesn't inject interrupts mid-instruction translation block
Why it matters: Latent bug that could manifest under stress

Required Fix:

```
lock inc word [rdi + 2]  ; Atomic increment (if needed)
; OR, since we disable interrupts and run single-core:
; Document: "Safe only because interrupts disabled and single-core"
```

GAP 4.2.3: QEMU Timing is Unrealistic

Hidden Behavior: QEMU's TSC advances in bursts, not continuously
Consequence: Timeout logic may behave differently

In QEMU (especially TCG mode):

- TSC jumps when control returns from QEMU to guest
- A "10µs delay" might take 0 TSC ticks or millions

Required Fix:

```
Document as explicit limitation:
  "Timing tests in QEMU are NOT representative of real hardware timing.
   Hardware testing required for timing-sensitive paths."
```

4.3 smoltcp Guarantees That Don't Exist

GAP 4.3.1: smoltcp Does NOT Guarantee Bounded poll() Time

Assumed: "smoltcp poll() completes in bounded time"
Reality: smoltcp's poll() time depends on:

- Number of sockets (iterates all sockets)
- Packet rate (processes all pending packets)
- Retransmission state (may generate retransmits)

In pathological cases (many sockets, burst of packets), poll() may take longer than expected.

Mitigation:

```
1. Limit number of sockets (we use ~3 max)
2. Limit packets processed per poll (via RX_BUDGET before poll)
3. Accept that poll() time is O(sockets + packets), bound by our limits
```

GAP 4.3.2: smoltcp Socket Buffers Are Fixed at Creation

Assumed: Flexible buffer management
Reality: Socket buffer sizes set at socket creation, cannot grow

```
let tcp_rx_buffer = tcp::SocketBuffer::new(vec![0; 65535]);
let tcp_tx_buffer = tcp::SocketBuffer::new(vec![0; 65535]);
let socket = tcp::Socket::new(tcp_rx_buffer, tcp_tx_buffer);
```

Consequence: Must pre-allocate maximum expected buffer size
Memory Impact: Each TCP socket: 128KB minimum (64KB RX + 64KB TX)

Required Fix:

```
Document memory requirements explicitly:
  - Per TCP socket: 128KB
  - Maximum sockets: N
  - Total smoltcp memory: N * 128KB + overhead

For 3 sockets: ~400KB for smoltcp alone (separate from DMA region)
```

GAP 4.3.3: smoltcp DHCP is State-Machine Based But Opaque

Assumed: Can observe DHCP progress
Reality: smoltcp's DHCP client exposes limited state

```
// What smoltcp provides:
socket.poll();  // Returns Dhcpv4Event enum
```

```
pub enum Dhcpv4Event {
    Deconfigured,
    Configured(Config),
}
```

Cannot distinguish: Discovering vs Requesting vs Rebinding

Mitigation:

```
Accept coarse-grained DHCP status:
  - HasIP (Configured)
  - NoIP (Deconfigured or in-progress)

Use timeout to bound total DHCP time, not per-state timeouts.
```

4.4 Hidden Reliance on Interrupts or Background Progress

GAP 4.4.1: TCP Retransmission Requires Timely poll()

Hidden Dependency: TCP retransmission timers only advance during poll()

If main loop stalls (e.g., processing large download), TCP connections may:

- Time out spuriously (retransmit timer fires repeatedly)
- Miss retransmission window
- Have increased latency

Required Guarantee:

```
INVARIANT: Main loop MUST call smoltcp.poll() at least once per 100ms
           (or RTO_MIN / 2, whichever is smaller)

Current design: ~1ms loop iteration satisfies this
```

GAP 4.4.2: ARP Table Expires Without Refresh

Hidden Dependency: ARP entries expire; smoltcp expects periodic poll() to manage

smoltcp ARP cache default expiry: 60 seconds

If we don't poll() for >60s, ARP entries expire, requiring re-resolution.

Mitigation: Already handled by continuous poll() loop.

4.5 Architectural Contradictions in Original Document

GAP 4.5.1: Section 4 Duplicated Content

The original document has Section 4.4 through 4.9, then restarts at 4.4 again with different content. This suggests copy-paste error or incomplete editing.

Affected Sections:

- "4.4 ASM DMA Allocator" appears once
- "4.4 VirtIO Notification Sequence" appears later
- Section numbering is inconsistent

Resolution: Renumber sections in corrected document.

GAP 4.5.2: Contradictory Buffer Counts

Document states:

- "256 × 2KB buffers" in DMA layout (§4.3)
- "16 entries" for virtqueue size (§4.6)

These are inconsistent. With 16 virtqueue entries, we can only have 16 buffers in-flight.

Resolution: Clarify:

- Virtqueue size: 16-256 (configurable)
- Buffer pool: should be ≥ 2× virtqueue size for double-buffering
- Recommended: 32-entry virtqueues, 64 buffers per direction

GAP 4.5.3: ASM Interface is Incomplete

Section 4.7 lists "Complete ASM Interface" but:

- `asm_phy_write` declared but not implemented
- `asm_mac_reset`, `asm_mac_init`, etc. declared but not implemented
- No error return conventions specified

Resolution: Either implement all declared functions or explicitly mark as "future work."

4.6 Missing Security Considerations

GAP 4.6.1: No Input Validation on Received Packets

Document states: "smoltcp validation" handles malformed packets

Reality: smoltcp validates protocol headers but:

- Cannot validate application-layer data
- Cannot prevent resource exhaustion attacks
- Cannot prevent slowloris-style attacks

For research project: Acceptable to defer **For production**: Would need rate limiting, connection limits

GAP 4.6.2: DHCP is Unauthenticated

Any device on the network can respond to DHCP. Attacker can:

- Provide malicious DNS server
- Provide malicious gateway
- Redirect traffic

For research project: Document as known limitation **For production**: Would need DHCP snooping or static config option

---

4.7 Summary: Critical Fixes Required

Gap ID	Description	Severity	Fix Effort
4.1.1	Cache coherency not addressed	CRITICAL	Medium — change memory mapping
4.1.2	IOMMU not considered	HIGH	Low — document requirement
4.1.3	Reset timing not handled	MEDIUM	Low — add delay
4.1.4	Link negotiation time	MEDIUM	Medium — add PHY polling
4.2.1	QEMU hides async TX	HIGH	Already fixed in design
4.3.2	smoltcp memory not budgeted	HIGH	Medium — calculate and document
4.5.2	Buffer count inconsistent	MEDIUM	Low — clarify numbers
4.5.3	ASM interface incomplete	MEDIUM	Medium — implement or remove

---

End of Part 4. Continue to Part 5 for Corrected Design Specification.

Part 5: Corrected, Locked-Down Design Specification

This section provides the authoritative design that a contributor can implement without guessing. All ambiguities from the original document are resolved.

