MorpheusX Single-Core Bare-Metal Network Stack v1 Architecture

Document Status

- **Version**: 1.0 (Frozen)
- **Date**: 2026-01-09
- **Scope**: Post-ExitBootServices Bare-Metal Network Stack
- **Authority**: This document is reconciled against NETWORK_STACK_AUDIT.md

---

Execution Model (Locked)

This section defines the fundamental execution model for the v1 network stack. These constraints are **non-negotiable** and apply to all code within this architecture.

Property	Value	Notes
Core count	1 (BSP only)	No SMP, no inter-processor interrupts
Thread count	1	No threading runtime, no task switching
Interrupt state	Disabled	No ISRs, no timer interrupts
Scheduling	Cooperative	Progress between function returns
Execution pattern	Poll-driven	Explicit polling, no event callbacks
Phase orientation	Sequential	RX → Process → TX → App → Repeat
Preemption	None	No preemptive context switches

Main Loop Contract:

```
loop {
    Phase 1: poll_rx()           // Collect received packets
    Phase 2: smoltcp.poll()      // Process protocols (exactly once)
    Phase 3: drain_tx()          // Submit outbound packets
    Phase 4: app_state.step()    // Advance application state
    Phase 5: collect_completions() // Handle TX completions
}
```

Invariants:

- Every function call returns in bounded time
- No function may yield, await, or suspend
- No function may loop waiting for external state
- The main loop is the ONLY execution entry point

Audit Reference: §2.8 (LOOP-1, LOOP-2), §7.2.8

Time & Progress Guarantees

This section documents what the v1 architecture **guarantees** and what it **does not guarantee** regarding timing.

Guaranteed

Property	Guarantee
TSC monotonicity	TSC increases monotonically (single core)
TSC invariance	TSC frequency does not change with P-states
Timeout detection	Timeouts are detectable via TSC comparison
Bounded iteration	Each main loop phase completes in bounded cycles
Progress	Each poll advances state if work is available

NOT Guaranteed

Property	Why Not Guaranteed
Wall-clock accuracy	TSC frequency varies per-CPU; calibrated, not exact
Latency bounds	Hypervisor may preempt guest; firmware SMI possible
Throughput	Single-core polling is CPU-bound
Real-time deadlines	No real-time kernel; best-effort only
Fairness	Single operation at a time; no fairness needed

Contract: All timeout values are **advisory**. The system makes best-effort progress but cannot guarantee deadlines.

Audit Reference: §7.1.4, §7.1.5

DMA & Memory Model Assumptions

This section documents all memory model assumptions for DMA-based NIC operation.

x86-64 Memory Model

Property	Assumed Value	Notes
Cache coherency	**Yes**	x86 maintains cache coherency for DMA
Store ordering	TSO (Total Store Order)	Stores visible in program order
Load ordering	TSO	Loads visible in program order

Property	Assumed Value	Notes
Store-Load reorder	Possible	Requires mfence if ordering needed

DMA Requirements

Requirement	Mechanism
Allocation	`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.AllocateBuffer()`
Address translation	`Map()` returns bus address for device descriptors
CPU access	Use CPU pointer from AllocateBuffer directly
Device access	Use bus_addr from Map() in descriptor tables
Memory type	UC or WC preferred; cacheable works on x86

Memory Barriers

Barrier	When Used
`sfence`	Before device notification (ensure stores visible)
`lfence`	After reading device-written memory (ensure loads complete)
`mfence`	When both load and store ordering needed
`clflush`	NOT required on x86 for DMA (cache coherent)

Contract:

- `mfence` ensures store visibility, NOT cache flush
- `cli`/`sti` are NOT memory barriers
- Compiler barriers (`volatile`, `asm!`) prevent compiler reordering only

Audit Reference: §7.2.2, §7.2.3, §7.2.11

Explicit Non-Goals (v1)

This section documents architectural exclusions. These are not deferred features—they are explicit scope boundaries for v1.

Non-Goal	Rationale
Interrupt support	Polling sufficient; avoids IDT complexity
Multi-core / SMP	Avoids cache coherency, locking complexity
Async runtime	No futures, no executor; explicit state machines
Scheduler	Single execution context; no scheduling needed
Kernel abstraction	We ARE the kernel; direct hardware access

Non-Goal	Rationale
TLS / HTTPS	Trusted network; crypto complexity avoided
IPv6	IPv4 sufficient for bootstrap use case
Jumbo frames	Standard 1500 MTU sufficient
Zero-copy	Simplicity over performance
Fair queuing	Single stream; no fairness needed
Congestion control tuning	smoltcp defaults sufficient
Hardware offloads	Checksum in software; portable

These exclusions are architectural. Violating them requires a new architecture document.

Audit Reference: §5.1 (scope), §7.1.1 (constraints)

Section 1: Problem Statement

1.1 The Fundamental Deadlock

The current MorpheusX network stack has a **fatal architectural flaw**: it attempts to use blocking patterns in a single-threaded, cooperative environment without a scheduler.

Current TX Path (Broken):

```
smoltcp.poll()
  → wants to send DHCP DISCOVER
  → calls DeviceAdapter::transmit()
  → calls TxToken::consume()
  → calls VirtioNetDevice::transmit()
  → BLOCKS: loop { poll_transmit(); tsc_delay_us(10); }
  → Waits for VirtIO device to process packet
  → DEADLOCK: Main loop never advances; no RX/TX progress
```

Why This Happens:

1. VirtIO is designed for interrupt-driven operation
2. Guest submits buffer → hypervisor processes → hypervisor signals completion
3. Without interrupts, we must poll for completion
4. Polling inside `transmit()` blocks the entire system
5. smoltcp.poll() never returns; no other work can happen

Audit Reference: §3.7, §4.2.1 confirm this is the root cause.

1.2 The UEFI Interference Problem

Before `ExitBootServices()`, UEFI firmware:

- Maintains its own memory pools and DMA mappings
- May intercept PCI/MMIO accesses via its own drivers
- Has active SNP/NII protocols that may be using the NIC
- Can move memory regions unpredictably

Consequence: Any NIC driver running while UEFI is active risks:

- Conflicting DMA mappings (double-mapped buffers)
- Race conditions on device registers
- Unexpected device state changes
- Memory corruption

Contract: Full device control requires `ExitBootServices()` FIRST.

Audit Reference: §7.2.9 confirms NIC initialization MUST occur AFTER ExitBootServices.

1.3 The Memory Ordering Problem

Correct NIC operation requires precise memory ordering. The following are distinct concerns:

Concern	Mechanism	Notes
Compiler reordering	`volatile` / inline ASM	Prevents Rust optimizer reordering
CPU store buffer	`sfence` / `mfence`	Ensures stores visible to other agents
Cache coherency	UC mapping OR `clflush`	**mfence does NOT flush cache**

Critical Correction: `cli`/`sti` are NOT memory barriers. They only control interrupt delivery.

Contract: DMA buffer regions MUST be mapped Uncached (UC) or Write-Combining (WC), OR explicit cache line flushes MUST be performed before device access.

Audit Reference: §7.2.2, §7.2.3, §7.2.11 — mfence does not ensure cache coherency; CLI is not a barrier.

1.4 Current Blocking Violations

Location	Pattern	Severity
`virtio.rs:322`	`loop { poll_transmit(); delay(); }`	CRITICAL
`native.rs:202`	`while !has_ip() { poll(); delay(); }`	CRITICAL
`native.rs:261`	DNS resolution loop	CRITICAL
`native.rs:352`	TCP connect wait	CRITICAL
`native.rs:380`	`send_all()` loop	CRITICAL
`native.rs:407`	`recv()` loop	CRITICAL
`init.rs:205`	DHCP wait loop	HIGH
`pci.rs:252`	`tsc_delay_us()` spin	CRITICAL (root cause)

Root Cause: `tsc_delay_us()` is the blocking primitive enabling all violations.

1.5 Section 1 Invariant Summary

Invariant	Description
BLOCK-1	No function may loop waiting for external state change
BLOCK-2	TX submission returns immediately; completion is collected separately
BLOCK-3	RX polling returns immediately with `Some(packet)` or `None`
EBS-1	NIC initialization occurs only AFTER ExitBootServices
MEM-1	DMA regions are Uncached OR cache-flushed before device access

Section 2: Design Constraints (Locked)

2.1 Hardware Constraints

Constraint	Value	Rationale
CPU Cores	1 (BSP only)	No SMP synchronization complexity
Interrupt State	Disabled	Pure polling; no ISRs
Paging	Enabled, identity-mapped	UEFI leaves paging on; we maintain identity map for UEFI-managed regions
Memory Model	x86-64 TSO	Strong ordering; explicit barriers for DMA only
Floating Point	Disabled	no_std, soft-float; no FPU state save/restore
Stack	Pre-allocated, 64KB minimum	Allocated before EBS, survives EBS

Audit Reference: §5.1.1, §7.1.1, §7.1.7

2.2 Execution Model Constraints

Constraint	Description
All functions must return	No function may block indefinitely
Bounded execution time	Every call completes in finite cycles
Cooperative scheduling	Progress happens between calls, not within
Explicit state machines	All multi-step operations use state enums
Single poll() per iteration	smoltcp.poll() called exactly once per main loop

Contract: Every function call returns in bounded time. There are no exceptions.

