# MorpheusX ASM ↔ Rust ABI Contract v1.0

**Status**: FROZEN
**Date**: 2026-01-09
**Authority**: Derived from NETWORK_STACK_REDESIGN.md (v1.0 Frozen) and NETWORK_STACK_AUDIT.md (v1.1)
**Scope**: Post-ExitBootServices bare-metal networking runtime
**Target**: x86_64-unknown-uefi

---

## Document Structure

This contract is divided into mandatory chunks. Each chunk is self-contained and must be read in order. No implicit assumptions carry between chunks unless explicitly stated.

---

# Chunk 1 — Contract Scope & Global Invariants

## 1.1 Purpose of This Contract

This document defines the **frozen v1 ABI** between assembly (ASM) code and Rust code for the MorpheusX post-ExitBootServices networking runtime.

**This contract specifies:**

- What ASM exposes to Rust
- What Rust may assume about ASM behavior
- What ASM may assume about Rust behavior
- What constitutes undefined behavior (UB)
- Memory ownership and lifetime rules
- Calling conventions and register usage

**This contract does NOT specify:**

- Internal ASM implementation details
- Internal Rust implementation details
- Anything outside the ASM ↔ Rust boundary

**Binding Authority:**

- NETWORK_STACK_REDESIGN.md §Execution Model (Locked) — execution constraints
- NETWORK_STACK_AUDIT.md Part 7 — authoritative corrections
- Where conflicts exist, AUDIT.md Part 7 supersedes REDESIGN.md

## 1.2 System Context (Non-Negotiable)

The following system context is immutable for v1. Any code violating these constraints exhibits undefined behavior.

| Property | Value | Source |
|---|---|---|
| Architecture | x86_64 | REDESIGN §2.1 |
| Execution mode | Long mode (64-bit) | AUDIT §7.1.1 |
| ABI | Microsoft x64 (UEFI-compatible) | AUDIT §7.1.6 |
| Runtime phase | Post-ExitBootServices | REDESIGN §5.1, AUDIT §7.2.9 |
| Interrupt state | Disabled (IF=0) | REDESIGN §2.1 |
| Core count | 1 (BSP only) | REDESIGN §Execution Model |
| Thread count | 1 | REDESIGN §Execution Model |
| Scheduling | None (cooperative progress) | REDESIGN §Execution Model |
| Execution pattern | Poll-driven | REDESIGN §Execution Model |
| Preemption | None | REDESIGN §Execution Model |
| Floating point | Disabled | REDESIGN §2.1 |
| Async runtime | None | REDESIGN §Explicit Non-Goals |
| UEFI Boot Services | UNAVAILABLE | REDESIGN §5.5 |

## 1.3 Trust Boundaries

### 1.3.1 ASM Owns

| Domain | Rationale | Source |
|---|---|---|
| All hardware interaction | Direct MMIO, port I/O | REDESIGN §3.1 |
| DMA buffer submission | Descriptor ring manipulation | REDESIGN §4.5 |
| NIC device state | Register reads/writes | REDESIGN §4.3 |
| Memory barriers | sfence/lfence/mfence placement | AUDIT §5.4 |
| TSC reads | rdtsc instruction | REDESIGN §7.4 |
| Virtqueue ring updates | Available/used ring manipulation | REDESIGN §4.5 |
| Device notification | MMIO notify writes | REDESIGN §4.5 |

### 1.3.2 Rust Owns

| Domain | Rationale | Source |
|---|---|---|
| Protocol logic | smoltcp integration | REDESIGN §6.1 |
| State machines | DHCP, TCP, HTTP states | REDESIGN §8.1 |
| Application orchestration | Main loop control | REDESIGN §3.2 |

| Domain | Rationale | Source |
|---|---|---|
| Buffer content preparation | Packet payload | REDESIGN §6.3 |
| Timeout calculations | TSC delta interpretation | REDESIGN §7.4 |
| Error handling policy | Retry vs. fatal decisions | REDESIGN §9.4 |

### 1.3.3 Boundary Crossings

The ASM ↔ Rust boundary is crossed ONLY through the functions defined in Chunk 3 (Canonical ASM Interface Table). Any other mechanism for crossing this boundary is undefined behavior.

**Permitted crossing mechanisms:**

- `extern "win64" fn` declarations in Rust calling ASM symbols
- ASM calling Rust functions via the `call` instruction with MS x64 ABI

**Forbidden crossing mechanisms:**

- Inline assembly modifying ASM-owned state
- Rust directly accessing MMIO regions
- Rust directly manipulating virtqueue rings
- Any form of shared mutable state without explicit contract

## 1.4 Global Invariants

These invariants are derived from REDESIGN.md and AUDIT.md and must hold at all times. Violation constitutes undefined behavior.

### 1.4.1 Execution Invariants

| ID | Invariant | Source |
|---|---|---|
| **EXEC-INV-1** | All function calls return in bounded time | REDESIGN §2.2 |
| **EXEC-INV-2** | No function may yield, await, or suspend | REDESIGN §Execution Model |
| **EXEC-INV-3** | No function may loop waiting for external state | REDESIGN §1.5 BLOCK-1 |
| **EXEC-INV-4** | Interrupts remain disabled (IF=0) | REDESIGN §2.1 |
| **EXEC-INV-5** | Single core execution (no SMP) | REDESIGN §Execution Model |

### 1.4.2 Memory Invariants

| ID | Invariant | Source |
|---|---|---|
| **MEM-INV-1** | DMA region is identity-mapped (phys == virt) | AUDIT §7.1.1 |
| **MEM-INV-2** | DMA region is UC or WC (not WB) | AUDIT §7.2.2 |
| **MEM-INV-3** | bus_addr from Map() used for device, cpu_ptr for software | AUDIT §7.2.1 |

| ID | Invariant | Source |
|---|---|---|
| **MEM-INV-4** | Buffers have exactly one owner at any time | REDESIGN §3.3 |
| **MEM-INV-5** | DEVICE-OWNED buffers are never accessed by driver | REDESIGN §3.3 |
| **MEM-INV-6** | DMA region below 4GB (32-bit compatibility) | REDESIGN §4.7 |

### 1.4.3 Timing Invariants

| ID | Invariant | Source |
|---|---|---|
| **TIME-INV-1** | TSC frequency calibrated at boot, never hardcoded | REDESIGN §7.6, AUDIT §7.1.4 |
| **TIME-INV-2** | Invariant TSC verified via CPUID before use | REDESIGN §5.4, AUDIT §7.1.4 |
| **TIME-INV-3** | TSC monotonic on single core | AUDIT §7.1.5 |
| **TIME-INV-4** | wrapping_sub() for TSC comparisons | REDESIGN §7.5 |

### 1.4.4 VirtIO Invariants

| ID | Invariant | Source |
|---|---|---|
| **VIO-INV-1** | Header size is 12 bytes (VERSION_1 devices) | AUDIT §7.2.7 |
| **VIO-INV-2** | Feature negotiation complete before queue setup | AUDIT §7.2.4 |
| **VIO-INV-3** | Queue size read from device, not hardcoded | AUDIT §7.2.5 |
| **VIO-INV-4** | Device initialization only after ExitBootServices | AUDIT §7.2.9 |
| **VIO-INV-5** | Descriptor table entries: 16 bytes each | VirtIO Spec 1.2 §2.6 |
| **VIO-INV-6** | avail.idx increments monotonically (mod 2^16) | VirtIO Spec 1.2 §2.7 |

### 1.4.5 Barrier Invariants

| ID | Invariant | Source |
|---|---|---|
| **BAR-INV-1** | sfence before avail.idx write | AUDIT §5.4.1 |
| **BAR-INV-2** | mfence before device notification | AUDIT §5.4.1 |
| **BAR-INV-3** | lfence after reading used.idx (or volatile) | AUDIT §5.4.2 |
| **BAR-INV-4** | cli/sti are NOT memory barriers | AUDIT §7.2.11 |