5.1 System Model & Constraints

5.1.1 Execution Environment

Property	Value	Rationale
CPU Cores	1 (BSP only)	Simplicity; no SMP synchronization
Interrupt State	Disabled	No ISRs; pure polling model
Paging	Enabled, identity-mapped	UEFI leaves paging on; we maintain identity map
Memory Model	x86-64 Total Store Order	Strong ordering; explicit barriers for DMA only

Property	Value	Rationale
Floating Point	Disabled (no FPU/SSE state)	no_std, soft-float; avoids save/restore
Stack	Pre-allocated, 64KB minimum	Allocated before EBS, survives EBS

5.1.2 Memory Map Requirements

DMA Region:

- **Size**: 2MB (0x200000 bytes)
- **Alignment**: Page-aligned (4KB boundary)
- **UEFI Memory Type**: `EfiLoaderData` (type 2)
- **Caching**: Uncached (UC) — mapped with PCD=1, PWT=1 in page tables
- **Location**: Below 4GB preferred (for 32-bit DMA compatibility)

Allocation Procedure (before EBS):

```
// Step 1: Allocate with UEFI
let dma_region = boot_services.allocate_pages(
    AllocateType::MaxAddress(0xFFFF_FFFF),  // Below 4GB
    MemoryType::LOADER_DATA,                // Survives EBS
    512,                                    // 512 pages = 2MB
)?;

// Step 2: Record in handoff structure
handoff.dma_base = dma_region as u64;
handoff.dma_size = 2 * 1024 * 1024;

// Step 3: After EBS, remap as Uncached in page tables
// (Requires custom page table setup)
```

5.1.3 Timing Subsystem

TSC Requirements:

- Invariant TSC required (CPUID.80000007H:EDX bit 8)
- Calibrated at boot using UEFI Stall() or PIT
- Frequency stored in handoff structure

Calibration Procedure:

```
fn calibrate_tsc(boot_services: &BootServices) -> u64 {
    let start = rdtsc();
    boot_services.stall(1_000_000);  // 1 second
    let end = rdtsc();
    end - start  // TSC ticks per second
}
```

Timeout Precision: ±5% (acceptable for network protocols)

5.2 Boot Sequence Specification

5.2.1 Phase 1: UEFI Boot Services Active

Must Complete Before ExitBootServices:

Step	Action	Output
1	Load and parse configuration	Config struct
2	Scan PCI for VirtIO NIC	DeviceFunction, MMIO base
3	Record MAC address (via PCI config)	6-byte MAC (optional)
4	Allocate DMA region (2MB, EfiLoaderData)	Physical address
5	Allocate stack (64KB, EfiLoaderData)	Stack top pointer
6	Calibrate TSC	Ticks per second
7	Get final memory map	Memory map buffer
8	Populate handoff structure	BootHandoff struct
9	Set up identity-mapped page tables	CR3 value

Must NOT Do Before ExitBootServices:

- Initialize VirtIO device (reset, feature negotiation)
- Submit DMA buffers to device
- Enable device operation

5.2.2 ExitBootServices Call

```
// Critical: Get fresh memory map immediately before EBS
let (map, map_key) = boot_services.memory_map()?;

// Store map for post-EBS use (optional)
handoff.memory_map_ptr = map.as_ptr() as u64;
handoff.memory_map_size = map.len() as u64;

// The call itself
let status = boot_services.exit_boot_services(image_handle, map_key);

if status != Status::SUCCESS {
    // Memory map changed; retry
    let (new_map, new_key) = boot_services.memory_map()?;
    boot_services.exit_boot_services(image_handle, new_key)?;
}

// POINT OF NO RETURN — Boot Services are gone
```

5.2.3 Phase 2: Bare Metal Execution

Immediate Post-EBS Setup:

Step	Action	Notes
1	Switch to pre-allocated stack	RSP = handoff.stack_top
2	Remap DMA region as UC	Modify page tables, flush TLB
3	Initialize serial output (optional)	For debugging
4	Initialize VirtIO NIC	Full device setup
5	Create smoltcp Interface	With device adapter
6	Enter main poll loop	Never returns

5.3 VirtIO Network Driver Specification

5.3.1 Feature Negotiation

Required Features (MUST be present; reject device otherwise):

- `VIRTIO_F_VERSION_1` (bit 32): Modern device protocol

Desired Features (negotiate if available):

- `VIRTIO_NET_F_MAC` (bit 5): MAC address in config space
- `VIRTIO_NET_F_STATUS` (bit 16): Link status reporting
- `VIRTIO_F_RING_EVENT_IDX` (bit 29): Efficient notification suppression

Forbidden Features (MUST NOT negotiate):

- `VIRTIO_NET_F_GUEST_TSO4` (bit 7): No TSO support
- `VIRTIO_NET_F_GUEST_TSO6` (bit 8): No TSO support
- `VIRTIO_NET_F_GUEST_UFO` (bit 10): No UFO support
- `VIRTIO_NET_F_MRG_RXBUF` (bit 15): Simplifies header handling
- `VIRTIO_NET_F_CTRL_VQ` (bit 17): No control virtqueue

Header Size Determination:

```
let header_size = if negotiated_features.contains(VIRTIO_F_VERSION_1) {
    12  // Modern device
} else {
    10  // Legacy device
};
```

5.3.2 Virtqueue Configuration

Queue	Index	Direction	Size	Buffer Size
RX	0	Device → Driver	32 entries	2048 bytes
TX	1	Driver → Device	32 entries	2048 bytes

Memory Layout (within DMA region):

```
Offset      Size    Content
0x0000      0x200   RX descriptor table (32 × 16 bytes)
0x0200      0x044   RX available ring (2 + 2 + 32×2 + 2 bytes, padded)
0x0244      0x1BC   (padding to 0x400)
0x0400      0x104   RX used ring (2 + 2 + 32×8 bytes, padded)
0x0504      0x2FC   (padding to 0x800)
0x0800      0x200   TX descriptor table (32 × 16 bytes)
0x0A00      0x044   TX available ring
0x0A44      0x1BC   (padding)
0x0C00      0x104   TX used ring
0x0D04      0x2FC   (padding to 0x1000)
0x1000      0x10000 RX buffers (32 × 2048 bytes)
0x11000     0x10000 TX buffers (32 × 2048 bytes)
0x21000     ...     (remaining ~1.87MB reserved for future use)
```

5.3.3 Device Initialization Sequence

```
fn virtio_net_init(mmio_base: u64, dma_region: &mut DmaRegion) ->
Result<VirtioNet> {
    // 1. Reset device
    write_status(mmio_base, 0);
    wait_for_reset(mmio_base, 100_000)?;  // 100ms timeout

    // 2. Set ACKNOWLEDGE
    write_status(mmio_base, STATUS_ACKNOWLEDGE);

    // 3. Set DRIVER
    write_status(mmio_base, STATUS_ACKNOWLEDGE | STATUS_DRIVER);

    // 4. Read and negotiate features
    let device_features = read_features(mmio_base);
    let our_features = negotiate_features(device_features)?;
    write_features(mmio_base, our_features);

    // 5. Set FEATURES_OK
    write_status(mmio_base, STATUS_ACKNOWLEDGE | STATUS_DRIVER |
STATUS_FEATURES_OK);

    // 6. Verify FEATURES_OK
    let status = read_status(mmio_base);
    if status & STATUS_FEATURES_OK == 0 {
        write_status(mmio_base, STATUS_FAILED);
```

```
        return Err(VirtioError::FeatureNegotiationFailed);
    }

    // 7. Setup virtqueues
    setup_virtqueue(mmio_base, 0, dma_region.rx_queue())?;  // RX
    setup_virtqueue(mmio_base, 1, dma_region.tx_queue())?;  // TX

    // 8. Pre-fill RX queue with buffers
    for i in 0..32 {
        submit_rx_buffer(dma_region, i)?;
    }

    // 9. Set DRIVER_OK
    write_status(mmio_base, STATUS_ACKNOWLEDGE | STATUS_DRIVER |
                            STATUS_FEATURES_OK | STATUS_DRIVER_OK);

    // 10. Get MAC address
    let mac = if our_features & VIRTIO_NET_F_MAC != 0 {
        read_mac_from_config(mmio_base)
    } else {
        generate_random_mac()  // Or fail
    };

    Ok(VirtioNet { mmio_base, dma_region, mac })
}
```