Audit Reference: §2.8 LOOP-1, LOOP-2; §7.2.8 confirms single poll() call

2.3 UEFI Lifecycle Constraints

Phase	What's Available	What's Forbidden
Pre-ExitBootServices	Memory allocation, console, PCI I/O Protocol	Full NIC control, DMA submission
Post-ExitBootServices	Raw hardware access, identity map	Any UEFI Boot Service call

Memory Type Contract: DMA region MUST be allocated as `EfiBootServicesData`:

- Survives ExitBootServices
- Not affected by `SetVirtualAddressMap()` (we don't call it)
- Must be below 4GB for 32-bit DMA compatibility

DMA Allocation Contract: Use `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`:

- `AllocateBuffer()` — returns memory suitable for DMA
- `Map()` — returns device-visible bus address (handles IOMMU if present)
- `SetAttributes()` — set UC/WC memory type

Audit Reference: §7.2.1 (DMA allocation), §7.2.10 (memory type)

2.4 Timing Constraints

Constraint	Description
Time source	TSC (invariant TSC required)
TSC calibration	MUST calibrate at boot; no hardcoded frequency
Timeout mechanism	Check TSC delta; never spin-wait
Maximum loop time	< 5ms per main loop iteration

TSC Contract:

1. MUST verify CPUID.80000007H:EDX bit 8 (invariant TSC)
2. MUST calibrate TSC frequency using UEFI Stall() or PIT before EBS
3. TSC is monotonic on single core only (we are single core)
4. Use `wrapping_sub()` for timeout comparisons

Audit Reference: §1.2 T1-T4, §7.1.4, §7.1.5

2.5 Section 2 Invariant Summary

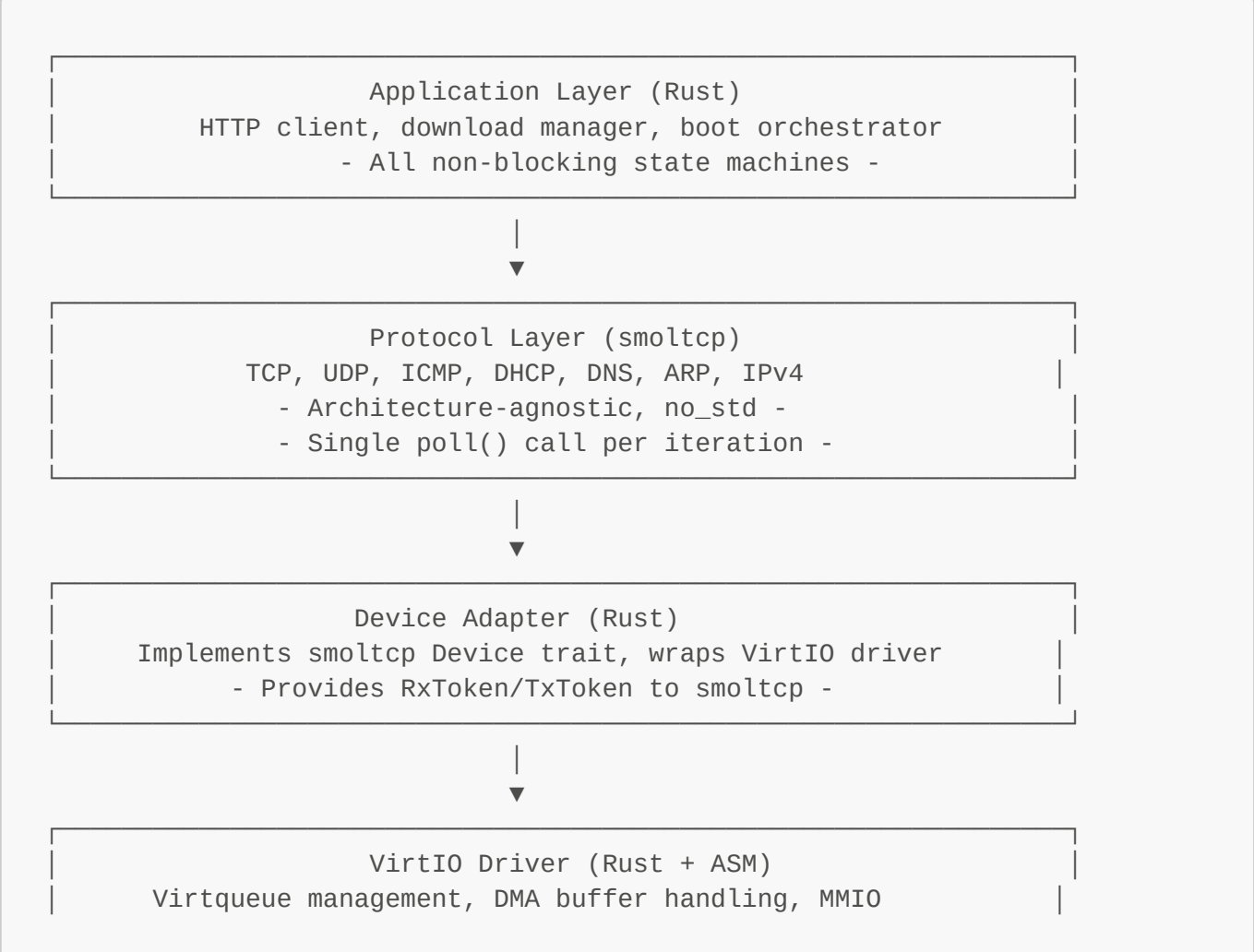
Invariant	Description

Invariant	Description
EXEC-1	Single core, single thread, no preemption
EXEC-2	Interrupts disabled throughout execution
EXEC-3	Every function returns in bounded time
EXEC-4	smoltcp.poll() called exactly once per main loop iteration
MEM-2	DMA region allocated via PCI I/O Protocol, not raw AllocatePages
MEM-3	DMA region is EfiBootServicesData, mapped UC or WC
MEM-4	Identity mapping only guaranteed for UEFI-described regions
TIME-1	TSC invariant feature MUST be verified at boot
TIME-2	TSC frequency MUST be calibrated, never hardcoded
TIME-3	No spin-waits; timeouts are checked, not waited

Non-Goals: See global "Explicit Non-Goals (v1)" section for architectural exclusions.

Section 3: Architecture Overview (Locked)

3.1 Layer Stack



```
│              - Memory barriers in ASM where needed -                │
├─────────────────────────────────────────────────────────────────────┤
│  Key Operations:                                                    │
│    submit_rx_buffer(idx) → void                                     │
│    poll_rx() → Option<(idx, len)>                                   │
│    submit_tx(buf, len) → Result<()>                                 │
│    poll_tx_complete() → Option<idx>                                 │
│    notify_device(queue) → void                                      │
└─────────────────────────────────────────────────────────────────────┘
                                │
                                ▼
┌─────────────────────────────────────────────────────────────────────┐
│                    Hardware (VirtIO-net)                            │
│          PCI MMIO registers, Virtqueues, DMA buffers                │
│            - 12-byte virtio_net_hdr + Ethernet frame -              │
└─────────────────────────────────────────────────────────────────────┘
```

3.2 The Main Loop (Single Entry Point)

All network activity flows through ONE loop with ONE smoltcp.poll() call per iteration.

```
┌─────────────────────────────────────────────────────────────┐
│                    MAIN POLL LOOP                           │
│                                                             │
│  loop {                                                     │
│      let now_tsc = rdtsc();                                 │
│      let timestamp = tsc_to_instant(now_tsc);               │
│                                                             │
│      // Phase 1: Refill RX queue with available buffers     │
│      device.rx_queue_refill();                              │
│                                                             │
│      // Phase 2: Single smoltcp poll (handles all TX/RX)    │
│      // Do NOT call this multiple times per iteration       │
│      iface.poll(timestamp, &mut device, &mut sockets);      │
│                                                             │
│      // Phase 3: Collect TX completions (return buffers)    │
│      device.tx_collect_completions();                       │
│                                                             │
│      // Phase 4: Application state machine step             │
│      app_state.step(now_tsc);                               │
│  }                                                          │
│                                                             │
│  INVARIANT: Each iteration < 5ms                            │
│  INVARIANT: poll() called exactly once                      │
└─────────────────────────────────────────────────────────────┘
```

Audit Reference: §5.5, §7.2.8 — smoltcp.poll() is level-triggered; one call handles all pending work.