## 1.5 Explicit Non-Goals (Contract Scope Exclusions)

The following are explicitly excluded from this contract. Any assumption that they are supported is incorrect.

| Non-Goal | Rationale | Source |
|---|---|---|
| Interrupt support | Polling model | REDESIGN §Explicit Non-Goals |

| Non-Goal | Rationale | Source |
|---|---|---|
| Multi-core / SMP | Single-core design | REDESIGN §Explicit Non-Goals |
| Async runtime | No futures/executor | REDESIGN §Explicit Non-Goals |
| TLS / HTTPS | No crypto | REDESIGN §Explicit Non-Goals |
| IPv6 | IPv4 only | REDESIGN §Explicit Non-Goals |
| Jumbo frames | 1500 MTU only | REDESIGN §Explicit Non-Goals |
| Hardware offloads | Software checksums | REDESIGN §Explicit Non-Goals |
| UEFI Boot Services | Post-EBS only | REDESIGN §5.5 |
| Runtime memory allocation | Pre-allocated pools only | REDESIGN §5.3 |
| Context switching | Single execution context | REDESIGN §Execution Model |

## 1.6 Contract Versioning

| Field | Value |
|---|---|
| Contract Version | 1.0 |
| Compatible REDESIGN.md | v1.0 (Frozen) |
| Compatible AUDIT.md | v1.1 |
| Breaking Change Policy | New major version required |

**Version Compatibility Rule:** Code compiled against this contract version MUST NOT be linked with code compiled against a different major version without explicit compatibility verification.

## 1.7 Terminology

| Term | Definition |
|---|---|
| ASM | Assembly code (x86_64 NASM/GAS syntax) |
| Rust | Rust code compiled for x86_64-unknown-uefi |
| UB | Undefined Behavior |
| EBS | ExitBootServices |
| DMA | Direct Memory Access |
| TSC | Time Stamp Counter |
| UC | Uncached memory type |
| WC | Write-Combining memory type |
| Virtqueue | VirtIO ring buffer structure |
| MMIO | Memory-Mapped I/O |

| Term | Definition |
|---|---|
| BSP | Bootstrap Processor (core 0) |

# Chunk 2 — Calling Convention & ABI Rules

## 2.1 Calling Convention: Microsoft x64 ABI

All ASM ↔ Rust boundary crossings use the **Microsoft x64 calling convention** (also known as "Win64 ABI"). This is mandated by UEFI and must be used consistently.

**Source:** AUDIT §7.1.6 — "On x86-64, UEFI uses the Microsoft 'Win64' calling convention"

### 2.1.1 Parameter Passing (Integer/Pointer)

| Parameter | Register | Notes |
|---|---|---|
| 1st | RCX | First integer/pointer argument |
| 2nd | RDX | Second integer/pointer argument |
| 3rd | R8 | Third integer/pointer argument |
| 4th | R9 | Fourth integer/pointer argument |
| 5th+ | Stack | Right-to-left, 8-byte aligned |

**Stack layout for 5th+ parameters:**

```
[RSP + 0x28]  5th parameter
[RSP + 0x30]  6th parameter
...
```

### 2.1.2 Return Values

| Type | Location | Notes |
|---|---|---|
| Integer ≤ 64 bits | RAX | Includes pointers, enums |
| Integer 128 bits | RAX:RDX | Low 64 in RAX, high 64 in RDX |
| Struct ≤ 8 bytes | RAX | Returned by value |
| Struct > 8 bytes | Hidden pointer | Caller provides buffer in RCX |

### 2.1.3 Shadow Space Requirement

The caller MUST allocate 32 bytes of shadow space on the stack before any `call` instruction, even if fewer than 4 parameters are passed.

```
; CORRECT: Allocating shadow space before call
sub rsp, 32          ; Shadow space (minimum)
mov rcx, param1
mov rdx, param2
call target_function
add rsp, 32          ; Restore stack
```

**Violation:** Calling without shadow space is undefined behavior.

### 2.1.4 Stack Alignment

The stack MUST be 16-byte aligned **before** the `call` instruction. The `call` itself pushes 8 bytes (return address), so at function entry RSP mod 16 == 8.

```
Before call:     RSP mod 16 == 0   (aligned)
After call:      RSP mod 16 == 8   (misaligned by return address)
After prologue:  RSP mod 16 == 0   (realigned by push/sub)
```

**Violation:** Misaligned stack causes undefined behavior (potential crashes on SSE instructions, though we don't use SSE).

## 2.2 Register Classification

### 2.2.1 Volatile Registers (Caller-Saved)

These registers may be modified by any function call. The caller must save them if their values are needed after the call.

| Register | Purpose in MS x64 |
|---|---|
| RAX | Return value, scratch |
| RCX | 1st parameter, scratch |
| RDX | 2nd parameter, scratch |
| R8 | 3rd parameter, scratch |
| R9 | 4th parameter, scratch |
| R10 | Scratch |
| R11 | Scratch |
| RFLAGS | Volatile (except DF) |
| XMM0-XMM5 | Floating point (unused in this contract) |

### 2.2.2 Non-Volatile Registers (Callee-Saved)

These registers must be preserved across function calls. If a function modifies them, it must save and restore them.

| Register | Notes |
|---|---|
| RBX | Must be preserved |
| RBP | Frame pointer (optional), must be preserved |
| RDI | Must be preserved |
| RSI | Must be preserved |
| RSP | Stack pointer, must be preserved |
| R12 | Must be preserved |
| R13 | Must be preserved |
| R14 | Must be preserved |
| R15 | Must be preserved |
| XMM6-XMM15 | Must be preserved (unused in this contract) |

### 2.2.3 Special Registers

| Register | Constraint |
|---|---|
| DF (Direction Flag) | Must be 0 (clear) at call/return |
| IF (Interrupt Flag) | Must remain 0 (interrupts disabled) |
| RSP | 16-byte aligned before call |

## 2.3 Red Zone: NOT AVAILABLE

**Critical:** The Microsoft x64 ABI does **not** have a red zone.

Unlike the System V AMD64 ABI (which provides 128 bytes below RSP that are safe from signals), the MS x64 ABI provides no such guarantee.

**Implication for ASM:**

- Do NOT store data below RSP without first decrementing RSP
- All local variables must be in properly allocated stack space
- Interrupt handlers (if ever added) may clobber memory below RSP

```
; WRONG: Using non-existent red zone
mov [rsp - 8], rax   ; UNSAFE - no red zone in MS ABI

; CORRECT: Properly allocate stack space
sub rsp, 8
mov [rsp], rax
```

## 2.4 Clobber Rules for ASM Functions

### 2.4.1 ASM Function Prologue Requirements

Every ASM function callable from Rust MUST:

1. Preserve all non-volatile registers it uses
2. Maintain 16-byte stack alignment
3. Allocate shadow space before calling other functions

```
; Standard ASM function prologue
my_function:
    push rbx            ; Save non-volatile (if used)
    push r12            ; Save non-volatile (if used)
    sub rsp, 40         ; 32 shadow + 8 alignment
    ; ... function body ...
    add rsp, 40
    pop r12
    pop rbx
    ret
```

### 2.4.2 ASM Function Epilogue Requirements

Before returning, ASM functions MUST:

1. Place return value in RAX (or RAX:RDX)
2. Restore all non-volatile registers
3. Restore RSP to entry value (minus 8 for return address)
4. Clear DF if it was modified