5.4 Memory Barrier Specification

5.4.1 TX Path Barriers

```
TX SUBMIT SEQUENCE:
   ┌─────────────────────────────────────────────────────────────┐
   │ 1. Write buffer data                                        │
   │    - CPU stores to TX buffer in DMA region                  │
   │    - If WB memory: CLFLUSH buffer, then SFENCE              │
   │    - If UC memory: no action needed (writes go to memory)   │
   ├─────────────────────────────────────────────────────────────┤
   │ 2. Write descriptor                                         │
   │    - desc[idx].addr = buffer_phys_addr                      │
   │    - desc[idx].len = packet_length + header_size            │
   │    - desc[idx].flags = 0                                    │
   │    *** SFENCE *** (ensure descriptor visible)               │
   ├─────────────────────────────────────────────────────────────┤
   │ 3. Update available ring                                    │
   │    - avail.ring[avail.idx & mask] = desc_idx                │
   │    *** SFENCE *** (ensure ring entry visible)               │
   │    - avail.idx += 1                                         │
   │    *** MFENCE *** (full barrier before notify)              │
   ├─────────────────────────────────────────────────────────────┤
   │ 4. Notify device (if needed)                                │
```

```
    │    - Check used.flags for NO_NOTIFY                                  │
    │    - If notification needed: write to notify register                │
    └──────────────────────────────────────────────────────────────────────┘
```

5.4.2 RX Path Barriers

```
RX POLL SEQUENCE:
    ┌──────────────────────────────────────────────────────────────────────┐
    │ 1. Read used.idx                                                     │
    │    - Read with acquire semantics (or plain read + LFENCE)            │
    │    - Compare with last_used_idx                                      │
    │    - If equal: no new completions, return                            │
    ├──────────────────────────────────────────────────────────────────────┤
    │ 2. Read used ring entry                                              │
    │    *** LFENCE *** (if not using acquire)                             │
    │    - entry = used.ring[last_used_idx & mask]                         │
    │    - desc_idx = entry.id                                             │
    │    - bytes_written = entry.len                                       │
    ├──────────────────────────────────────────────────────────────────────┤
    │ 3. Read buffer data                                                  │
    │    - If WB memory: LFENCE before reading buffer                      │
    │      (ensures we see device's writes)                                │
    │    - If UC memory: no barrier needed                                 │
    │    - Copy packet data from RX buffer                                 │
    ├──────────────────────────────────────────────────────────────────────┤
    │ 4. Update tracking                                                   │
    │    - last_used_idx += 1                                              │
    │    - Return buffer to free pool OR resubmit to available             │
    └──────────────────────────────────────────────────────────────────────┘
```

5.5 Main Loop Specification

```rust
/// Main loop — the ONLY entry point for all network activity
fn main_loop(
    net: &mut VirtioNet,
    iface: &mut smoltcp::Interface,
    sockets: &mut smoltcp::SocketSet,
    app: &mut ApplicationState,
    tsc_freq: u64,
) -> ! {
    const RX_BUDGET: usize = 32;   // Max RX per iteration
    const TX_BUDGET: usize = 32;   // Max TX per iteration

    loop {
        let now_tsc = rdtsc();
        let now_ms = tsc_to_ms(now_tsc, tsc_freq);
        let timestamp = smoltcp::time::Instant::from_millis(now_ms as i64);
```

```
        // Phase 1: Collect RX completions
        for _ in 0..RX_BUDGET {
            match net.poll_rx() {
                Some(packet) => net.rx_queue.push(packet),
                None => break,
            }
        }

        // Phase 2: Run smoltcp poll
        // smoltcp will call device.receive() and device.transmit()
        iface.poll(timestamp, &mut DeviceAdapter::new(net), sockets);

        // Phase 3: Drain TX queue (fire-and-forget)
        for _ in 0..TX_BUDGET {
            match net.tx_queue.pop() {
                Some(packet) => {
                    if net.submit_tx(&packet).is_err() {
                        net.tx_queue.push_front(packet);  // Requeue
                        break;
                    }
                }
                None => break,
            }
        }

        // Phase 4: Collect TX completions (free buffers)
        while let Some(_token) = net.poll_tx_complete() {
            // Buffer automatically returned to pool
        }

        // Phase 5: Application state machine step
        match app.step(iface, sockets, now_tsc) {
            StepResult::Continue => {}
            StepResult::Done(result) => {
                handle_completion(result);
                // May exit loop or start new operation
            }
            StepResult::Fatal(error) => {
                handle_fatal_error(error);
                // Does not return
            }
        }
    }
}
```

5.6 State Machine Specifications

5.6.1 DHCP State Machine

```rust
pub enum DhcpState {
    /// Initial state
    Init,

    /// Waiting for IP configuration from smoltcp DHCP
    Discovering {
        start_tsc: u64,
    },

    /// IP address obtained
    Bound {
        config: DhcpConfig,
        obtained_tsc: u64,
    },

    /// DHCP failed (timeout or error)
    Failed {
        error: DhcpError,
    },
}

impl DhcpState {
    pub fn step(&mut self, iface: &Interface, now_tsc: u64, timeout_tsc:
u64) -> StepResult {
        match self {
            DhcpState::Init => {
                *self = DhcpState::Discovering { start_tsc: now_tsc };
                StepResult::Continue
            }

            DhcpState::Discovering { start_tsc } => {
                // Check timeout
                if now_tsc.wrapping_sub(*start_tsc) > timeout_tsc {
                    *self = DhcpState::Failed { error: DhcpError::Timeout
};
                    return StepResult::Done(Err(DhcpError::Timeout));
                }

                // Check if we got an IP
                if let Some(ip) = iface.ipv4_addr() {
                    // smoltcp's DHCP socket has configured the interface
                    *self = DhcpState::Bound {
                        config: DhcpConfig::from_interface(iface),
                        obtained_tsc: now_tsc,
                    };
                    return StepResult::Done(Ok(()));
                }

                StepResult::Continue
            }

            DhcpState::Bound { .. } => StepResult::Done(Ok(())),
            DhcpState::Failed { error } =>
```

```
    StepResult::Done(Err(error.clone())),
            }
        }
    }
```

5.6.2 HTTP Download State Machine

```
pub enum HttpDownloadState {
    /// Initial state
    Init { url: Url },

    /// Resolving hostname
    Resolving {
        url: Url,
        start_tsc: u64,
    },

    /// Connecting to server
    Connecting {
        url: Url,
        ip: Ipv4Addr,
        socket: SocketHandle,
        start_tsc: u64,
    },

    /// Sending HTTP request
    Sending {
        socket: SocketHandle,
        request: Vec<u8>,
        sent: usize,
        start_tsc: u64,
    },

    /// Receiving response headers
    ReceivingHeaders {
        socket: SocketHandle,
        buffer: Vec<u8>,
        start_tsc: u64,
    },

    /// Receiving response body
    ReceivingBody {
        socket: SocketHandle,
        headers: Headers,
        received: usize,
        expected: Option<usize>,
        sink: Box<dyn FnMut(&[u8]) -> Result<()>>,
        start_tsc: u64,
    },

    /// Download complete
```

```
    Done { total_bytes: usize },

    /// Download failed
    Failed { error: HttpError },
}
```

5.7 Non-Goals & Explicit Limitations

This section documents what the runtime explicitly does NOT provide.