3.3 Buffer Ownership Model

```
INVARIANT DMA-OWN-1: Buffer Exclusive Ownership

  At any time, each buffer is in EXACTLY ONE state:

     ┌──────────┐      alloc()      ┌─────────────────┐
     │  FREE    │ ─────────────────▶ │ DRIVER-OWNED    │
     └──────────┘                    └─────────────────┘
          ▲                                  │
          │                                  │ submit to virtqueue
          │ free()                           ▼
          │                         ┌─────────────────┐
          └──────────────────────── │ DEVICE-OWNED    │
               poll returns         └─────────────────┘

  DRIVER-OWNED: Rust code may read/write buffer
  DEVICE-OWNED: Driver MUST NOT access buffer
  FREE: Buffer available for allocation
```

RX Buffer Lifecycle:

1. Allocate buffer → DRIVER-OWNED
2. Submit to RX virtqueue → DEVICE-OWNED
3. Device receives packet, marks used → still DEVICE-OWNED
4. Driver polls used ring → DRIVER-OWNED
5. smoltcp RxToken consumes data
6. Resubmit to virtqueue OR free

TX Buffer Lifecycle:

1. Allocate buffer → DRIVER-OWNED
2. Fill with packet data (12-byte header + frame)
3. Submit to TX virtqueue → DEVICE-OWNED
4. Device transmits, marks used → still DEVICE-OWNED
5. Driver polls used ring → DRIVER-OWNED
6. Free buffer (or reuse)

3.4 Key Invariants (Locked)

ID	Invariant	Enforcement
LOOP-1	No function loops waiting for external state	Code review; no `while !condition` patterns
LOOP-2	Main loop iteration < 5ms	Bounded budgets per phase
POLL-1	smoltcp.poll() called exactly once per iteration	Structural; single call site
TX-1	TX submit returns immediately	No completion wait
TX-2	TX completion collected opportunistically	Separate phase in loop

ID	Invariant	Enforcement
RX-1	RX poll returns immediately with Some or None	Non-blocking virtqueue poll
RX-2	RX buffer remains valid while RxToken exists	Lifetime enforcement
TIME-4	Time is observed, not waited	TSC checked, never spun on
STATE-1	Multi-step operations are state machines	Enum per operation

3.5 Section 3 Invariant Summary

Execution Model:

- Single core, single thread, poll-driven, phase-oriented
- No preemption, no interrupts, no async

Memory Ownership:

- Buffers have exactly one owner at any time
- Ownership transfers are explicit (submit/poll)
- Driver never accesses DEVICE-OWNED buffers

Progress Guarantees:

- Main loop always makes progress (no blocking)
- Each phase has bounded work
- smoltcp handles retransmission timers internally

Failure Semantics:

- TX queue full → backpressure (requeue packet)
- RX queue full → drop packet (device behavior)
- Timeout → state machine transitions to Failed

Section 4: VirtIO Driver Specification (Locked)

4.1 Why Explicit Memory Barriers?

Memory barriers are required for correctness, not just performance.

Concern	Mechanism	When Required
Compiler reordering	`volatile` + compiler fence	All MMIO and DMA buffer access
CPU store buffer	`sfence`	Before updating avail.idx
Full serialization	`mfence`	Before device notification
Cache coherency	UC memory type OR `clflush`	DMA buffer setup

Critical Correction: `cli`/`sti` are NOT memory barriers. They only control interrupt delivery. Do not use them for synchronization.

Audit Reference: §7.2.11

4.2 VirtIO Feature Negotiation (Locked)

Required Features (device MUST support; reject otherwise):

Feature	Bit	Purpose
`VIRTIO_F_VERSION_1`	32	Modern device protocol

Desired Features (negotiate if available):

Feature	Bit	Purpose
`VIRTIO_NET_F_MAC`	5	MAC address in config space
`VIRTIO_NET_F_STATUS`	16	Link status reporting
`VIRTIO_F_RING_EVENT_IDX`	29	Efficient notification suppression

Forbidden Features (MUST NOT negotiate):

Feature	Bit	Reason
`VIRTIO_NET_F_GUEST_TSO4`	7	No TSO support
`VIRTIO_NET_F_GUEST_TSO6`	8	No TSO support
`VIRTIO_NET_F_GUEST_UFO`	10	No UFO support
`VIRTIO_NET_F_MRG_RXBUF`	15	Simplifies header handling
`VIRTIO_NET_F_CTRL_VQ`	17	Unnecessary complexity

Audit Reference: §7.2.4, §5.3.1

4.3 Device Initialization Sequence (Locked)

```
SEQUENCE (must execute in this exact order):

1. RESET
   - Write 0 to status register
   - Wait until status reads 0 (with timeout)
   - Wait additional 100ms for conservative reset completion

2. ACKNOWLEDGE
   - Write STATUS_ACKNOWLEDGE (0x01) to status

3. DRIVER
   - Write STATUS_ACKNOWLEDGE | STATUS_DRIVER (0x03) to status
```

```
4. FEATURE NEGOTIATION
   - Read device_features register
   - Compute our_features = device_features & ALLOWED_FEATURES
   - Verify required features present
   - Write our_features to driver_features register

5. FEATURES_OK
   - Write STATUS_FEATURES_OK (0x08) to status
   - Read status back
   - If FEATURES_OK not set → device rejected; write STATUS_FAILED, abort

6. VIRTQUEUE SETUP
   - For each queue (RX=0, TX=1):
     a. Write queue index to queue_select
     b. Read queue_num_max (device maximum size)
     c. Choose size = min(32, queue_num_max)
     d. Write size to queue_num
     e. Write descriptor table address
     f. Write available ring address
     g. Write used ring address
     h. Write 1 to queue_enable

7. PRE-FILL RX QUEUE
   - Submit 32 empty buffers to RX queue
   - Each buffer: 12-byte header space + 1514-byte frame space

8. DRIVER_OK
   - Write STATUS_DRIVER_OK (0x04) to status
   - Device is now operational

ON ERROR at any step:
   - Write STATUS_FAILED (0x80) to status
   - Return initialization failure
```

Audit Reference: §5.3.3, §7.2.3

4.4 Virtqueue Size Contract

Virtqueue size is NOT hardcoded. It MUST be negotiated:

```
CONTRACT: Queue Size Negotiation

1. Read queue_num_max from device
2. Choose queue_size where:
   - queue_size >= 2
   - queue_size <= queue_num_max
   - queue_size <= 32768
   - queue_size is power of 2
3. Write queue_size to queue_num register
4. Allocate structures accordingly
```

v1 Default: Use 32 entries if device supports it.

Audit Reference: §7.2.5

4.5 Memory Barrier Contracts

TX Submit Sequence

```
CONTRACT MEM-ORD-1: TX Submission Ordering

SEQUENCE (must execute in this order):
  1. WRITE: descriptor.addr, descriptor.len, descriptor.flags
  2. BARRIER: sfence (ensures descriptor visible before index)
  3. WRITE: avail.ring[idx & mask] = descriptor_index
  4. BARRIER: sfence (ensures ring entry visible before index)
  5. WRITE: avail.idx += 1
  6. BARRIER: mfence (full barrier before notify)
  7. CONDITIONAL: if notification needed, MMIO write to notify register

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

RX Completion Sequence

```
CONTRACT MEM-ORD-2: RX Completion Polling

SEQUENCE:
  1. READ: used.idx (volatile read)
  2. COMPARE: if used.idx == last_used_idx, return None
  3. BARRIER: lfence (ensures index read before entry read)
  4. READ: used.ring[last_used_idx & mask]
  5. READ: buffer contents
  6. UPDATE: last_used_idx += 1
  7. RETURN: buffer to caller

RATIONALE:
  - lfence prevents speculative read of ring entry
  - Ensures we see device's writes before reading buffer
```

Audit Reference: §2.3, §5.4

4.6 VirtIO Network Header (Locked)

Header Size: 12 bytes (for modern VIRTIO_F_VERSION_1 devices)

```rust
#[repr(C)]
pub struct VirtioNetHdr {
    pub flags: u8,          // VIRTIO_NET_HDR_F_*
    pub gso_type: u8,       // VIRTIO_NET_HDR_GSO_*
    pub hdr_len: u16,       // Ethernet + IP + TCP header length
    pub gso_size: u16,      // GSO segment size (0 if no GSO)
    pub csum_start: u16,    // Checksum start offset
    pub csum_offset: u16,   // Checksum offset from csum_start
    pub num_buffers: u16,   // Only with MRG_RXBUF (we don't use it)
}
```

For our use (no offloads): Zero the entire 12-byte header.

Buffer Layout:

```
Offset 0-11:  VirtioNetHdr (12 bytes, zeroed)
Offset 12+:   Ethernet frame (14-byte header + IP payload)
Total:        12 + 1514 = 1526 bytes minimum per buffer
```

Audit Reference: §7.2.7

4.7 DMA Memory Layout (Locked)

```
DMA Region (2MB minimum, allocated via PCI I/O Protocol):

Offset      Size      Content
──────────────────────────────────────────────────────────────
0x0000      0x200     RX descriptor table (32 × 16 bytes)
0x0200      0x048     RX available ring (4 + 32×2 + 2 bytes, padded)
0x0400      0x108     RX used ring (4 + 32×8 + 2 bytes, padded)
0x0800      0x200     TX descriptor table (32 × 16 bytes)
0x0A00      0x048     TX available ring
0x0C00      0x108     TX used ring
0x1000      0x10000   RX buffers (32 × 2KB = 64KB)
0x11000     0x10000   TX buffers (32 × 2KB = 64KB)
0x21000     ...       Reserved (~1.87MB)
──────────────────────────────────────────────────────────────
Total used: ~136KB
```

Memory Type: Uncached (UC) or Write-Combining (WC) **Alignment**: Page-aligned (4KB boundary)
Address: Below 4GB (32-bit DMA compatibility)

4.8 Notification Suppression (Simplified for v1)

For v1, we use the simplest notification model:

- Set `avail.flags = 0` (do not suppress device interrupts)

- Ignore `used.flags` (we poll anyway)
- Do NOT use `event_idx` feature
- Always notify device after TX submit

Rationale: Simplicity. Notification suppression is an optimization that adds complexity.

Audit Reference: §7.2.6

4.9 Section 4 Invariant Summary

Invariant	Description
VIRTIO-1	Device initialization follows exact status sequence
VIRTIO-2	Feature negotiation rejects forbidden features
VIRTIO-3	Queue size read from device, never hardcoded
VIRTIO-4	Header size is 12 bytes (VERSION_1 devices)
VIRTIO-5	Memory barriers placed per MEM-ORD-1 and MEM-ORD-2
DMA-1	DMA region allocated via PCI I/O Protocol
DMA-2	DMA region mapped UC or WC
DMA-3	DMA region below 4GB
DMA-4	Buffer ownership tracked; never access DEVICE-OWNED

Section 5: Boot Sequence & ExitBootServices Boundary (Locked)

5.1 The Two-Phase Boot Model

```
┌──────────────────────────────────────────────────────────────────┐
│                    PHASE 1: UEFI ACTIVE                          │
│                (Boot Services Available)                         │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│  1. UEFI loads bootloader.efi                                    │
│  2. Bootloader allocates memory via PCI I/O Protocol:            │
│     - AllocateBuffer() for DMA region (returns CPU pointer)      │
│     - Map() to get device-visible bus address                    │
│     - SetAttributes() to mark UC or WC                           │
│  3. Allocate stack (EfiLoaderData, 64KB minimum)                 │
│  4. Calibrate TSC using UEFI Stall() as reference                │
│  5. Scan PCI for VirtIO NIC:                                     │
│     - Record MMIO base addresses                                 │
│     - Does NOT initialize device                                 │
│  6. Get final memory map                                         │
│  7. Populate BootHandoff structure                               │
│                                                                  │
│  ═══════════ POINT OF NO RETURN ═══════════                      │
```

```
│                                                                          │
│  8. Call ExitBootServices(image_handle, map_key)                         │
│     - Retry with fresh map if map_key mismatch                           │
│                                                                          │
└──────────────────────────────────────────────────────────────────────────┘
                                    │
                                    ▼
┌──────────────────────────────────────────────────────────────────────────┐
│                        PHASE 2: BARE METAL                               │
│                    (Boot Services UNAVAILABLE)                           │
├──────────────────────────────────────────────────────────────────────────┤
│                                                                          │
│  9. Switch to pre-allocated stack                                        │
│ 10. Remap DMA region as UC (if not done via SetAttributes)               │
│ 11. Initialize VirtIO NIC (full sequence per §4.3)                       │
│ 12. Create smoltcp Interface with device adapter                         │
│ 13. Enter main poll loop (never returns)                                 │
│                                                                          │
└──────────────────────────────────────────────────────────────────────────┘
```

Audit Reference: §5.2, §7.2.1, §7.2.9

5.2 Pre-ExitBootServices Requirements (Locked)

MUST DO before ExitBootServices():

Task	Method	Why
Allocate DMA region	`EFI_PCI_ROOT_BRIDGE_IO.AllocateBuffer()`	Returns DMA-safe memory
Get bus address	`EFI_PCI_ROOT_BRIDGE_IO.Map()`	Device-visible address (IOMMU-aware)
Set memory attributes	`SetAttributes(UC or WC)`	Cache coherency
Allocate stack	`AllocatePages(EfiLoaderData)`	Survives EBS
Calibrate TSC	`Stall(1_000_000)` + `rdtsc`	Timing reference
Record NIC location	PCI enumeration	MMIO base address
Get memory map	`GetMemoryMap()`	Required for EBS call

MUST NOT DO before ExitBootServices():

Action	Why Not
Reset NIC	UEFI SNP/NII may be using it
Write to NIC MMIO	Conflicts with firmware drivers

Action	Why Not
Submit DMA buffers	Device state owned by UEFI
Set up virtqueues	Wait until post-EBS

Audit Reference: §7.2.1, §7.2.9

5.3 DMA Allocation Contract

```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,  // Or UNCACHED
        cpu_address,
        size,
    )?;

    Ok(DmaBuffer {
        cpu_ptr: cpu_address,       // For Rust code access
        bus_addr: bus_address,      // For device DMA programming
        size,
        mapping,
    })
}
```

Critical: The `bus_addr` may differ from `cpu_ptr` if IOMMU is active. Always use `bus_addr` when programming device descriptors.

Audit Reference: §7.2.1, §7.2.2

5.4 TSC Calibration Contract

```rust
/// Calibrate TSC using UEFI Stall() as reference
fn calibrate_tsc(boot_services: &BootServices) -> u64 {
    let start = unsafe { core::arch::x86_64::_rdtsc() };
    boot_services.stall(1_000_000);  // 1 second
    let end = unsafe { core::arch::x86_64::_rdtsc() };
    end - start  // TSC ticks per second
}

// MUST verify invariant TSC before relying on this
fn verify_invariant_tsc() -> bool {
    // CPUID.80000007H:EDX bit 8
    let result = unsafe { core::arch::x86_64::__cpuid(0x80000007) };
    (result.edx & (1 << 8)) != 0
}
```

Audit Reference: §7.1.4, §2.5 TIME-2

5.5 Post-ExitBootServices Constraints (Locked)

After `ExitBootServices()` returns successfully:

Available	NOT Available
CPU registers, stack	Boot Services (ANY)
Pre-allocated DMA region	Memory allocation
PCI MMIO access (via cached base)	Console output
TSC instruction	File system
Identity-mapped memory	Timer services
Raw port I/O	PCI config via UEFI

5.6 Error Recovery

If ExitBootServices() fails (map_key mismatch):

```rust
loop {
    let (map, key) = get_memory_map()?;
    match exit_boot_services(image, key) {
        Ok(()) => break,
        Err(Status::INVALID_PARAMETER) => continue,  // Retry
        Err(e) => return Err(e),  // Fatal
    }
    retries -= 1;
    if retries == 0 { halt(); }
}
```

If NIC init fails after ExitBootServices():

- No recovery possible (no allocator)
- Log error to serial/framebuffer if available
- Halt with error code

5.7 Handoff Data Structure (Locked)

```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 (0 if none found)
    pub nic_mmio_base: u64,

    /// PCI location: bus, device, function
    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],

    pub _pad1: [u8; 2],

    // === DMA Region ===

    /// DMA region CPU pointer (for Rust code access)
    pub dma_cpu_ptr: u64,

    /// DMA region bus address (for device DMA programming)
    pub dma_bus_addr: u64,

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

    // === Timing ===

    /// TSC frequency in Hz (calibrated at boot)
    pub tsc_freq: u64,
```

```
    // === Stack ===

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

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

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

    /// Framebuffer base for debug output (or 0)
    pub framebuffer_base: u64,
    pub framebuffer_size: u64,
    pub framebuffer_width: u32,
    pub framebuffer_height: u32,
    pub framebuffer_stride: u32,
    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_cpu_ptr != 0 &&
        self.dma_size >= 2 * 1024 * 1024 &&
        self.tsc_freq > 0 &&
        self.stack_top != 0
    }
}
```

Audit Reference: §5.8

5.8 Section 5 Invariant Summary

Invariant	Description
BOOT-1	DMA region allocated via PCI I/O Protocol, not AllocatePages
BOOT-2	DMA bus address stored separately from CPU pointer
BOOT-3	TSC frequency calibrated at boot, never hardcoded
BOOT-4	Invariant TSC verified via CPUID before use
BOOT-5	NIC initialization occurs only AFTER ExitBootServices
BOOT-6	All required data stored in BootHandoff before EBS

Section 6: smoltcp Integration (Locked)

6.1 smoltcp Verified Behaviors

The following smoltcp behaviors have been verified against source (§3.6):

Behavior	Verified
`poll()` is non-blocking	✓ Returns immediately
`poll()` called once handles all work	✓ Level-triggered
RxToken valid until consumed	✓ Lifetime-tied to device
TxToken consume may be called with any size ≤ MTU	✓
Millisecond timestamp resolution sufficient	✓

Audit Reference: §3.6, §4.3.1-4.3.3

6.2 The Device Trait Bridge

smoltcp requires implementing its `Device` trait. The device adapter wraps our VirtIO driver.