### 2.4.3 Declared Clobber Sets

For Rust `asm!` blocks calling ASM, the following clobber sets apply:

```
// Standard ASM call clobber declaration
unsafe {
    asm!(
        "call {target}",
        target = sym asm_function_name,
        // Clobbers (all volatile registers)
        out("rax") result,
        out("rcx") _,
        out("rdx") _,
        out("r8") _,
        out("r9") _,
        out("r10") _,
        out("r11") _,
        // RBX, RBP, RDI, RSI, RSP, R12-R15 preserved
```

```
    );
}
```

## 2.5 Floating Point and SIMD: DISABLED

Per REDESIGN §2.1: Floating point is disabled. This contract forbids:

- Use of x87 FPU instructions
- Use of SSE/AVX registers (XMM, YMM, ZMM)
- Use of any floating-point types across the boundary

**Rationale:** No FPU state save/restore; keeps context minimal.

**Violation:** Any floating-point operation across the ASM ↔ Rust boundary is undefined behavior.

## 2.6 Rust Declaration Syntax

Rust code declaring ASM functions must use `extern "win64"`:

```rust
extern "win64" {
    /// Read TSC counter
    fn asm_read_tsc() -> u64;

    /// Submit buffer to TX queue
    /// Returns: 0 on success, non-zero error code on failure
    fn asm_tx_submit(
        virtqueue_state: *mut VirtqueueState,  // RCX
        buffer_idx: u16,                       // RDX (zero-extended)
        length: u16,                           // R8 (zero-extended)
    ) -> i32;
}
```

**Note:** On `x86_64-unknown-uefi` target, `extern "C"` is equivalent to `extern "win64"`. For clarity, this contract mandates explicit `extern "win64"`.

## 2.7 Parameter Type Mappings

### 2.7.1 Integer Types

| Rust Type | Size | Register Usage | Notes |
|---|---|---|---|
| `u8` | 1 byte | Low 8 bits of register | Zero-extended |
| `u16` | 2 bytes | Low 16 bits of register | Zero-extended |
| `u32` | 4 bytes | Low 32 bits of register | Zero-extended |
| `u64` | 8 bytes | Full register | — |
| `usize` | 8 bytes | Full register | Same as u64 |

| Rust Type | Size | Register Usage | Notes |
|---|---|---|---|
| `i8` | 1 byte | Low 8 bits | Sign-extended |
| `i16` | 2 bytes | Low 16 bits | Sign-extended |
| `i32` | 4 bytes | Low 32 bits | Sign-extended |
| `i64` | 8 bytes | Full register | — |
| `isize` | 8 bytes | Full register | Same as i64 |
| `bool` | 1 byte | Low 8 bits | 0 or 1 only |

### 2.7.2 Pointer Types

| Rust Type | Size | Notes |
|---|---|---|
| `*const T` | 8 bytes | Raw pointer, passed as u64 |
| `*mut T` | 8 bytes | Raw pointer, passed as u64 |
| `&T` | 8 bytes | Reference, passed as pointer |
| `&mut T` | 8 bytes | Reference, passed as pointer |

**Pointer validity:** The caller guarantees pointer validity. Null pointers are undefined behavior unless explicitly documented as permitted.

### 2.7.3 Struct Types

| Condition | Passing Method |
|---|---|
| Size ≤ 8 bytes | By value in register |
| Size > 8 bytes | By hidden pointer |

**Contract simplification:** This ABI contract does not pass structs > 8 bytes across the boundary. All complex data is passed via pointers to pre-defined structures.

## 2.8 Function Attributes

### 2.8.1 Required Attributes for ASM Symbols

ASM functions must be declared with:

- Global visibility (`.global` or `global`)
- Proper alignment (function entry 16-byte aligned recommended)
- No stack protector dependencies

### 2.8.2 Required Attributes for Rust FFI

```
#[no_mangle]  // If Rust function called from ASM
extern "win64" fn rust_callback(...) { }
```

### 2.9 Calling Convention Summary Table

| Aspect | Requirement | Violation Consequence |
|---|---|---|
| Parameter registers | RCX, RDX, R8, R9 | Wrong values passed |
| Return register | RAX (or RAX:RDX) | Wrong return value |
| Shadow space | 32 bytes minimum | Stack corruption |
| Stack alignment | 16-byte before call | Potential crash |
| Non-volatile preservation | RBX, RBP, RDI, RSI, R12-R15 | Caller state corruption |
| Direction flag | Clear (DF=0) | String ops malfunction |
| Red zone | None (forbidden) | Stack corruption |

---

# Chunk 3 — Canonical ASM Interface Table

### 3.1 Interface Completeness Guarantee

This section defines the **complete** set of ASM-exported symbols that Rust may call. Any symbol not listed here is internal to ASM and must not be called from Rust.

**Invariant:** The ASM ↔ Rust interface is CLOSED. No additional functions may be added without a contract revision.

### 3.2 Symbol Naming Convention

All ASM symbols callable from Rust follow this pattern:

```
asm_<subsystem>_<operation>
```

| Subsystem | Purpose |
|---|---|
| `tsc` | Time Stamp Counter operations |
| `vq` | Virtqueue operations |
| `mmio` | Memory-mapped I/O |
| `bar` | Memory barrier operations |
| `nic` | NIC-specific operations |

# 3.3 Complete ASM Interface Table

## 3.3.1 Time Stamp Counter Operations

### asm_tsc_read

```
Symbol:     asm_tsc_read
Purpose:    Read the current TSC value
Inputs:     None
Outputs:    RAX = 64-bit TSC value
Clobbers:   RAX, RDX (rdtsc clobbers EDX:EAX)
Barriers:   None (not serializing)
Errors:     None (always succeeds)
Latency:    ~20-40 cycles
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_tsc_read() -> u64;
}
```

**ASM implementation requirements:**

```
asm_tsc_read:
    rdtsc                   ; EDX:EAX = TSC
    shl rdx, 32
    or rax, rdx             ; RAX = full 64-bit TSC
    ret
```

### asm_tsc_read_serialized

```
Symbol:     asm_tsc_read_serialized
Purpose:    Read TSC with serialization (for precise measurement)
Inputs:     None
Outputs:    RAX = 64-bit TSC value
Clobbers:   RAX, RBX, RCX, RDX (cpuid + rdtsc)
Barriers:   Full serialization via CPUID
Errors:     None
Latency:    ~100-200 cycles (CPUID overhead)
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_tsc_read_serialized() -> u64;
}
```

**ASM implementation requirements:**

```
asm_tsc_read_serialized:
    push rbx                ; CPUID clobbers RBX (non-volatile)
    xor eax, eax
    cpuid                   ; Serialize
    rdtsc
    shl rdx, 32
    or rax, rdx
    pop rbx
    ret
```

### 3.3.2 Virtqueue Operations

**asm_vq_submit_tx**

```
Symbol:     asm_vq_submit_tx
Purpose:    Submit a buffer to the TX virtqueue with proper barriers
Inputs:
    RCX = *mut VirtqueueState (pointer to TX queue state)
    RDX = buffer_index: u16 (descriptor index to submit)
    R8  = buffer_length: u16 (total length including 12-byte header)
Outputs:
    RAX = 0 on success
    RAX = 1 if queue full
Clobbers:   RAX, RCX, RDX, R8, R9, R10, R11
Barriers:
    - sfence after descriptor write
    - sfence after avail.ring write
    - mfence before notify (if notify performed)
Errors:     Returns 1 if no free descriptors
Latency:    ~50-100 cycles (barrier-dependent)
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_vq_submit_tx(
        vq_state: *mut VirtqueueState,
        buffer_index: u16,
        buffer_length: u16,
    ) -> u32;
}
```

**Preconditions (Rust must guarantee):**

- `vq_state` points to valid, initialized VirtqueueState
- `buffer_index` < queue_size
- Buffer at `buffer_index` is DRIVER-OWNED
- Buffer content already written (including 12-byte VirtIO header)