5.7.1 Functional Non-Goals

Non-Goal	Rationale
TLS/HTTPS support	Complexity; requires crypto library integration
IPv6 support	Complexity; IPv4 sufficient for bootstrap
UDP applications	Only TCP needed for HTTP
Multi-connection parallelism	Single connection sufficient for download
Persistent storage	Boot only; no filesystem writes
User input during network ops	Non-interactive bootstrap

5.7.2 Performance Non-Goals

Non-Goal	Rationale
>1 Gbps throughput	Not needed for ISO download
<1ms latency	Network latency dominates
Zero-copy packet processing	Simplicity over performance
Jumbo frame support	Standard MTU sufficient

5.7.3 Security Non-Goals

Non-Goal	Rationale
Protection from malicious DHCP	Trusted network assumed
Protection from MitM attacks	No TLS; trusted network
DoS resistance	Single-use bootstrap
Secure random number generation	Not needed

5.7.4 Explicit Guarantees

The runtime DOES guarantee:

Guarantee	Bound
Main loop iteration time	< 5ms
DHCP completion or timeout	< 30 seconds
TCP connection or timeout	< 30 seconds
HTTP response start or timeout	< 60 seconds
Memory usage (total)	< 4MB
No blocking calls	All functions return in bounded time

5.8 Handoff Structure Definition

```rust
/// Data passed from UEFI phase to bare-metal phase.
/// Must be placed in EfiLoaderData memory that survives EBS.
#[repr(C)]
pub struct BootHandoff {
    /// Magic number for validation: 0x4D4F5250_48455553 ("MORPHEUS")
    pub magic: u64,

    /// Structure version (currently 1)
    pub version: u32,

    /// Size of this structure in bytes
    pub size: u32,

    // === NIC Information ===

    /// MMIO base address for VirtIO NIC (or 0 if none found)
    pub nic_mmio_base: u64,

    /// PCI location of NIC
    pub nic_pci_bus: u8,
    pub nic_pci_device: u8,
    pub nic_pci_function: u8,

    /// NIC type: 0=None, 1=VirtIO, 2=Intel, 3=Realtek
    pub nic_type: u8,

    /// MAC address (valid if nic_type != 0)
    pub mac_address: [u8; 6],

    /// Padding for alignment
    pub _pad1: [u8; 2],

    // === DMA Region ===

    /// DMA region base (page-aligned, UC-mapped)
```

```
    pub dma_base: u64,

    /// DMA region size in bytes (minimum 2MB)
    pub dma_size: u64,

    // === Timing ===

    /// TSC frequency in Hz (ticks per second)
    pub tsc_freq: u64,

    // === Stack ===

    /// Stack top address for post-EBS execution
    pub stack_top: u64,

    /// Stack size in bytes
    pub stack_size: u64,

    // === Memory Map (optional) ===

    /// Pointer to UEFI memory map (may be stale after EBS)
    pub memory_map_ptr: u64,

    /// Memory map size in bytes
    pub memory_map_size: u64,

    /// Memory map descriptor size
    pub memory_map_desc_size: u64,

    // === Debug Output (optional) ===

    /// Framebuffer base for debug output (or 0)
    pub framebuffer_base: u64,

    /// Framebuffer size
    pub framebuffer_size: u64,

    /// Framebuffer width in pixels
    pub framebuffer_width: u32,

    /// Framebuffer height in pixels
    pub framebuffer_height: u32,

    /// Framebuffer stride in bytes
    pub framebuffer_stride: u32,

    /// Padding
    pub _pad2: u32,
}

impl BootHandoff {
    pub const MAGIC: u64 = 0x4D4F5250_48455553;
    pub const VERSION: u32 = 1;
```

```
    pub fn validate(&self) -> bool {
        self.magic == Self::MAGIC &&
        self.version == Self::VERSION &&
        self.size as usize == core::mem::size_of::<Self>() &&
        self.dma_base != 0 &&
        self.dma_size >= 2 * 1024 * 1024 &&
        self.tsc_freq > 0 &&
        self.stack_top != 0
    }
}
```

End of Part 5. Continue to Part 6 for Documentation Upgrade Plan.

Part 6: Systematic Documentation Upgrade Plan

This section provides a step-by-step plan to evolve documentation alongside implementation, ensuring the design document remains aligned with code.

6.1 Phase 1: Foundation (Week 1-2)

Objective: Establish verifiable baseline before any code changes.

6.1.1 Document Current State

Task	Output	Owner
Audit existing `virtio.rs` blocking patterns	List of line numbers with blocking loops	-
Audit existing `native.rs` blocking patterns	List of line numbers with blocking loops	-
Document current buffer ownership model	Ownership diagram	-
Record current memory usage	Measured values	-

6.1.2 Create Test Infrastructure

Task	Output
QEMU test script for network stack	`testing/run-network-test.sh`
Timeout verification test	Test that detects blocking loops
Memory bounds test	Test that measures DMA region usage

6.1.3 Baseline Measurements

```
REQUIRED MEASUREMENTS (record in docs/BASELINE.md):
```

```
1. DHCP completion time (average, P99)
2. TCP connection time (average, P99)
3. HTTP download throughput (MB/s)
4. Main loop iteration time (average, max)
5. Memory usage breakdown (DMA, smoltcp, stack)
6. TSC frequency on test machine
```

6.2 Phase 2: Memory Model Hardening (Week 3-4)

Objective: Correct cache coherency and memory mapping.

6.2.1 Tasks

Task	Priority	Verification
Implement UC mapping for DMA region	CRITICAL	Test on real hardware
Add page table manipulation code	CRITICAL	Verify with CR3 dump
Document memory type in handoff struct	HIGH	Code review
Add runtime check for identity mapping	MEDIUM	Assert at boot

6.2.2 Documentation Updates

- ☐ Add "Memory Model" section to implementation docs
- ☐ Document page table structure used post-EBS
- ☐ Add comments to DMA allocation code explaining memory type
- ☐ Create diagram of physical memory layout

6.2.3 Validation Experiments

```
EXPERIMENT 2.1: Verify UC Mapping
  1. Allocate DMA buffer
  2. Write pattern to buffer
  3. Read back via MMIO (if possible) or device DMA
  4. Verify pattern matches without CLFLUSH

EXPERIMENT 2.2: Measure UC Performance Impact
  1. Benchmark memcpy to UC region vs WB region
  2. Document throughput difference
  3. Verify acceptable for our use case
```

6.3 Phase 3: VirtIO Driver Correctness (Week 5-6)

Objective: Ensure VirtIO driver meets specification requirements.