```
/// VirtIO-backed network device for smoltcp
pub struct VirtioNetDevice {
    /// VirtIO driver state
    driver: VirtioNet,
    /// TSC frequency (from boot handoff)
    tsc_freq: u64,
}

impl smoltcp::phy::Device for VirtioNetDevice {
    type RxToken<'a> = VirtioRxToken<'a>;
    type TxToken<'a> = VirtioTxToken<'a>;

    fn capabilities(&self) -> DeviceCapabilities {
        let mut caps = DeviceCapabilities::default();
        caps.max_transmission_unit = 1514;  // Ethernet MTU
        caps.medium = Medium::Ethernet;
        caps
    }

    fn receive(&mut self, _timestamp: Instant) ->
Option<(Self::RxToken<'_>, Self::TxToken<'_>)> {
        // Poll virtqueue for received packet
        match self.driver.poll_rx() {
            Some((idx, len)) => Some((
                VirtioRxToken { driver: &mut self.driver, idx, len },
                VirtioTxToken { driver: &mut self.driver },
            )),
            None => None,
        }
```

```
    }

    fn transmit(&mut self, _timestamp: Instant) ->
Option<Self::TxToken<'_>> {
        // Check if TX queue has space
        if self.driver.tx_queue_has_space() {
            Some(VirtioTxToken { driver: &mut self.driver })
        } else {
            None
        }
    }
}
```

6.3 Token Implementation

```
pub struct VirtioRxToken<'a> {
    driver: &'a mut VirtioNet,
    idx: u16,    // Buffer index
    len: usize,  // Received length (including 12-byte header)
}

impl<'a> smoltcp::phy::RxToken for VirtioRxToken<'a> {
    fn consume<R, F>(self, f: F) -> R
    where
        F: FnOnce(&mut [u8]) -> R,
    {
        // Get buffer, skip 12-byte VirtIO header
        let buffer = self.driver.get_rx_buffer(self.idx);
        let frame = &mut buffer[12..self.len];
        let result = f(frame);

        // Resubmit buffer to RX queue
        self.driver.submit_rx_buffer(self.idx);

        result
    }
}

pub struct VirtioTxToken<'a> {
    driver: &'a mut VirtioNet,
}

impl<'a> smoltcp::phy::TxToken for VirtioTxToken<'a> {
    fn consume<R, F>(self, len: usize, f: F) -> R
    where
        F: FnOnce(&mut [u8]) -> R,
    {
        // Allocate TX buffer from pool
        let idx = self.driver.alloc_tx_buffer();
        let buffer = self.driver.get_tx_buffer(idx);
```

```
        // Zero the 12-byte VirtIO header (no offloads)
        buffer[..12].fill(0);

        // Let smoltcp fill the frame
        let result = f(&mut buffer[12..12 + len]);

        // Submit to TX queue - NON-BLOCKING, fire-and-forget
        self.driver.submit_tx(idx, 12 + len);

        result
    }
}
```

6.4 The Critical Difference

WRONG (current implementation):

```
fn transmit(&mut self, packet: &[u8]) -> Result<()> {
    let token = self.inner.transmit_begin(&tx_buf)?;
    loop {  // ← BLOCKING LOOP - FORBIDDEN
        if let Some(t) = self.inner.poll_transmit() {
            if t == token { return Ok(()); }
        }
        tsc_delay_us(10);  // ← BUSY WAIT - FORBIDDEN
    }
}
```

CORRECT (new design):

```
fn transmit(&mut self, packet: &[u8]) -> Result<()> {
    // Submit and return immediately
    self.driver.submit_tx(idx, len);
    Ok(())
    // Completion collected in main loop's Phase 3
}
```

6.5 Timestamp Handling (Corrected)

smoltcp needs timestamps in milliseconds. TSC frequency is calibrated at boot.

```
/// Convert TSC ticks to smoltcp Instant
fn tsc_to_instant(tsc: u64, tsc_freq: u64) -> Instant {
    // tsc_freq is ticks per second
    // Convert to milliseconds: tsc * 1000 / tsc_freq
    let millis = tsc / (tsc_freq / 1000);
    Instant::from_millis(millis as i64)
}
```

```
// In main loop:
let now_tsc = rdtsc();
let timestamp = tsc_to_instant(now_tsc, handoff.tsc_freq);
iface.poll(timestamp, &mut device, &mut sockets);
```

Contract: `tsc_freq` comes from `BootHandoff`, calibrated at boot. Never hardcoded.

Audit Reference: §7.1.4, §2.5 TIME-2

6.6 smoltcp Memory Requirements

Per socket buffer allocation (must be done at initialization):

Resource	Size	Notes
TCP socket RX buffer	64KB	Per socket
TCP socket TX buffer	64KB	Per socket
Interface overhead	~10KB	ARP cache, etc.

Total for 3 TCP sockets: ~400KB

This is SEPARATE from the 2MB DMA region.

Audit Reference: §4.3.2

6.7 Section 6 Invariant Summary

Invariant	Description
SMOLTCP-1	poll() called exactly once per main loop iteration
SMOLTCP-2	Device receive() and transmit() return immediately
SMOLTCP-3	RxToken buffer valid until consume() returns
SMOLTCP-4	TxToken consume() submits packet without waiting
SMOLTCP-5	Timestamp derived from calibrated TSC, never hardcoded
SMOLTCP-6	Socket buffer memory allocated at initialization

Section 7: Timing, Polling Budgets & Determinism (Locked)

Audit Reference: §7.1.4 (TSC calibration), §2.5 (TIME-2), §5.3 (timing patterns)

7.1 The Polling Budget Model

Each main loop iteration has a **fixed time budget** (independent of CPU frequency):

```
┌─────────────────────────────────────────────────────────────┐
│                    MAIN LOOP TIME BUDGET                    │
│                  (Target: 1ms per iteration)                │
├──────────────────────────────┬──────────────────────────────┤
│ Phase                        │ Budget (wall time)           │
├──────────────────────────────┼──────────────────────────────┤
│ 1. RX Poll (16 checks)       │ ~20µs                        │
│ 2. smoltcp poll()            │ ~200µs                       │
│ 3. TX Drain (16 packets)     │ ~40µs                        │
│ 4. App state step            │ ~400µs                       │
│ 5. TX completion collect     │ ~20µs                        │
│ 6. Overhead/margin           │ ~320µs                       │
├──────────────────────────────┼──────────────────────────────┤
│ TOTAL                        │ 1ms                          │
└──────────────────────────────┴──────────────────────────────┘
```

Contract: Cycle counts are derived from `tsc_freq * time_budget_seconds`. Never hardcode cycle counts.

7.2 RX Polling Strategy

```rust
const RX_POLL_BUDGET: usize = 16;  // Max packets per iteration

fn poll_rx_phase(device: &mut AsmNetDevice, rx_queue: &mut RxQueue) {
    for _ in 0..RX_POLL_BUDGET {
        let mut buf = [0u8; 1514];
        let len = unsafe { asm_poll_rx(buf.as_mut_ptr(), buf.len()) };

        if len == 0 {
            break; // No more packets
        }

        rx_queue.push(RxPacket {
            buffer: buf,
            len: len as usize,
        });
    }
}
```

Key property: Loop bounded by `RX_POLL_BUDGET`, not by packet availability.

7.3 TX Drain Strategy

```rust
const TX_DRAIN_BUDGET: usize = 16;  // Max packets per iteration

fn drain_tx_phase(tx_queue: &mut TxQueue) {
    for _ in 0..TX_DRAIN_BUDGET {
        if let Some(pkt) = tx_queue.pop() {
```

```
            let result = unsafe {
                asm_poll_tx(pkt.buffer.as_ptr(), pkt.len)
            };

            if result != 0 {
                // Queue full, put packet back
                tx_queue.push_front(pkt);
                break;
            }
        } else {
            break; // Queue empty
        }
    }
}
```

7.4 Timeout Calculation (Parameterized)

All timeouts computed from calibrated `tsc_freq`. **No hardcoded cycle counts.**