**Postconditions (ASM guarantees):**

- On success (0): Buffer is now DEVICE-OWNED
- On failure (1): Buffer ownership unchanged

### `asm_vq_poll_tx_complete`

```
Symbol:     asm_vq_poll_tx_complete
Purpose:    Poll TX used ring for completed transmissions
Inputs:
    RCX = *mut VirtqueueState (pointer to TX queue state)
Outputs:
    RAX = buffer_index: u32 (low 16 bits) if completion found
    RAX = 0xFFFFFFFF if no completion
Clobbers:   RAX, RCX, RDX, R8, R9, R10, R11
Barriers:
    - lfence after reading used.idx (or volatile read)
    - lfence before reading buffer index from used ring
Errors:     None (empty queue returns sentinel)
Latency:    ~30-50 cycles
```

**Rust declaration:**

```
extern "win64" {
    fn asm_vq_poll_tx_complete(vq_state: *mut VirtqueueState) -> u32;
}
```

**Postconditions:**

- If return != 0xFFFFFFFF: Buffer at returned index is now DRIVER-OWNED
- Caller must return buffer to free pool or reuse

### `asm_vq_submit_rx`

```
Symbol:     asm_vq_submit_rx
Purpose:    Submit an empty buffer to RX virtqueue for receiving
Inputs:
    RCX = *mut VirtqueueState (pointer to RX queue state)
```

```
    RDX = buffer_index: u16 (descriptor index)
    R8  = buffer_capacity: u16 (buffer size, must be ≥ 12 + MTU)
Outputs:
    RAX = 0 on success
    RAX = 1 if queue full
Clobbers:   RAX, RCX, RDX, R8, R9, R10, R11
Barriers:   Same as asm_vq_submit_tx
Errors:     Returns 1 if no free descriptors
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_vq_submit_rx(
        vq_state: *mut VirtqueueState,
        buffer_index: u16,
        buffer_capacity: u16,
    ) -> u32;
}
```

**Preconditions:**

- Buffer at `buffer_index` is DRIVER-OWNED
- `buffer_capacity` ≥ 1526 (12-byte header + 1514-byte max frame)

### `asm_vq_poll_rx`

```
Symbol:     asm_vq_poll_rx
Purpose:    Poll RX used ring for received packets
Inputs:
    RCX = *mut VirtqueueState (pointer to RX queue state)
    RDX = *mut RxResult (output structure for result)
Outputs:
    RAX = 0 if no packet available
    RAX = 1 if packet received (result written to RDX)
Clobbers:   RAX, RCX, RDX, R8, R9, R10, R11
Barriers:
    - lfence after reading used.idx
    - lfence before reading packet data
Errors:     None
```

**RxResult structure:**

```rust
#[repr(C)]
pub struct RxResult {
    pub buffer_index: u16,    // Which buffer received the packet
    pub length: u16,          // Bytes written (including 12-byte header)
```

```
    pub _reserved: u32,      // Padding for alignment
}
```

**Rust declaration:**

```
extern "win64" {
    fn asm_vq_poll_rx(
        vq_state: *mut VirtqueueState,
        result: *mut RxResult,
    ) -> u32;
}
```

**Postconditions:**

- If return == 1: Buffer at `result.buffer_index` is DRIVER-OWNED
- Caller must process packet, then resubmit buffer or free

**asm_vq_notify**

```
Symbol:     asm_vq_notify
Purpose:    Notify device that new buffers are available
Inputs:
    RCX = *mut VirtqueueState (pointer to queue state)
Outputs:    None (void)
Clobbers:   RAX, RCX, RDX, R8, R9, R10, R11
Barriers:   mfence before MMIO write (implicit in implementation)
Errors:     None
Latency:    ~100-500 cycles (MMIO write)
```

**Rust declaration:**

```
extern "win64" {
    fn asm_vq_notify(vq_state: *mut VirtqueueState);
}
```

**Notes:**

- Notification may be suppressed if device indicates no notification needed
- Suppression check is internal to ASM

### 3.3.3 Memory Barrier Operations

**asm_bar_sfence**

```

```

```
Symbol:     asm_bar_sfence
Purpose:    Store fence - ensures all prior stores are visible
Inputs:     None
Outputs:    None
Clobbers:   None
Barriers:   sfence
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_bar_sfence();
}
```

### asm_bar_lfence

```
Symbol:     asm_bar_lfence
Purpose:    Load fence - ensures all prior loads complete
Inputs:     None
Outputs:    None
Clobbers:   None
Barriers:   lfence
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_bar_lfence();
}
```

### asm_bar_mfence

```
Symbol:     asm_bar_mfence
Purpose:    Full fence - ensures all prior loads AND stores complete
Inputs:     None
Outputs:    None
Clobbers:   None
Barriers:   mfence
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_bar_mfence();
```

```
}
```

### 3.3.4 MMIO Operations

**asm_mmio_read32**

```
Symbol:     asm_mmio_read32
Purpose:    Read 32-bit value from MMIO address
Inputs:
    RCX = address: u64 (MMIO address, must be 4-byte aligned)
Outputs:
    RAX = value: u32 (zero-extended to 64 bits)
Clobbers:   RAX
Barriers:   Implicit (MMIO on x86 is UC/strongly ordered)
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_mmio_read32(address: u64) -> u32;
}
```

**asm_mmio_write32**

```
Symbol:     asm_mmio_write32
Purpose:    Write 32-bit value to MMIO address
Inputs:
    RCX = address: u64 (MMIO address, must be 4-byte aligned)
    RDX = value: u32
Outputs:    None
Clobbers:   None
Barriers:   Implicit
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_mmio_write32(address: u64, value: u32);
}
```

**asm_mmio_read16 / asm_mmio_write16**

```
```

```
Symbol:     asm_mmio_read16 / asm_mmio_write16
Purpose:    16-bit MMIO read/write
Inputs/Outputs: Same pattern as 32-bit variants
Alignment:  2-byte aligned
```

### 3.3.5 NIC Initialization

**asm_nic_reset**

```
Symbol:     asm_nic_reset
Purpose:    Reset VirtIO NIC device
Inputs:
    RCX = mmio_base: u64 (device MMIO base address)
Outputs:
    RAX = 0 on success
    RAX = 1 on timeout (device did not reset)
Clobbers:   RAX, RCX, RDX, R8, R9, R10, R11
Barriers:   Internal barriers as needed
Timeout:    100ms maximum
```

**Rust declaration:**

```rust
extern "win64" {
    fn asm_nic_reset(mmio_base: u64) -> u32;
}
```

**asm_nic_set_status**

```
Symbol:     asm_nic_set_status
Purpose:    Write device status register
Inputs:
    RCX = mmio_base: u64
    RDX = status: u8 (status bits to write)
Outputs:    None
Clobbers:   RAX, RCX, RDX
```

**asm_nic_get_status**

```
Symbol:     asm_nic_get_status
Purpose:    Read device status register
Inputs:
    RCX = mmio_base: u64
Outputs:
```

```
    RAX = status: u8 (current status bits)
Clobbers:   RAX
```

**asm_nic_read_features**

```
Symbol:     asm_nic_read_features
Purpose:    Read device feature bits
Inputs:
    RCX = mmio_base: u64
Outputs:
    RAX = features: u64 (device-offered features)
Clobbers:   RAX, RCX, RDX
```

**asm_nic_write_features**

```
Symbol:     asm_nic_write_features
Purpose:    Write driver-accepted feature bits
Inputs:
    RCX = mmio_base: u64
    RDX = features: u64 (features to accept)
Outputs:    None
Clobbers:   RAX, RCX, RDX
```

**asm_nic_read_mac**

```
Symbol:     asm_nic_read_mac
Purpose:    Read MAC address from device config space
Inputs:
    RCX = mmio_base: u64
    RDX = *mut [u8; 6] (output buffer for MAC)
Outputs:
    RAX = 0 on success
    RAX = 1 if MAC not available (feature not negotiated)
Clobbers:   RAX, RCX, RDX, R8
```