6.3.1 Code Inspection Checklist

- [ ] Feature negotiation matches CONTRACT VIRTIO-FEAT-1
- [ ] Device status sequence matches CONTRACT VIRTIO-FEAT-2
- [ ] Header size is dynamically determined, not hardcoded
- [ ] Memory barriers present per MEM-ORD-1 and MEM-ORD-2
- [ ] Virtqueue indices wrap correctly (mod 2^16)
- [ ] No blocking loops in TX/RX paths

6.3.2 Required Code Changes

File	Change	Invariant Enforced
`virtio.rs`	Remove blocking loop in `transmit()`	LOOP-2
`virtio.rs`	Add dynamic header size	V5 resolution
`virtio.rs`	Add feature negotiation validation	VQ-1
New file	`virtio_barriers.rs` or inline ASM	MEM-ORD-1

6.3.3 Documentation Updates

- [ ] Add "VirtIO Compliance" section documenting which spec sections are implemented
- [ ] Add comments citing VirtIO spec for each protocol step
- [ ] Document which features are negotiated and why

6.4 Phase 4: State Machine Refactor (Week 7-8)

Objective: Replace all blocking patterns with state machines.

6.4.1 State Machines to Implement

State Machine	Current Code	New Location
DHCP	Blocking in `wait_for_network()`	`state/dhcp.rs`
DNS	Blocking in `try_dns_query()`	`state/dns.rs`
TCP Connect	Blocking in `wait_for_connection()`	`state/tcp.rs`
HTTP	Multiple blocking calls	`state/http.rs`

6.4.2 Refactor Pattern

For each blocking function:

```
// BEFORE (blocking)
fn wait_for_x(&mut self) -> Result<()> {
    let start = self.now();
```

```
    while !condition {
        self.poll();
        if timeout { return Err(Timeout); }
        delay(1000);  // BLOCKING
    }
    Ok(())
}

// AFTER (state machine)
enum XState {
    Waiting { start_tsc: u64 },
    Done,
    Failed(Error),
}

fn step_x(&mut self, state: &mut XState, now_tsc: u64) -> StepResult {
    match state {
        XState::Waiting { start_tsc } => {
            if now_tsc - *start_tsc > TIMEOUT {
                *state = XState::Failed(Error::Timeout);
                return StepResult::Done;
            }
            if condition {
                *state = XState::Done;
                return StepResult::Done;
            }
            StepResult::Continue  // Will be called again
        }
        _ => StepResult::Done,
    }
}
```

6.4.3 Documentation Updates

- ☐ Add state diagram for each state machine
- ☐ Document timeout values and rationale
- ☐ Add examples of state machine usage

6.5 Phase 5: smoltcp Integration Audit (Week 9-10)

Objective: Verify smoltcp usage matches its actual API contract.

6.5.1 Source Code Inspection Tasks

Task	smoltcp File	Question to Answer
Verify poll() is non-blocking	`iface/interface.rs`	Any internal loops?
Check token lifetime	`phy/mod.rs`	How long is RxToken valid?
Verify buffer requirements	`socket/tcp.rs`	Minimum buffer sizes?

Task	smoltcp File	Question to Answer
Check timestamp requirements	`time.rs`	Resolution requirements?

6.5.2 Document Findings

Create `docs/SMOLTCP_INTEGRATION.md`:

```
# smoltcp Integration Notes

## Version: 0.11.x

## Verified Behaviors
- poll() does not block internally [verified: interface.rs:XXX]
- RxToken must be consumed before next receive() call
- TxToken consume() may be called with any buffer size ≤ MTU

## Memory Requirements
- Per TCP socket: 128KB (64KB RX + 64KB TX default)
- Interface overhead: ~10KB
- Total for 3 sockets: ~400KB

## Timestamp Requirements
- Millisecond resolution sufficient
- Monotonic required
- Wrap-around: handled at i64 max (292 million years)
```

6.6 Phase 6: ASM Layer Specification (Week 11-12)

Objective: Fully specify and implement ASM interface.

6.6.1 ASM Functions to Implement

Function	Status	Priority
`asm_read_tsc`	Exists in `pci_io.S`	Done
`asm_virtq_submit`	Not implemented	HIGH
`asm_virtq_poll`	Not implemented	HIGH
`asm_notify_device`	Not implemented	HIGH
`asm_memory_barrier`	Not implemented	MEDIUM

6.6.2 ASM Documentation Requirements

Each ASM function must have:

```

```

```
;
================================================================================
==
; Function: asm_virtq_submit
; Purpose:  Submit buffer to virtqueue with correct memory barriers
;
; Inputs:
;   RCX: Pointer to virtqueue state structure
;   RDX: Descriptor index to submit
;
; Outputs:
;   RAX: 0 on success, non-zero on error (queue full)
;
; Clobbers:
;   RAX, RDX, R8, R9, R10, R11, RFLAGS
;
; Memory Ordering:
;   - SFENCE after descriptor write
;   - SFENCE after available ring update
;   - MFENCE before device notification
;
; Invariants Enforced:
;   - VQ-2: Available ring consistency
;   - MEM-ORD-1: Submission ordering
;
; Reference:
;   VirtIO Spec 1.2, Section 2.7.4
;
================================================================================
==
```

6.7 Phase 7: Integration Testing (Week 13-14)

Objective: Verify complete system works correctly.

6.7.1 Test Matrix

Test	Environment	Pass Criteria
DHCP acquisition	QEMU + VirtIO	IP obtained in <10s
HTTP download (small)	QEMU + VirtIO	1MB file, correct checksum
HTTP download (large)	QEMU + VirtIO	100MB file, no timeout
Main loop timing	QEMU + VirtIO	Max iteration <5ms
Memory bounds	QEMU + VirtIO	Total <4MB

6.7.2 Real Hardware Testing

Test	Hardware	Pass Criteria
Cache coherency	Intel NUC or similar	DMA works without corruption
TSC calibration	Multiple machines	Frequency detected correctly
Link detection	Real NIC	Link state detected

6.8 Documentation Maintenance Process

6.8.1 Code-Documentation Coupling

For every code change:

1. **Before coding**: Update design doc with planned change
2. **During coding**: Add inline comments referencing design doc sections
3. **After coding**: Verify doc matches implementation

6.8.2 Invariant Verification

Each invariant in Part 2 should have:

- ☐ Code location where it's enforced
- ☐ Test that verifies it
- ☐ CI check that prevents regression

6.8.3 Living Document Process

```
MONTHLY REVIEW CHECKLIST:

[ ] All code changes reflected in docs
[ ] All invariants still valid
[ ] All spec references still current
[ ] All performance claims still accurate
[ ] No new assumptions introduced without documentation
```

6.9 Document Hierarchy

After completing this plan, the documentation structure should be:

```
docs/
├── NETWORK_STACK_AUDIT.md        # This document (authoritative design)
├── NETWORK_STACK_REDESIGN.md     # Original (deprecated, keep for history)
├── SMOLTCP_INTEGRATION.md        # smoltcp-specific integration notes
├── VIRTIO_COMPLIANCE.md          # VirtIO spec compliance notes
├── BASELINE.md                   # Performance baseline measurements
├── REFACTOR_BLOCKING_PATTERNS.md # Existing (update with new patterns)
└── TESTING.md                    # Test procedures and results
```