```
/// Timeout configuration - all values in SECONDS or MILLISECONDS
/// Actual TSC ticks computed at runtime using tsc_freq
pub struct TimeoutConfig {
    pub tsc_freq: u64,  // From BootHandoff, calibrated at boot
}

impl TimeoutConfig {
    /// Create from calibrated TSC frequency
    pub fn new(tsc_freq: u64) -> Self {
        Self { tsc_freq }
    }

    /// Convert microseconds to TSC ticks
    #[inline]
    pub fn us_to_ticks(&self, us: u64) -> u64 {
        us * self.tsc_freq / 1_000_000
    }

    /// Convert milliseconds to TSC ticks
    #[inline]
    pub fn ms_to_ticks(&self, ms: u64) -> u64 {
        ms * self.tsc_freq / 1_000
    }

    /// Convert seconds to TSC ticks
    #[inline]
    pub fn sec_to_ticks(&self, sec: u64) -> u64 {
        sec * self.tsc_freq
    }

    // Protocol timeouts (in wall time, converted to ticks when needed)
    pub fn dhcp_discover(&self) -> u64 { self.sec_to_ticks(5) }
```

```
    pub fn dhcp_request(&self) -> u64 { self.sec_to_ticks(3) }
    pub fn tcp_connect(&self) -> u64 { self.sec_to_ticks(30) }
    pub fn tcp_keepalive(&self) -> u64 { self.sec_to_ticks(60) }
    pub fn dns_query(&self) -> u64 { self.sec_to_ticks(5) }
}
```

Contract: `TimeoutConfig` initialized once from `BootHandoff.tsc_freq`, passed to all timeout checks.

Audit Reference: §7.1.4, TIME-2 invariant

7.5 Timeout Checking Pattern

```
/// Check timeout without blocking
fn is_timed_out(start_tsc: u64, timeout_ticks: u64) -> bool {
    let now = unsafe { asm_read_tsc() };
    now.wrapping_sub(start_tsc) > timeout_ticks
}

// Usage in state machine (timeout_ticks from TimeoutConfig):
fn step(&mut self, timeouts: &TimeoutConfig) -> StepResult {
    match &mut self.state {
        State::Connecting { start_tsc, .. } => {
            if is_timed_out(*start_tsc, timeouts.tcp_connect()) {
                self.state = State::Failed(Error::Timeout);
                return StepResult::Done;
            }
            // Check connection status...
        }
        // ...
    }
}
```

7.6 TSC Calibration (Pre-ExitBootServices)

TSC calibration MUST occur before ExitBootServices using UEFI Stall():

```
/// Calibrate TSC using UEFI Stall() - MUST be called before
ExitBootServices
fn calibrate_tsc(boot_services: &BootServices) -> u64 {
    // Stall for 100ms (100,000 microseconds) - balance accuracy vs boot
time
    let start = unsafe { asm_read_tsc() };
    boot_services.stall(100_000); // 100ms
    let end = unsafe { asm_read_tsc() };

    // Scale to ticks per second
    (end - start) * 10
}
```

```
// Store in BootHandoff - NO DEFAULT VALUE
pub struct BootHandoff {
    pub tsc_freq: u64,  // REQUIRED, no default
    // ... other fields
}
```

Contract:

- Calibration MUST occur before ExitBootServices
- No default/fallback values - if calibration fails, boot fails
- Result stored in `BootHandoff.tsc_freq` and passed to all timing code

Audit Reference: §7.1.4 "calibrate_tsc() MUST be called before ExitBootServices"

7.7 Determinism Guarantees

Property	Guarantee	Mechanism
Bounded iteration time	< 2ms per loop	Fixed budgets
Bounded RX latency	< 1ms from wire to smoltcp	Priority RX phase
Bounded TX latency	< 2ms from smoltcp to wire	Immediate submission
No unbounded waits	All loops bounded	Budget constants
Predictable timing	±100µs variance	ASM critical paths

7.8 Anti-Patterns to Avoid

```
// ✗ WRONG: Unbounded loop
while !condition {
    do_work();
}

// ✓ CORRECT: Bounded check
for _ in 0..MAX_ITERATIONS {
    if condition { break; }
    do_work();
}
if !condition { return Err(Timeout); }

// ✗ WRONG: Busy wait
while time_elapsed < timeout {
    spin_loop();
}

// ✓ CORRECT: Check and return
if time_elapsed > timeout {
    return Err(Timeout);
}
// Continue with non-blocking work
```

```
// ✖ WRONG: Blocking inside callback
fn on_tx_submit(&mut self) {
    while !self.tx_complete() { } // BLOCKS
}

// ✔ CORRECT: State machine
fn step(&mut self) -> StepResult {
    match self.state {
        TxPending { .. } => {
            if self.tx_complete() {
                self.state = TxDone;
            }
            StepResult::Pending
        }
        // ...
    }
}
```

7.9 Section 7 Invariant Summary

Invariant	Description
TIME-1	TSC calibrated at boot via UEFI Stall(), not hardcoded
TIME-2	No hardcoded TSC frequency (no `2_500_000`, no "@ 2.5GHz")
TIME-3	All timeouts expressed in wall time, converted via `TimeoutConfig`
TIME-4	`wrapping_sub()` used for TSC comparisons (handles wraparound)
BUDGET-1	Each loop phase has bounded iteration count (e.g., `RX_POLL_BUDGET: 16`)
BUDGET-2	Total main loop iteration target: 1ms wall time
BUDGET-3	No unbounded loops (`while condition {}` forbidden)
DETER-1	Timing variance ±100µs via ASM critical paths

Section 8: Protocol State Machines (Locked)

Audit Reference: §2.7 (state machine patterns), §5.6 (protocol state machines)

8.1 The State Machine Principle

Every multi-step operation is a **state machine**, not a loop:

```
WRONG: Loop-based thinking
┌─────────────────────────────────────────────┐
│ fn do_http_request() {                      │
│     connect();     // blocks                │
│     send();        // blocks                │
```

```
|     recv();        // blocks            |
|     return response;                     |
| }                                        |
└──────────────────────────────────────────┘

CORRECT: State machine thinking
┌──────────────────────────────────────────┐
| enum HttpState {                         |
|     Resolving,                           |
|     Connecting,                          |
|     SendingHeaders,                      |
|     SendingBody,                         |
|     ReceivingHeaders,                    |
|     ReceivingBody,                       |
|     Done(Response),                      |
|     Failed(Error),                       |
| }                                        |
|                                          |
| fn step() -> bool { /* one step */ }     |
└──────────────────────────────────────────┘
```

8.2 HTTP Client State Machine

```rust
pub enum HttpState {
    /// Initial state, need to resolve hostname
    Idle,

    /// DNS query in flight
    Resolving {
        host: String,
        query_handle: QueryHandle,
        start_tsc: u64,
    },

    /// TCP connection in progress
    Connecting {
        ip: Ipv4Addr,
        port: u16,
        socket: SocketHandle,
        start_tsc: u64,
    },

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

    /// Sending request body (POST/PUT)
```

```
    SendingBody {
        socket: SocketHandle,
        body: 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,
        body: Vec<u8>,
        content_length: Option<usize>,
        start_tsc: u64,
    },

    /// Request complete
    Done(Response),

    /// Request failed
    Failed(HttpError),
}

impl HttpState {
    /// Advance state machine by one step.
    /// Returns true if reached terminal state.
    /// `timeouts` initialized from calibrated TSC frequency (no hardcoded
values).
    pub fn step(
        &mut self,
        iface: &mut NetInterface,
        now_tsc: u64,
        timeouts: &TimeoutConfig,
    ) -> bool {
        match self {
            HttpState::Idle => false,

            HttpState::Resolving { host, query_handle, start_tsc } => {
                // Check timeout using calibrated value
                if now_tsc - *start_tsc > timeouts.dns_query() {
                    *self = HttpState::Failed(HttpError::DnsTimeout);
                    return true;
                }

                // Poll DNS
                match iface.get_dns_result(*query_handle) {
                    Ok(Some(ip)) => {
```

```
                        *self = HttpState::Connecting {
                            ip,
                            port: 80,
                            socket: iface.tcp_socket().unwrap(),
                            start_tsc: now_tsc,
                        };
                    }
                    Ok(None) => {} // Still resolving
                    Err(e) => {
                        *self = HttpState::Failed(HttpError::DnsFailed);
                        return true;
                    }
                }
                false
            }

            HttpState::Connecting { ip, port, socket, start_tsc } => {
                // Check timeout using calibrated value
                if now_tsc - *start_tsc > timeouts.tcp_connect() {
                    *self = HttpState::Failed(HttpError::ConnectTimeout);
                    return true;
                }