**Rust declaration:**

```
extern "win64" {
    fn asm_nic_read_mac(mmio_base: u64, mac_out: *mut [u8; 6]) -> u32;
}
```

## 3.4 Shared Data Structures

### 3.4.1 VirtqueueState

```rust
/// Virtqueue state shared between ASM and Rust.
/// This structure MUST NOT be modified by Rust except at initialization.
#[repr(C, align(64))]  // Cache line aligned
pub struct VirtqueueState {
    // === Descriptor Table ===
    /// Physical address of descriptor table
    pub desc_table_phys: u64,
    /// CPU pointer to descriptor table
    pub desc_table_ptr: *mut VirtqDesc,

    // === Available Ring ===
    /// Physical address of available ring
    pub avail_ring_phys: u64,
    /// CPU pointer to available ring
    pub avail_ring_ptr: *mut VirtqAvail,

    // === Used Ring ===
    /// Physical address of used ring
    pub used_ring_phys: u64,
    /// CPU pointer to used ring
    pub used_ring_ptr: *mut VirtqUsed,

    // === Queue Parameters ===
    /// Number of entries (power of 2, ≤ 32768)
    pub queue_size: u16,
    /// Queue index (0 = RX, 1 = TX)
    pub queue_index: u16,
    /// Padding
    pub _pad0: u32,

    // === Driver State (ASM-owned) ===
    /// Next descriptor index to use
    pub next_desc_idx: u16,
    /// Last seen used index
    pub last_used_idx: u16,
    /// Number of free descriptors
    pub free_desc_count: u16,
    /// Padding
    pub _pad1: u16,

    // === Device Notification ===
    /// MMIO address for queue notification
    pub notify_addr: u64,
    /// Notification offset (for modern VirtIO)
    pub notify_offset: u32,
    /// Padding
    pub _pad2: u32,

    // === Buffer Tracking ===
    /// Pointer to buffer physical addresses array
```

```
    pub buffer_phys_addrs: *const u64,
    /// Pointer to buffer CPU addresses array
    pub buffer_cpu_addrs: *const *mut u8,
    /// Size of each buffer in bytes
    pub buffer_size: u32,
    /// Number of buffers
    pub buffer_count: u32,
}
```

### 3.4.2 VirtIO Descriptor

```
/// VirtIO descriptor table entry (16 bytes)
#[repr(C)]
pub struct VirtqDesc {
    /// Physical address of buffer
    pub addr: u64,
    /// Length of buffer
    pub len: u32,
    /// Flags (NEXT, WRITE, INDIRECT)
    pub flags: u16,
    /// Next descriptor index (if NEXT flag set)
    pub next: u16,
}
```

### 3.4.3 VirtIO Available Ring

```
/// VirtIO available ring
#[repr(C)]
pub struct VirtqAvail {
    /// Flags (notification suppression)
    pub flags: u16,
    /// Next available index (incremented by driver)
    pub idx: u16,
    /// Ring entries (variable size, queue_size entries)
    pub ring: [u16; 0],  // Flexible array member
    // Followed by: used_event: u16 (if VIRTIO_F_EVENT_IDX)
}
```

### 3.4.4 VirtIO Used Ring

```
/// VirtIO used ring
#[repr(C)]
pub struct VirtqUsed {
    /// Flags (notification suppression)
    pub flags: u16,
    /// Next used index (incremented by device)
```

```
    pub idx: u16,
    /// Ring entries (variable size, queue_size entries)
    pub ring: [VirtqUsedElem; 0],  // Flexible array member
    // Followed by: avail_event: u16 (if VIRTIO_F_EVENT_IDX)
}

#[repr(C)]
pub struct VirtqUsedElem {
    /// Descriptor chain head index
    pub id: u32,
    /// Bytes written by device
    pub len: u32,
}
```

### 3.5 Interface Constraints Summary

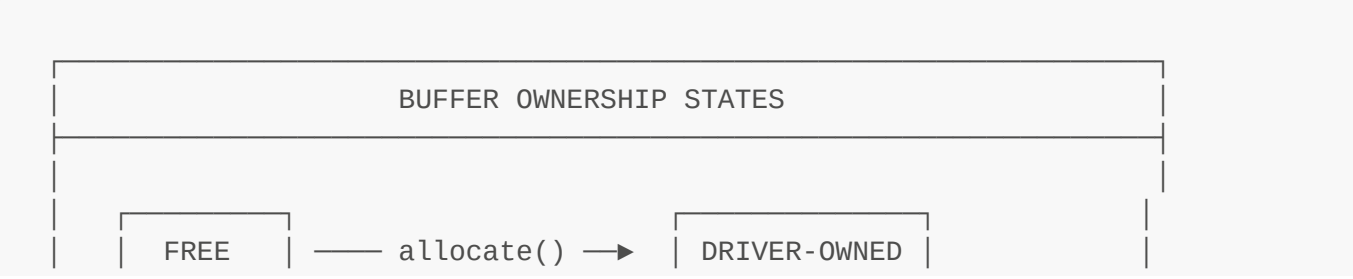
| Symbol | Max Latency | May Block | Barriers |
|---|---|---|---|
| `asm_tsc_read` | 40 cycles | No | None |
| `asm_tsc_read_serialized` | 200 cycles | No | CPUID |
| `asm_vq_submit_tx` | 100 cycles | No | sfence, mfence |
| `asm_vq_poll_tx_complete` | 50 cycles | No | lfence |
| `asm_vq_submit_rx` | 100 cycles | No | sfence, mfence |
| `asm_vq_poll_rx` | 50 cycles | No | lfence |
| `asm_vq_notify` | 500 cycles | No | mfence |
| `asm_bar_*` | 50 cycles | No | As named |
| `asm_mmio_*` | 200 cycles | No | Implicit |
| `asm_nic_reset` | 100ms | Yes (bounded) | Internal |

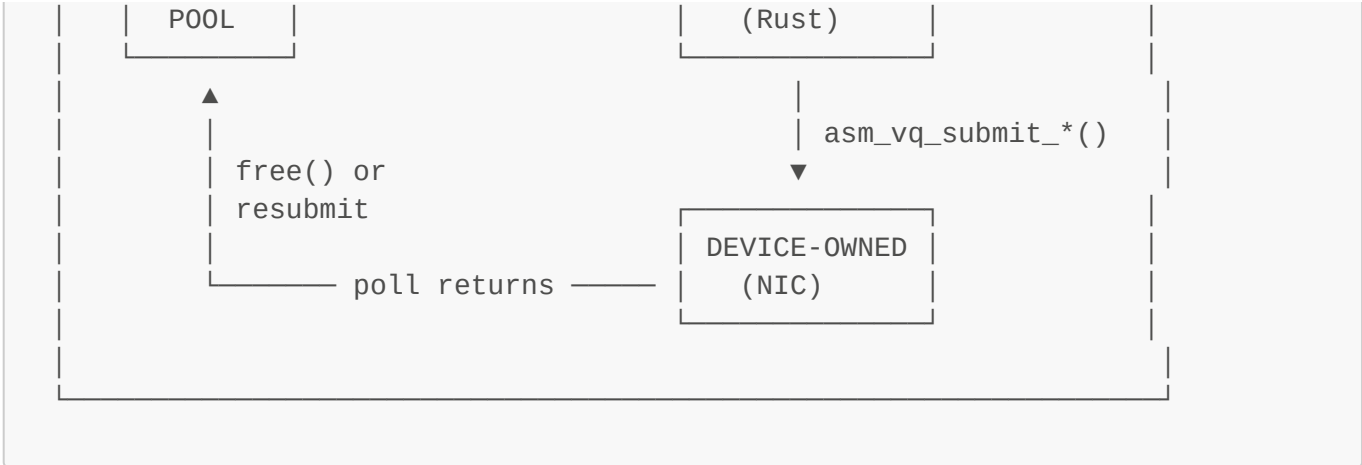
---

# Chunk 4 — Ownership & Lifetime Semantics

### 4.1 Ownership Model Overview

Buffer ownership in this ABI follows an **exclusive ownership** model. At any instant, each DMA buffer is in exactly one of three states:






**Source:** REDESIGN §3.3 Buffer Ownership Model

## 4.2 State Definitions

### 4.2.1 FREE State

A buffer is FREE when:

- It is in the allocator's free list
- No descriptor references it
- Neither driver nor device may access its contents

**Transitions:**

- FREE → DRIVER-OWNED: Via buffer allocation

### 4.2.2 DRIVER-OWNED State

A buffer is DRIVER-OWNED when:

- It has been allocated but not submitted to a virtqueue
- OR it has been returned from the used ring but not freed

**Permitted operations (Rust):**

- Read buffer contents
- Write buffer contents
- Free buffer (return to FREE)
- Submit to virtqueue (transition to DEVICE-OWNED)

**Forbidden operations (Rust):**

- None (full access permitted)

### 4.2.3 DEVICE-OWNED State

A buffer is DEVICE-OWNED when:

- It has been submitted to a virtqueue via `asm_vq_submit_*`
- AND it has not yet appeared in the used ring

**Permitted operations:**

- None from Rust
- Device (NIC) may read (TX) or write (RX) the buffer

**Forbidden operations (Rust):**

- Read buffer contents
- Write buffer contents
- Free buffer
- Resubmit to any queue

**Violation:** Any access to a DEVICE-OWNED buffer from Rust is **undefined behavior**.

## 4.3 Ownership Transfer Rules

### 4.3.1 Rule OWN-1: Submission Transfers Ownership to Device

```
RULE OWN-1: Virtqueue Submission

PRECONDITION:
    Buffer B is DRIVER-OWNED
    Buffer content is fully initialized (TX) or ready to receive (RX)

ACTION:
    Call asm_vq_submit_tx(state, B.index, length)
    OR asm_vq_submit_rx(state, B.index, capacity)

POSTCONDITION (if success):
    Buffer B is DEVICE-OWNED
    Rust MUST NOT access B until poll returns it

POSTCONDITION (if failure - queue full):
    Buffer B remains DRIVER-OWNED
    Rust may retry or take other action
```

**Source:** REDESIGN §3.3 DMA-OWN-1, AUDIT §2.1

### 4.3.2 Rule OWN-2: Poll Completion Transfers Ownership to Driver

```
RULE OWN-2: Virtqueue Completion

PRECONDITION:
    Buffer B is DEVICE-OWNED (submitted earlier)
    Device has placed B's descriptor index in used ring

ACTION:
    Call asm_vq_poll_tx_complete(state) → returns B.index
    OR asm_vq_poll_rx(state, result) → returns B.index in result
```

```
POSTCONDITION:
    Buffer B is DRIVER-OWNED
    Rust may access B.contents
    For RX: B contains packet data (12-byte header + frame)
    For TX: B may be freed or reused
```

**Source:** REDESIGN §3.3 DMA-OWN-2, DMA-OWN-3

### 4.3.3 Rule OWN-3: Free Returns to Pool

```
RULE OWN-3: Buffer Deallocation

PRECONDITION:
    Buffer B is DRIVER-OWNED
    No references to B exist

ACTION:
    Return B to free pool (Rust-side allocator)

POSTCONDITION:
    Buffer B is FREE
    B may be reallocated for different purpose
```

## 4.4 RX Buffer Lifecycle

```
LIFECYCLE RX-1: Receive Buffer

1. ALLOCATE
   Rust: B = alloc_rx_buffer()
   State: B is DRIVER-OWNED

2. SUBMIT
   Rust: asm_vq_submit_rx(state, B.index, capacity)
   State: B is DEVICE-OWNED
   Rust: MUST NOT access B

3. RECEIVE (async, by device)
   Device: Writes packet to B
   Device: Marks B in used ring with length
   State: B is still DEVICE-OWNED (from Rust perspective)

4. POLL
   Rust: asm_vq_poll_rx(state, result) → returns 1
   State: B is DRIVER-OWNED
   result.buffer_index = B.index
   result.length = bytes written (including 12-byte header)

5. CONSUME
   Rust: Process packet in B[12..result.length]
```

```
   B[0..12] contains VirtIO header (typically ignored)

6. RECYCLE or FREE
   Option A: asm_vq_submit_rx(state, B.index, capacity) → goto step 2
   Option B: free_rx_buffer(B) → B is FREE
```

**Timing constraint:** Steps 4-6 should complete within one main loop iteration to prevent RX queue starvation.

## 4.5 TX Buffer Lifecycle

```
LIFECYCLE TX-1: Transmit Buffer

1. ALLOCATE
   Rust: B = alloc_tx_buffer()
   State: B is DRIVER-OWNED

2. PREPARE
   Rust: Zero B[0..12] (VirtIO header, no offloads)
   Rust: Copy Ethernet frame to B[12..12+frame_len]
   Total length = 12 + frame_len

3. SUBMIT
   Rust: asm_vq_submit_tx(state, B.index, 12 + frame_len)
   State: B is DEVICE-OWNED
   Rust: MUST NOT access B

4. TRANSMIT (async, by device)
   Device: Reads packet from B
   Device: Transmits on wire
   Device: Marks B in used ring
   State: B is still DEVICE-OWNED (from Rust perspective)

5. POLL COMPLETION
   Rust: asm_vq_poll_tx_complete(state) → returns B.index
   State: B is DRIVER-OWNED

6. FREE
   Rust: free_tx_buffer(B)
   State: B is FREE
```

**Note:** TX completion may be delayed. Rust must not assume completion timing.

## 4.6 smoltcp Token Lifetimes

smoltcp's Device trait uses tokens to manage buffer lifetimes. This section specifies how tokens interact with ASM ownership.

### 4.6.1 RxToken Lifetime

```rust
pub struct VirtioRxToken<'a> {
    driver: &'a mut VirtioDriver,
    buffer_index: u16,
    length: u16,
}
```

**Lifetime rule:**

- RxToken borrows `&mut VirtioDriver`
- While RxToken exists, buffer is DRIVER-OWNED
- `consume()` must be called before RxToken is dropped
- After `consume()`, buffer may be resubmitted

```
INVARIANT TOKEN-RX-1: RxToken Buffer Validity

WHILE RxToken exists:
    Buffer at token.buffer_index is DRIVER-OWNED
    Buffer contents are valid and readable

ON RxToken::consume():
    Closure F receives &mut buffer[12..length]
    After F returns, buffer may be resubmitted

ON RxToken::drop() without consume():
    Buffer must still be resubmitted or freed
    Packet data is lost (acceptable)
```

### 4.6.2 TxToken Lifetime

```rust
pub struct VirtioTxToken<'a> {
    driver: &'a mut VirtioDriver,
}
```

**Lifetime rule:**

- TxToken borrows `&mut VirtioDriver`
- `consume(len, F)` allocates buffer, calls F, then submits
- After `consume()`, buffer is DEVICE-OWNED

```
INVARIANT TOKEN-TX-1: TxToken Submission

ON TxToken::consume(len, F):
    1. Allocate buffer B (DRIVER-OWNED)
    2. Zero B[0..12] (VirtIO header)
    3. Call F(&mut B[12..12+len])
    4. Call asm_vq_submit_tx(state, B.index, 12+len)
```

```
    5. B is now DEVICE-OWNED

ON TxToken::drop() without consume():
    No buffer allocated, no action needed
```

## 4.7 Descriptor Table Ownership

The descriptor table is shared between ASM and Rust with specific ownership rules.