6.10 Success Criteria

The documentation effort is complete when:

- ☐ All invariants from Part 2 are implemented and tested
- ☐ All gaps from Part 4 are resolved or explicitly deferred
- ☐ All blocking patterns replaced with state machines
- ☐ Main loop iteration time verified <5ms
- ☐ DHCP + HTTP download works end-to-end
- ☐ At least one real hardware test passes
- ☐ Documentation matches implementation

Appendix A: Glossary

Term	Definition
EBS	ExitBootServices — UEFI call that ends Boot Services phase
DMA	Direct Memory Access — device reads/writes memory directly
TSC	Time Stamp Counter — CPU cycle counter
UC	Uncached — memory type with no caching
WB	Write-Back — memory type with full caching
Virtqueue	VirtIO's ring buffer structure for device communication
smoltcp	Rust no_std TCP/IP stack library
HAL	Hardware Abstraction Layer

Appendix B: Reference Documents

1. **VirtIO Specification 1.2** (OASIS, 2022) https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html
2. **UEFI Specification 2.10** (UEFI Forum, 2022) https://uefi.org/specifications
3. **Intel 64 and IA-32 Architectures Software Developer's Manual**
   https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
4. **smoltcp Documentation** https://docs.rs/smoltcp/
5. **virtio-drivers Crate** https://github.com/rcore-os/virtio-drivers

---

Appendix C: Change Log

Date	Version	Change

Date	Version	Change
2026-01-09	1.0	Initial audit document created
2026-01-09	1.1	Expert review corrections incorporated (Part 7)

Part 7: Expert Review & Authoritative Corrections

This section incorporates expert feedback that refines, corrects, or supersedes analysis in Parts 1-6. Where conflicts exist, Part 7 is authoritative.

7.1 Confirmed Assumptions (Validated with Nuance)

7.1.1 UEFI Identity Mapping — CONFIRMED WITH SCOPE

Original Analysis: Questioned whether identity mapping exists
Expert Confirmation:

> In UEFI long mode before ExitBootServices(), any memory described in the UEFI memory map is identity-mapped (virtual == physical). In practice firmware enables flat paging with all "UEFI-managed" memory identity-mapped.

Critical Nuance: Identity mapping **only covers regions in the UEFI memory map**. Other regions (and any changes after EBS) are undefined. The OS/driver must explicitly manage page tables after ExitBootServices.

Revised Requirement:

- Before EBS: Rely on UEFI's identity mapping for allocated regions
- After EBS: Either continue with firmware's CR3 or install custom CR3 that also identity-maps DMA regions
- Verify CR3 and PTE settings after EBS

7.1.2 DMA Address vs CPU Physical Address — CONFIRMED

Expert Confirmation:

> Device DMA uses bus addresses, not CPU virtual addresses. Without an IOMMU or special mapping, a device's DMA address is not guaranteed to equal the CPU's physical address.

Key Insight: UEFI provides DMA services via `PCI_ROOT_BRIDGE_IO_PROTOCOL`:

- `AllocateBuffer()` — allocates memory suitable for DMA
- `Map()` — obtains the device-visible (bus) address

Revised Requirement: Use UEFI PCI I/O protocols, not raw `AllocatePages()`.

7.1.3 Cache Coherency — CONFIRMED: mfence DOES NOT FLUSH CACHES

Expert Confirmation:

> The CPU's write-combining or write-back caches are not coherent with DMA devices. On x86, the mfence does not flush CPU caches to memory. Without explicit cache flush/invalidate, CPUs and devices can have "cacheline sharing problems".

Revised Requirement: Either:

- Use uncached/write-combining memory for DMA buffers, OR
- Explicitly flush cache lines (CLFLUSH) before/after DMA

7.1.4 TSC Invariant Requirement — CONFIRMED

Expert Confirmation:

> Most x86-64 CPUs today include an invariant TSC that runs at a constant rate regardless of power-saving P-states or turbo. This is indicated by CPUID.80000007H:EDX bit 8.

Revised Requirement:

- Check CPUID leaf 0x80000007 bit 8 for invariant TSC support
- Calibrate TSC frequency at runtime (cannot hardcode)
- Use CPUID leaf 0x15/0x16 or timing against known timer

7.1.5 TSC Monotonicity — CONFIRMED FOR SINGLE CORE ONLY

Expert Confirmation:

> On a single CPU core, an invariant TSC is strictly monotonic. However, TSCs on different CPU cores are not guaranteed synchronized. The Intel manuals state "no promise that the timestamp counters of multiple CPUs will be synchronized".

Revised Requirement: Pin execution to one core or disable CPU migrations when using TSC.

7.1.6 UEFI Calling Convention — CONFIRMED: MS ABI

Expert Confirmation:

> On x86-64, UEFI uses the Microsoft "Win64" calling convention (first four integer args in RCX/RDX/R8/R9, stack alignment to 16 bytes). Rust's default (on Linux targets) is the SysV AMD64 ABI, which is incompatible.

Revised Requirement:

- Compile with `--target x86_64-unknown-uefi` (uses MS ABI)
- All UEFI callbacks and driver entry points must use MS x64 conventions
- Verify inline ASM declares correct register usage

7.1.7 Flat Segments — CONFIRMED

Expert Confirmation:

> UEFI has already set up flat segments with selector bases at 0. The spec states "Selectors are set to be flat and are otherwise not used".

Implication: Can safely ignore segmentation; assume flat 64-bit linear address space.

7.1.8 Interrupts After EBS — CONFIRMED: May Be Disabled

Expert Confirmation:

> After ExitBootServices(), the UEFI spec notes that "interrupts [are] disabled or enabled at the discretion of the caller".

Revised Requirement: Leave interrupts disabled (no IDT). Fully polling design is UEFI-compliant.

7.2 Corrected Assumptions (Supersedes Parts 1-5)

7.2.1 CORRECTION M2: DMA Buffer Allocation

Original (WRONG): "Allocate a static 2MB pool before EBS and assume the NIC will see it"

Corrected: Simply allocating physical memory is **insufficient** for DMA on real hardware.

Required Approach:

```
1. Use EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.AllocateBuffer()
   - Returns memory safe for simultaneous CPU and device access

2. Call Map() to get the bus (device) address
   - Handles IOMMU remapping if present
   - Returns device-visible DMA address

3. Record BOTH:
   - CPU pointer (for software access)
   - Bus address (for device DMA programming)
```

Rationale: The PCI Root Bridge I/O spec explicitly defines `AllocateBuffer()` as "suitable for simultaneous access by both the processor and a PCI bus master".

7.2.2 CORRECTION M3: Cache Coherency Strategy

Original (INCOMPLETE): "Use mfence() to make writes visible to the device"

Corrected: `mfence` does NOT flush CPU caches to memory.

Required Approach:

```
OPTION A: Uncacheable Memory (Recommended)
  1. After AllocateBuffer(), call SetAttributes() with:
     EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE or UNCACHED
  2. No explicit cache management needed

OPTION B: Write-Back with Explicit Flushes
```

```
1. Before device reads (TX submit): CLFLUSH each cache line, then SFENCE
2. After device writes (RX complete): CLFLUSH or LFENCE before CPU reads
3. Use RootBridgeIo->Flush() to ensure posted writes complete
```

Rationale: Linux DMA API documentation confirms that without explicit cache management, "CPUs and devices can have cacheline sharing problems".