                // Check connection state
                if iface.tcp_is_connected(*socket) {
                    // Prepare headers
                    let headers = format!("GET / HTTP/1.1\r\nHost:
{}\r\n\r\n", ip);
                    *self = HttpState::SendingHeaders {
                        socket: *socket,
                        headers: headers.into_bytes(),
                        sent: 0,
                        start_tsc: now_tsc,
                    };
                }
                false
            }

            // ... other states follow same pattern

            HttpState::Done(_) | HttpState::Failed(_) => true,
        }
    }
}
```

8.3 TCP Connection State Machine

```
pub enum TcpConnState {
    /// Not connected
    Closed,
```

```
    /// SYN sent, waiting for SYN-ACK
    SynSent {
        socket: SocketHandle,
        start_tsc: u64,
    },

    /// Connection established
    Established {
        socket: SocketHandle,
    },

    /// FIN sent, waiting for FIN-ACK
    FinWait {
        socket: SocketHandle,
        start_tsc: u64,
    },

    /// Error state
    Error(TcpError),
}
```

8.4 DHCP State Machine

```
pub enum DhcpState {
    /// Not started
    Idle,

    /// DHCPDISCOVER sent, waiting for DHCPOFFER
    Discovering {
        start_tsc: u64,
        retries: u8,
    },

    /// DHCPREQUEST sent, waiting for DHCPACK
    Requesting {
        offered_ip: Ipv4Addr,
        server_ip: Ipv4Addr,
        start_tsc: u64,
    },

    /// Lease obtained
    Bound {
        ip: Ipv4Addr,
        subnet: Ipv4Addr,
        gateway: Option<Ipv4Addr>,
        dns: Option<Ipv4Addr>,
        lease_start_tsc: u64,
        lease_duration_tsc: u64,
    },

    /// Lease renewal in progress
```

```
    Renewing {
        current_ip: Ipv4Addr,
        start_tsc: u64,
    },

    /// Failed to obtain lease
    Failed(DhcpError),
}
```

8.5 Composing State Machines

Higher-level operations compose lower-level state machines:

```
pub enum IsoDownloadState {
    /// Starting up
    Init,

    /// Waiting for network (DHCP)
    WaitingForNetwork {
        dhcp: DhcpState,
    },

    /// Resolving mirror hostname
    ResolvingMirror {
        dns: DnsState,
    },

    /// Downloading ISO
    Downloading {
        http: HttpState,
        bytes_received: usize,
        total_size: Option<usize>,
    },

    /// Verifying checksum
    Verifying {
        hasher: Sha256State,
    },

    /// Complete
    Done {
        iso_ptr: *const u8,
        iso_len: usize,
    },

    /// Failed
    Failed(DownloadError),
}

impl IsoDownloadState {
    pub fn step(&mut self, iface: &mut NetInterface, now_tsc: u64) -> bool
```

```
{
        match self {
            IsoDownloadState::WaitingForNetwork { dhcp } => {
                if dhcp.step(iface, now_tsc) {
                    match dhcp {
                        DhcpState::Bound { .. } => {
                            *self = IsoDownloadState::ResolvingMirror {
                                dns: DnsState::new("mirror.example.com"),
                            };
                        }
                        DhcpState::Failed(e) => {
                            *self = IsoDownloadState::Failed(
                                DownloadError::NetworkFailed
                            );
                            return true;
                        }
                        _ => {}
                    }
                }
                false
            }
            // ... other states
            _ => false,
        }
    }
}
```

8.6 State Machine Testing

Each state machine can be tested in isolation:

```
#[test]
fn test_dhcp_discovery_timeout() {
    // Create timeout config with known calibrated value for testing
    let timeouts = TimeoutConfig::new(2_500_000_000); // Test with 2.5GHz

    let mut state = DhcpState::Discovering {
        start_tsc: 0,
        retries: 0,
    };

    // Simulate time passing without DHCP response
    let timeout_tsc = timeouts.dhcp_discover() + 1;
    let done = state.step(&mut mock_iface(), timeout_tsc, &timeouts);

    assert!(done);
    assert!(matches!(state, DhcpState::Failed(DhcpError::Timeout)));
}
```

8.7 Section 8 Invariant Summary

Invariant	Description
SM-1	Every multi-step operation is a state machine, not a blocking loop
SM-2	`step()` advances exactly one step and returns immediately
SM-3	`step()` returns `true` only on terminal states (Done, Failed)
SM-4	Each state carries its own `start_tsc` for timeout tracking
SM-5	Timeouts passed via `TimeoutConfig`, never hardcoded
SM-6	Higher-level machines compose lower-level machines
SM-7	State machines testable in isolation with mock interfaces

---

Section 9: Risks, Limitations & Mitigations

9.1 Known Risks

Risk	Severity	Likelihood	Mitigation
TSC frequency varies across CPUs	Medium	High	Calibrate at boot, store in handoff
QEMU VirtIO timing differs from real HW	Low	High	Test on both, use conservative timeouts
DMA region too small	High	Medium	Calculate needs upfront, reserve 2MB+
smoltcp bugs in edge cases	Medium	Low	Extensive testing, fallback paths
ASM bugs hard to debug	High	Medium	Extensive comments, unit tests
Real NICs differ from VirtIO	High	Certain	Abstract via trait, test each driver

9.2 Architectural Limitations

Limitation	Impact	Future Fix
Single-core only	Can't parallelize RX/TX	Phase 2: multi-core
No interrupt support	Higher latency, more CPU	Phase 3: MSI-X support
Polling-only model	CPU-intensive	Multi-core offload
VirtIO-only initially	Limited to VMs	Add Intel/Realtek drivers
No TLS/HTTPS	Insecure downloads	Add TLS state machine
Fixed buffer sizes	Memory waste	Dynamic allocation

9.3 Performance Limitations

Metric	Expected	Limitation Cause

Metric	Expected	Limitation Cause
Latency	1-2ms	Polling budget
Throughput	~500 Mbps	Single-core, no batching
CPU usage	50-80%	Continuous polling
Memory	~2MB DMA	Static allocation

9.4 Edge Cases & Failure Modes

Network Failures:

- DHCP server unreachable → Timeout, retry, fail gracefully
- DNS failure → Use hardcoded fallbacks
- TCP RST → Close socket, report error
- Packet corruption → Rely on TCP checksums

Hardware Failures:

- NIC not responding → Timeout, report error, halt
- DMA failure → Fatal, no recovery
- PCI access failure → Fatal, no recovery

Resource Exhaustion:

- RX queue full → Drop oldest packets
- TX queue full → Backpressure to smoltcp
- Memory exhausted → Fatal (pre-allocated)

9.5 Security Considerations

Threat	Mitigation
Malformed packets	smoltcp validation
DMA attacks	Identity mapping, no IOMMU
Buffer overflow	Bounds checking in ASM
Timing attacks	Deterministic execution

Note: This design does NOT protect against malicious hypervisor or hardware.

Section 10: Future Expansion Paths

10.1 Phase 2: Multi-Core Support

```
┌─────────────────────────────────────────────────────────────┐
│                    MULTI-CORE MODEL                         │
├─────────────────────────────────────────────────────────────┤
```

```
┌─────────────────────────────────────────────────────────────────────┐
│                                                                     │
│   Core 0 (BSP)                Core 1 (AP)                           │
│   ┌──────────────┐            ┌──────────────┐                      │
│   │ Main Loop    │            │ NIC Driver   │                      │
│   │ - smoltcp    │◄──────────►│ - RX poll    │                      │
│   │ - App state  │  IPC       │ - TX submit  │                      │
│   │ - Timeouts   │  Ring      │ - Completn   │                      │
│   └──────────────┘            └──────────────┘                      │
│                                                                     │
│   Benefits:                                                         │
│   - Core 0 never blocked by NIC                                     │
│   - Core 1 can poll continuously                                    │
│   - Lower latency, higher throughput                                │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```