### 4.7.1 Descriptor Lifecycle

```
DESCRIPTOR STATE MACHINE:

    FREE ──► DRIVER-OWNED ──► IN-FLIGHT ──► DRIVER-OWNED ──► FREE
             (preparing)      (submitted)    (completed)

FREE:
    - Descriptor in ASM free list
    - ASM may allocate

DRIVER-OWNED (preparing):
    - ASM has allocated descriptor
    - Rust fills in addr, len, flags via VirtqueueState pointers
    - Not yet in available ring

IN-FLIGHT:
    - Descriptor index in available ring
    - Device may read descriptor
    - ASM tracks in in_flight_count

DRIVER-OWNED (completed):
    - Descriptor index returned from used ring
    - Rust may read completion status
    - ASM returns to free list
```

### 4.7.2 Descriptor Access Rules

| Field | Who May Write | When |
|---|---|---|
| `desc.addr` | Rust (via ASM init) | Before submit only |
| `desc.len` | Rust (via ASM init) | Before submit only |
| `desc.flags` | Rust (via ASM init) | Before submit only |
| `desc.next` | ASM | Always (free list management) |
| `avail.idx` | ASM | On submit |
| `avail.ring[i]` | ASM | On submit |

| Field | Who May Write | When |
|---|---|---|
| `used.idx` | Device | Async |
| `used.ring[i]` | Device | Async |

## 4.8 Lifetime Invariants Summary

| ID | Invariant | Violation Consequence |
|---|---|---|
| **LIFE-1** | Buffer in exactly one state at any time | Data corruption |
| **LIFE-2** | DEVICE-OWNED buffers never accessed by Rust | Data corruption, UB |
| **LIFE-3** | RxToken consume() before resubmit | Use-after-free |
| **LIFE-4** | TxToken consume() transfers ownership | Buffer leak |
| **LIFE-5** | Descriptors freed only after used ring return | Descriptor leak |
| **LIFE-6** | Buffer physical address stable for lifetime | DMA corruption |

## 4.9 Buffer Memory Requirements

### 4.9.1 Buffer Sizing

| Buffer Type | Minimum Size | Recommended | Contents |
|---|---|---|---|
| RX | 1526 bytes | 2048 bytes | 12B header + 1514B max frame |
| TX | 1526 bytes | 2048 bytes | 12B header + 1514B max frame |

### 4.9.2 Buffer Alignment

| Requirement | Value | Rationale |
|---|---|---|
| Start alignment | 8 bytes | Descriptor addr field |
| DMA alignment | 4 bytes minimum | VirtIO spec |
| Recommended | 64 bytes | Cache line |

### 4.9.3 Buffer Count

| Queue | Minimum | Recommended | Maximum |
|---|---|---|---|
| RX | 2 | 32 | queue_size |
| TX | 2 | 32 | queue_size |

**Constraint:** Buffer count ≤ queue_size (negotiated with device).

---

# Chunk 5 — Memory Ordering & DMA Visibility

## 5.1 Memory Model Context

This ABI operates under the x86-64 Total Store Order (TSO) memory model with the following characteristics:

| Property | x86-64 Behavior | Source |
|---|---|---|
| Store-Store ordering | Preserved | Intel SDM §8.2.2 |
| Load-Load ordering | Preserved | Intel SDM §8.2.2 |
| Load-Store ordering | Preserved | Intel SDM §8.2.2 |
| Store-Load ordering | MAY reorder | Intel SDM §8.2.2 |
| Compiler reordering | Must be prevented | Volatile/barriers |

**Critical:** The TSO model applies to CPU ↔ CPU communication. DMA (CPU ↔ Device) has additional requirements.

**Source:** AUDIT §2.3, §7.2.2

## 5.2 The Three Ordering Concerns

This contract distinguishes three separate ordering concerns:

### 5.2.1 Compiler Ordering

**Problem:** The Rust/LLVM compiler may reorder memory accesses.

**Solution:** Use `volatile` operations or inline assembly barriers.

```
// Compiler barrier (does NOT emit CPU instruction)
core::sync::atomic::compiler_fence(Ordering::SeqCst);

// Volatile read (prevents compiler reordering)
core::ptr::read_volatile(addr);

// Volatile write (prevents compiler reordering)
core::ptr::write_volatile(addr, value);
```

**Contract rule:** All MMIO and DMA buffer accesses must use volatile operations or be inside ASM functions.

### 5.2.2 CPU Store Buffer Ordering

**Problem:** CPU stores may sit in the store buffer, invisible to other agents.

**Solution:** Use `sfence` (store fence) to flush store buffer.

```
; After writing to DMA buffer:
sfence          ; Ensure stores visible to coherency domain
```

**Contract rule:** ASM inserts `sfence` before any operation that makes data visible to device.

### 5.2.3 Cache Coherency

**Problem:** On x86 with standard write-back memory, CPU caches are coherent with DMA by hardware design. However, this requires:

- The memory region be in the coherency domain
- No non-coherent memory types (with some exceptions)

**Solution (this contract):** DMA regions mapped as Uncached (UC) or Write-Combining (WC).

**Source:** AUDIT §7.2.2 — "DMA region MUST be one of: a) Uncached (UC), b) Write-Combining (WC), c) Write-Back (WB) with explicit cache management"

**Contract choice:** UC or WC mapping. No explicit `clflush` required.

## 5.3 Barrier Placement Contracts

### 5.3.1 TX Submission Barrier Contract

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

SEQUENCE (must execute in this exact order):

1. WRITE buffer data
   - Rust: Writes packet to buffer[0..length]
   - Memory: UC/WC, so writes go directly to memory (no cache)
   - No barrier needed between buffer writes (TSO)

2. WRITE descriptor
   - ASM: Writes desc.addr, desc.len, desc.flags
   - Memory: UC mapping for descriptor table

3. SFENCE
   - ASM: sfence instruction
   - Purpose: Ensure descriptor visible before index update

4. WRITE available ring entry
   - ASM: avail.ring[avail.idx & mask] = desc_index

5. SFENCE
   - ASM: sfence instruction
   - Purpose: Ensure ring entry visible before index update

6. WRITE available index
   - ASM: avail.idx += 1
```

```
7. MFENCE
   - ASM: mfence instruction
   - Purpose: Full barrier before notify (store-load ordering)

8. CONDITIONAL NOTIFY
   - ASM: Check if notification needed
   - ASM: If yes, write to notify MMIO register

RATIONALE:
    Device may read descriptors immediately after seeing avail.idx change.
    Without barriers, device may see stale descriptor data.
    mfence before notify prevents notify being observed before idx update.
```

**Source:** AUDIT §5.4.1

### 5.3.2 RX Completion Barrier Contract

```
CONTRACT MEM-RX-1: RX Completion Polling Ordering

SEQUENCE (must execute in this exact order):

1. READ used.idx
   - ASM: Volatile read of used.idx
   - Note: x86 provides acquire semantics on reads to UC memory

2. COMPARE
   - ASM: if used.idx == last_used_idx, return (no completions)

3. LFENCE
   - ASM: lfence instruction
   - Purpose: Ensure idx read completes before ring entry read
   - Note: May be unnecessary on x86 TSO, but required for correctness
proof

4. READ used ring entry
   - ASM: entry = used.ring[last_used_idx & mask]
   - Obtains: desc_index and bytes_written

5. LFENCE (or implicit via data dependency)
   - ASM: lfence before reading buffer contents
   - Purpose: Ensure ring read completes before buffer access