7.2.3 CORRECTION V1: VirtIO Version Handling

Original (INCOMPLETE): "Assume virtio 1.0 or 0.9; use MMIO bar"

Corrected: Must explicitly handle both legacy and modern devices.

Required Approach:

```
// 1. Detect device type
let is_virtio = pci_vendor == 0x1AF4 && pci_device == 0x1000;

// 2. Check version via feature bits
let device_features = read_device_features();
let is_modern = device_features & VIRTIO_F_VERSION_1 != 0;

// 3. Use appropriate register layout
if is_modern {
    // Modern: use capability-based config
} else {
    // Legacy: use legacy I/O register layout
}
```

7.2.4 CORRECTION V2: Feature Negotiation — SPECIFIC FEATURES

Original (VAGUE): "Negotiate some unspecified set of features"

Corrected: Negotiate minimal required features explicitly.

Required Features:

Feature	Bit	Action
`VIRTIO_NET_F_MAC`	5	MUST negotiate (get MAC from device)
`VIRTIO_F_VERSION_1`	32	SHOULD negotiate (modern interface)
`VIRTIO_NET_F_STATUS`	16	OPTIONAL (link status)

Forbidden Features (do not negotiate):

Feature	Bit	Reason
`VIRTIO_NET_F_GUEST_TSO4/6`	7,8	We don't support TSO

Feature	Bit	Reason
`VIRTIO_NET_F_GUEST_UFO`	10	We don't support UFO
`VIRTIO_NET_F_CTRL_VQ`	17	Adds complexity

Sequence:

```
// 1. Read device features
let device_features = read_features();

// 2. Select features we support
let our_features = device_features & (
    VIRTIO_NET_F_MAC |
    VIRTIO_F_VERSION_1 |
    VIRTIO_NET_F_STATUS
);

// 3. Write accepted features
write_features(our_features);

// 4. Set FEATURES_OK
write_status(STATUS_FEATURES_OK);

// 5. Verify device accepted
if read_status() & STATUS_FEATURES_OK == 0 {
    return Err(FeatureNegotiationFailed);
}
```

7.2.5 CORRECTION V3: Virtqueue Size

Original (WRONG): "Assume 256 descriptors is fine"

Corrected: Queue size is device-specific; must read `QueueNumMax`.

Required Approach:

```
// 1. Query device maximum
let queue_num_max = read_queue_num_max(queue_index);

// 2. Choose size (power of 2, <= max)
let queue_size = min(desired_size, queue_num_max).next_power_of_two();

// 3. Verify valid
assert!(queue_size >= 2 && queue_size <= 32768);
assert!(queue_size.is_power_of_two());

// 4. Write chosen size
write_queue_num(queue_index, queue_size);
```

7.2.6 CORRECTION V4: Notification Suppression — NOT NEEDED

Original (OVERCOMPLICATED): "Use notification suppression flags"

Corrected: Since we're polling with no interrupts, skip notification flags entirely.

Simplified Approach:

- Set `avail->flags = 0` (device may interrupt, but we ignore)
- Do NOT use `used_event` field
- Poll `used.idx` directly in loop
- Ignore `used->flags` for notification suppression

7.2.7 CORRECTION V5: VirtIO Network Header Size — 12 BYTES

Original (WRONG): "10-byte header"

Corrected: The virtio-net header is **12 bytes** for legacy mode.

Buffer Layout:

```
Offset 0-11:  virtio_net_hdr (12 bytes)
  - flags, gso_type, hdr_len, gso_size, csum_start, csum_offset
Offset 12+:   Ethernet frame (14-byte header + payload)
```

On Receive:

```
let virtio_hdr = &buffer[0..12];   // Skip this
let ethernet_frame = &buffer[12..]; // Pass to smoltcp
```

On Transmit:

```
let mut buffer = [0u8; 12 + MTU];  // 12-byte header + frame
buffer[0..12].fill(0);              // Zero header (no offloads)
buffer[12..12+frame_len].copy_from_slice(frame);
submit_to_tx_queue(&buffer[..12+frame_len]);
```

7.2.8 CORRECTION S1/S2: smoltcp poll() — CALL EXACTLY ONCE

Original (WRONG): "Call poll() many times (once per packet)"

Corrected: Call `Interface::poll()` exactly **once** per main loop iteration.

Rationale: smoltcp internally handles any number of TX/RX operations in that call. It is level-triggered, not edge-triggered.

Correct Main Loop:

```
loop {
    // 1. Refill RX queue with empty buffers
    device.rx_queue_refill();

    // 2. Single poll() call - handles all TX/RX internally
    iface.poll(timestamp, &mut sockets, &mut device);

    // 3. Process TX completions
    device.tx_collect_completions();

    // Do NOT loop calling poll() multiple times
}
```

7.2.9 CORRECTION U1: NIC Initialization — AFTER ExitBootServices

Original (WRONG): "Start NIC before ExitBootServices"

Corrected: Perform hardware initialization **after** ExitBootServices.

Rationale: Firmware drivers may be using the NIC before EBS. The UEFI spec forbids calling Boot Services after EBS.

Correct Sequence:

```
BEFORE EBS:
  1. Scan PCI, record device locations
  2. Allocate DMA memory via UEFI services
  3. Get memory map

CALL ExitBootServices()

AFTER EBS:
  4. Reset NIC
  5. Program PCI BARs/queues
  6. Initialize VirtIO device
  7. Start network operations
```

7.2.10 CORRECTION U3: Memory Type for DMA Region

Original (UNCLEAR): Memory type not specified

Corrected: Mark DMA region with **non-runtime** type.

Required: Use `EfiBootServicesData` so that:

- Memory remains identity-mapped (no `SetVirtualAddressMap()` effect)
- Memory is not reclaimed by firmware

Note: We do not use UEFI Runtime Services after EBS, so no memory remapping occurs.

7.2.11 CORRECTION A2: CLI/STI Are NOT Memory Barriers

Original (WRONG): "Use CLI/STI as a lock (implicit barrier)"

Corrected: `cli` and `sti` only disable/enable interrupts; they do NOT serialize memory.

Intel Documentation: "CLI is not a memory barrier of any sort"

Required Approach:

```
; Correct pattern for critical section with DMA
cli                     ; Disable interrupts (prevents preemption)
; ... prepare descriptors ...
mfence                  ; Memory fence (ensures ordering for DMA)
; ... notify device ...
sti                     ; Re-enable interrupts (if needed)
```

7.2.12 CORRECTION A3: Position-Independent Code

Original (ASSUMPTION): "Code is at fixed addresses 0x50000, 0x60000"

Corrected: On x86-64, use RIP-relative addressing for all data references.