10.2 Phase 3: Interrupt Support

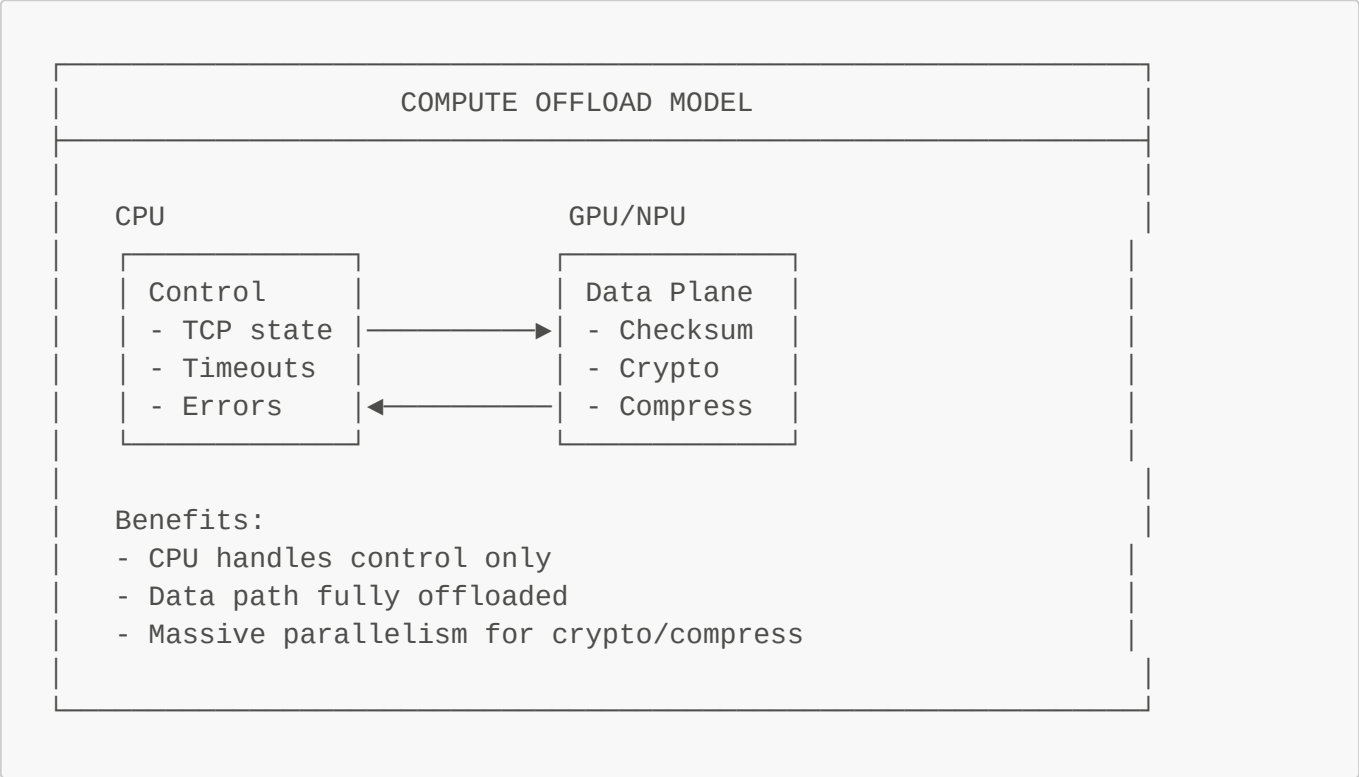
```rust
// Future: MSI-X interrupt handler
#[naked]
unsafe extern "C" fn nic_interrupt_handler() {
    asm!(
        "push rax",
        "push rcx",
        "push rdx",
        // Acknowledge interrupt
        "call asm_nic_ack_interrupt",
        // Signal main loop (set flag)
        "mov byte ptr [INTERRUPT_PENDING], 1",
        // EOI
        "mov al, 0x20",
        "out 0x20, al",
        "pop rdx",
        "pop rcx",
        "pop rax",
        "iretq",
        options(noreturn)
    );
}
```

10.3 Phase 4: Additional NIC Drivers

Driver	Priority	Complexity	Notes
Intel e1000	High	Medium	Common in QEMU
Intel e1000e	High	Medium	Modern Intel
Intel i219	Medium	Medium	Recent chipsets

Driver	Priority	Complexity	Notes
Realtek 8169	High	Low	Consumer boards
Realtek 8111	High	Medium	Modern consumer
Broadcom TG3	Low	High	Servers

10.4 Phase 5: GPU/NPU Offload



Section 11: Implementation Estimate

11.1 LOC Breakdown

Component	Estimated LOC	Language
ASM NIC driver (VirtIO)	400-500	x86_64 ASM
ASM HAL primitives	100-150	x86_64 ASM
Device adapter (Rust)	200-250	Rust
State machines (Rust)	600-800	Rust
Main loop (Rust)	150-200	Rust
Boot handoff (Rust)	100-150	Rust
Tests	300-400	Rust
Total	**~2000**	Mixed

11.2 Implementation Order

1. **Week 1**: ASM primitives (`asm_read_tsc`, `asm_poll_rx`, `asm_poll_tx`)
2. **Week 2**: VirtIO virtqueue setup in ASM
3. **Week 3**: Device adapter, smoltcp integration
4. **Week 4**: State machines (DHCP, TCP)
5. **Week 5**: HTTP state machine, testing
6. **Week 6**: Boot handoff, integration
7. **Week 7-8**: Testing, debugging, hardening

11.3 Testing Strategy

Test Type	Coverage	Tools
Unit tests	State machines	`cargo test`
Integration	Full stack	QEMU + VirtIO
Hardware	Real NICs	Physical machines
Stress	Edge cases	Fuzzing, packet loss sim

Section 12: Conclusion

12.1 Summary

This design addresses the fundamental deadlock in MorpheusX's network stack by:

1. **Eliminating blocking patterns** through state machines
2. **Separating ASM and Rust responsibilities** for determinism
3. **Respecting the ExitBootServices boundary** for UEFI compatibility
4. **Using fixed polling budgets** for predictable timing
5. **Designing for future expansion** to multi-core and interrupts

12.2 Consolidated Invariant Index

All invariants from this document, organized by category:

Memory & DMA (Section 2, 5)

ID	Invariant
MEM-1	x86 cache coherent for DMA; UC/WC mapping or clflush required only if device-specific
MEM-2	DMA memory allocated via PCI I/O Protocol AllocateBuffer, not raw AllocatePages
MEM-3	bus_addr from Map() passed to device, not CPU pointer
MEM-4	CPU pointer used for read/write, bus_addr for device descriptors
DMA-1	DMA region identity-mapped, accessible by both CPU and device
DMA-2	Minimum 2MB DMA reservation: 1MB RX + 1MB TX
DMA-3	Buffer ownership explicit: driver owns until returned to available ring

Timing (Section 7)

ID	Invariant
TIME-1	TSC calibrated at boot via UEFI Stall(), not hardcoded
TIME-2	No hardcoded TSC frequency anywhere in codebase
TIME-3	All timeouts expressed in wall time, converted via TimeoutConfig
TIME-4	wrapping_sub() used for TSC comparisons

Execution Model (Section 3)

ID	Invariant
EXEC-1	Single core, single thread, no preemption
EXEC-2	Main loop is ONLY entry point for all network activity
EXEC-3	Phase order: RX poll → smoltcp poll → TX drain → app step → TX completion
EXEC-4	No interrupts; purely poll-driven

VirtIO Driver (Section 4)

ID	Invariant
VIRTIO-1	12-byte header (flags, gso_type, hdr_len, gso_size, csum_start, csum_offset)
VIRTIO-2	Feature negotiation completes before ExitBootServices
VIRTIO-3	Queue size read from device via VIRTIO_PCI_QUEUE_NUM
VIRTIO-4	Descriptor chains: buffer addr + len + flags + next
VIRTIO-5	Memory barriers: sfence before queue notify, lfence after reading used ring
VIRTIO-6	Device notification via single 16-bit port write

smoltcp Integration (Section 6)

ID	Invariant
SMOLTCP-1	poll() called exactly once per main loop iteration
SMOLTCP-2	Device receive() and transmit() return immediately
SMOLTCP-3	RxToken buffer valid until consume() returns
SMOLTCP-4	TxToken consume() submits packet without waiting
SMOLTCP-5	Timestamp derived from calibrated TSC
SMOLTCP-6	Socket buffer memory allocated at initialization

State Machines (Section 8)

ID	Invariant
SM-1	Every multi-step operation is a state machine
SM-2	step() advances exactly one step and returns immediately
SM-3	step() returns true only on terminal states
SM-4	Each state carries its own start_tsc for timeout tracking
SM-5	Timeouts passed via TimeoutConfig

Budgets (Section 7)

ID	Invariant
BUDGET-1	Each loop phase has bounded iteration count
BUDGET-2	Total main loop iteration target: 1ms wall time
BUDGET-3	No unbounded loops

12.3 Success Criteria

- ☐ DHCP completes within 10 seconds
- ☐ TCP connection establishes within 5 seconds
- ☐ HTTP request completes without blocking
- ☐ Main loop iteration < 2ms guaranteed
- ☐ No `tsc_delay_us()` calls remain in codebase
- ☐ All blocking loops replaced with state machines
- ☐ No hardcoded TSC values (grep for `2_500`, `2500000`)
- ☐ All DMA allocation via PCI I/O Protocol

12.4 Document Status

This document represents the **frozen v1 architecture** for MorpheusX network stack.

Changes to this document require:

1. Audit against NETWORK_STACK_AUDIT.md
2. Explicit invariant impact analysis
3. Version increment

Document End

Status: v1 Architecture (Frozen) Reconciled Against: NETWORK_STACK_AUDIT.md Last Reconciliation: 2024