Required:

- Compile code as relocatable (PIC)
- Use RIP-relative addressing for global data
- Mark sections as executable or data for MMU protection

7.3 Revised Implementation Requirements

Based on expert corrections, the implementation MUST:

7.3.1 Memory Allocation (Supersedes Section 5.1.2)

```rust
/// Correct DMA buffer allocation using UEFI PCI I/O Protocol
fn allocate_dma_buffer(
    pci_io: &PciRootBridgeIo,
    size: usize,
) -> Result<DmaBuffer> {
    // 1. Allocate via UEFI (handles alignment, type)
    let cpu_address = pci_io.allocate_buffer(
        AllocateType::MaxAddress(0xFFFF_FFFF),  // Below 4GB
        MemoryType::BOOT_SERVICES_DATA,
        size / PAGE_SIZE,
    )?;
```

```
    // 2. Map to get device-visible address
    let (bus_address, mapping) = pci_io.map(
        PciIoOperation::BusMasterCommonBuffer,
        cpu_address,
        size,
    )?;

    // 3. Set attributes for cache coherency
    pci_io.set_attributes(
        EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE,
        cpu_address,
        size,
    )?;

    Ok(DmaBuffer {
        cpu_ptr: cpu_address,
        bus_addr: bus_address,
        size,
        mapping,
    })
}
```

7.3.2 VirtIO Initialization (Supersedes Section 5.3.3)

```
fn virtio_net_init(pci_device: &PciDevice, dma: &DmaBuffer) ->
Result<VirtioNet> {
    // 1. Reset device
    write_status(0);

    // 2. Wait for reset (device-specific timing)
    let mut timeout = 100_000;  // 100ms
    while read_status() != 0 && timeout > 0 {
        tsc_delay_us(100);
        timeout -= 100;
    }
    if read_status() != 0 {
        return Err(ResetTimeout);
    }

    // 3. Acknowledge
    write_status(STATUS_ACKNOWLEDGE);

    // 4. Driver
    write_status(STATUS_ACKNOWLEDGE | STATUS_DRIVER);

    // 5. Feature negotiation (corrected)
    let device_features = read_features();
    let our_features = device_features & (
        VIRTIO_NET_F_MAC |
        VIRTIO_F_VERSION_1 |
        VIRTIO_NET_F_STATUS
```

```
    );

    // Reject if MAC not available and we need it
    if our_features & VIRTIO_NET_F_MAC == 0 {
        // Either fail or generate random MAC
    }

    write_features(our_features);
    write_status(STATUS_ACKNOWLEDGE | STATUS_DRIVER | STATUS_FEATURES_OK);

    // 6. Verify features accepted
    if read_status() & STATUS_FEATURES_OK == 0 {
        write_status(STATUS_FAILED);
        return Err(FeatureNegotiationFailed);
    }

    // 7. Virtqueue setup (corrected - read max size)
    let rx_queue_max = read_queue_num_max(0);
    let tx_queue_max = read_queue_num_max(1);

    let rx_queue_size = min(32, rx_queue_max);
    let tx_queue_size = min(32, tx_queue_max);

    setup_virtqueue(0, rx_queue_size, dma)?;
    setup_virtqueue(1, tx_queue_size, dma)?;

    // 8. Pre-fill RX with 12-byte header space
    for i in 0..rx_queue_size {
        submit_rx_buffer(i, 12 + MTU)?;  // Header + frame
    }

    // 9. Driver OK
    write_status(STATUS_ACKNOWLEDGE | STATUS_DRIVER |
                 STATUS_FEATURES_OK | STATUS_DRIVER_OK);

    // 10. Get MAC (if negotiated)
    let mac = if our_features & VIRTIO_NET_F_MAC != 0 {
        read_mac_from_config()
    } else {
        generate_local_mac()
    };

    Ok(VirtioNet { mac, rx_queue_size, tx_queue_size, ... })
}
```

7.3.3 Main Loop (Supersedes Section 5.5)

```
/// Correct main loop — single poll() per iteration
fn main_loop(
    device: &mut VirtioNet,
    iface: &mut Interface,
```

```
    sockets: &mut SocketSet,
    tsc_freq: u64,
) -> ! {
    loop {
        let now_tsc = rdtsc();
        let timestamp = Instant::from_millis(tsc_to_ms(now_tsc, tsc_freq));

        // Phase 1: Refill RX queue with available buffers
        device.rx_queue_refill();

        // Phase 2: Single smoltcp poll (handles all TX/RX internally)
        // Do NOT call this multiple times per iteration
        iface.poll(timestamp, sockets, &mut DeviceAdapter::new(device));

        // Phase 3: Collect TX completions (return buffers to pool)
        device.tx_collect_completions();

        // Phase 4: Application state machine
        // ... (unchanged from Section 5.5)
    }
}
```

7.4 Authoritative Source Citations (from Expert Review)

Topic	Source	Key Quote/Reference
UEFI Identity Mapping	UEFI Spec 2.10	"memory defined by the UEFI memory map is identity mapped"
DMA Buffer Allocation	PCI Root Bridge I/O Protocol	"AllocateBuffer() suitable for simultaneous access by processor and PCI bus master"
Cache Coherency	Linux DMA API Docs	"without explicit cache flush/invalidate, CPUs and devices can have cacheline sharing problems"
CLI Not Barrier	Intel SDM	"CLI is not a memory barrier of any sort"
TSC Invariant	CPUID Documentation	"CPUID.80000007H:EDX bit 8"
TSC Multi-core	Intel SDM	"no promise that the timestamp counters of multiple CPUs will be synchronized"
VirtIO Features	VirtIO Spec 1.1	Feature negotiation, virtqueue size constraints
VirtIO Header	VirtIO Spec 1.1	"virtio_net_hdr" structure definition (12 bytes)

7.5 Updated Compliance Matrix

Requirement	Original Status	Corrected Status	Action Required
DMA allocation	✗ Raw AllocatePages	⚠ Must use PCI I/O	Use AllocateBuffer/Map
Cache coherency	✗ mfence only	⚠ Need UC/WC or flush	Set attributes or CLFLUSH
VirtIO header	✗ 10 bytes	✓ 12 bytes	Update buffer offsets
Feature negotiation	⚠ Vague	✓ Specific features	Implement per 7.2.4
Queue size	✗ Hardcoded 256	⚠ Read from device	Query QueueNumMax
smoltcp poll	✗ Multiple calls	✓ Single call	Restructure main loop
NIC init timing	✗ Before EBS	✓ After EBS	Reorder boot sequence
CLI as barrier	✗ Assumed barrier	✓ Add mfence	Explicit memory fences

End of Part 7. This section supersedes conflicting content in Parts 1-5.

Appendix D: Expert Review Sources

The corrections in Part 7 are derived from:

1. **UEFI Specification 2.10** — Memory map identity mapping, calling conventions, EBS behavior
2. **PCI Root Bridge I/O Protocol (UEFI)** — DMA buffer allocation, attributes, Map/Unmap
3. **VirtIO Specification 1.1** — Feature negotiation, virtqueue constraints, network header
4. **Intel 64 and IA-32 Architectures Software Developer's Manual** — CLI semantics, TSC behavior
5. **Linux Kernel DMA API Documentation** — Cache coherency requirements
6. **smoltcp Design Guidelines** — poll() semantics, Device trait requirements

---

Document End

This document supersedes NETWORK_STACK_REDESIGN.md for all design decisions. Implementation must conform to specifications in Part 5, as amended by Part 7. Part 7 corrections are authoritative where conflicts exist. Deviations must be documented with rationale.