6. RETURN to caller
   - ASM: Returns desc_index and length
   - Rust: May now access buffer contents

7. UPDATE tracking
   - ASM: last_used_idx += 1 (internal bookkeeping)

RATIONALE:
```

```
    Device writes buffer → writes used ring → increments used.idx
    Driver must read in reverse order with barriers to see consistent data.
```

**Source:** AUDIT §5.4.2

### 5.3.3 Device Notification Barrier Contract

```
CONTRACT MEM-NOTIFY-1: Device Notification

PRECONDITION:
    All relevant available ring updates complete
    avail.idx reflects all submitted descriptors

SEQUENCE:

1. MFENCE
   - Purpose: Ensure all stores visible before MMIO write

2. CHECK notification suppression (optional optimization)
   - Read used.flags for VIRTQ_USED_F_NO_NOTIFY
   - If set, skip notification

3. MMIO WRITE to notify register
   - VirtIO modern: queue_notify register
   - This is a posted write (no completion)

RATIONALE:
    MMIO writes are UC and serialize at the bus level.
    mfence ensures all prior DMA memory writes are visible
    before the device processes the notification.
```

## 5.4 Memory Type Requirements

### 5.4.1 DMA Region Memory Type

| Region | Required Type | Alternative | Source |
|---|---|---|---|
| Descriptor tables | UC | WC | AUDIT §2.4 CACHE-2 |
| Available rings | UC | WC | AUDIT §2.4 CACHE-2 |
| Used rings | UC | WC | AUDIT §2.4 CACHE-2 |
| RX buffers | UC | WC | AUDIT §2.4 CACHE-1 |
| TX buffers | UC | WC (preferred) | AUDIT §2.4 CACHE-1 |

**UC (Uncached):**

- Reads/writes go directly to memory
- No caching, no speculation

- Slowest but simplest

**WC (Write-Combining):**

- Writes may be combined in WC buffer
- Reads go directly to memory
- Better performance for sequential writes (TX buffers)
- Requires `sfence` to flush WC buffers

### 5.4.2 MMIO Memory Type

All MMIO regions (device registers) are implicitly UC on x86.

| Region | Type | Ordering |
|---|---|---|
| VirtIO capability registers | UC | Strongly ordered |
| Notify region | UC | Strongly ordered |
| Device config space | UC | Strongly ordered |

## 5.5 Visibility Guarantees

### 5.5.1 Rust → Device Visibility

```
GUARANTEE VIS-1: Rust Write Visibility to Device

GIVEN:
    Rust writes data to DMA buffer (DRIVER-OWNED)
    Rust calls asm_vq_submit_tx()

THEN:
    Device will see the written data

MECHANISM:
    1. Buffer is UC/WC → writes bypass cache
    2. ASM inserts sfence before avail.idx update
    3. ASM inserts mfence before notify
    4. Device reads after notify sees consistent data
```

### 5.5.2 Device → Rust Visibility

```
GUARANTEE VIS-2: Device Write Visibility to Rust

GIVEN:
    Device writes data to DMA buffer (RX)
    Device updates used ring

THEN:
    After asm_vq_poll_rx() returns, Rust sees device's writes
```

```
MECHANISM:
    1. Buffer is UC/WC → device writes go to memory
    2. ASM reads used.idx (volatile)
    3. ASM inserts lfence before reading ring entry
    4. ASM inserts lfence before returning buffer index
    5. Rust reads buffer after poll returns → sees device data
```

## 5.6 Completion Detection

### 5.6.1 TX Completion Detection

TX completion is detected by comparing used.idx against last_used_idx:

```
DETECTION TX-COMPLETE:

last_used_idx: Driver's record of last processed used entry
used.idx: Device's counter of completed descriptors

COMPLETION AVAILABLE iff:
    used.idx != last_used_idx
    (accounting for 16-bit wraparound)

ENTRIES AVAILABLE:
    count = (used.idx - last_used_idx) & 0xFFFF
```

### 5.6.2 RX Completion Detection

Identical mechanism to TX:

```
DETECTION RX-COMPLETE:

COMPLETION AVAILABLE iff:
    used.idx != last_used_idx

ON EACH COMPLETION:
    Buffer contains: 12-byte VirtIO header + Ethernet frame
    Length field: Total bytes written by device
    Payload: buffer[12..length]
```

## 5.7 Ordering Violations and Consequences

| Violation | Consequence | Detection |
|---|---|---|
| Missing sfence before avail.idx | Device sees stale descriptors | Corrupted packets, hangs |
| Missing mfence before notify | Device notified before data visible | Packet loss, corruption |

| Violation | Consequence | Detection |
|---|---|---|
| Missing lfence after used.idx | Rust reads stale ring entry | Wrong buffer returned |
| Reading DEVICE-OWNED buffer | Data race with DMA | Corruption, unpredictable |
| WB memory without clflush | Device reads stale cache data | Corrupted TX packets |

## 5.8 Barrier Summary Table

| Operation | Before | After | Rationale |
|---|---|---|---|
| Write descriptor | — | sfence | Visible before idx |
| Write avail.ring | — | sfence | Visible before idx |
| Write avail.idx | sfence | mfence | Visible before notify |
| Write notify | mfence | — | All data visible |
| Read used.idx | — | lfence | Before ring read |
| Read used.ring | lfence | lfence | Before buffer read |
| Read buffer | lfence | — | After ring read |

## 5.9 ASM Implementation Requirements

### 5.9.1 Submit Implementation Pattern

```
; asm_vq_submit_tx implementation pattern
asm_vq_submit_tx:
    ; ... setup, get descriptor ...

    ; Write descriptor fields
    mov [desc + VIRTQ_DESC_ADDR], buffer_addr
    mov [desc + VIRTQ_DESC_LEN], length
    mov [desc + VIRTQ_DESC_FLAGS], flags

    ; BARRIER: Ensure descriptor visible
    sfence

    ; Write to available ring
    mov [avail_ring + offset], desc_index

    ; BARRIER: Ensure ring entry visible
    sfence

    ; Increment available index
    inc word [avail + VIRTQ_AVAIL_IDX]

    ; BARRIER: Full fence before notify
    mfence
```

```
    ; Notify device (if needed)
    ; ... check suppression, write notify register ...

    ret
```

### 5.9.2 Poll Implementation Pattern

```
; asm_vq_poll_rx implementation pattern
asm_vq_poll_rx:
    ; Read used index (volatile)
    mov ax, [used + VIRTQ_USED_IDX]

    ; Compare with last seen
    cmp ax, [state + LAST_USED_IDX]
    je .no_completion

    ; BARRIER: Ensure idx read before ring read
    lfence

    ; Read used ring entry
    ; ... calculate offset, read id and len ...

    ; BARRIER: Ensure ring read before buffer access
    lfence

    ; Update last_used_idx
    inc word [state + LAST_USED_IDX]

    ; Return success with buffer info
    mov eax, 1
    ret

.no_completion:
    xor eax, eax
    ret
```

---

# Chunk 6 — Time & Progress Guarantees

## 6.1 Time Source

### 6.1.1 Sole Time Source: TSC

The **only** time source available in this ABI is the Time Stamp Counter (TSC).

| Property | Value | Source |
|---|---|---|
| Instruction | `rdtsc` / `rdtscp` | Intel SDM |

| Property | Value | Source |
|---|---|---|
| Resolution | Sub-nanosecond (at multi-GHz) | CPU-dependent |
| Width | 64 bits | Wraps at ~200 years @ 3GHz |
| Access | Ring 0 or Ring 3 | Always available |

**No other time sources:**

- No PIT (8254) — requires I/O ports, UEFI may have reconfigured
- No HPET — requires discovery, mapping
- No APIC timer — requires interrupt setup
- No RTC — not useful for sub-second timing

**Source:** REDESIGN §7.4, AUDIT §2.5 TIME-1

### 6.1.2 TSC Requirements

Before using TSC, the following MUST be verified:

```
REQUIREMENT TSC-1: Invariant TSC Verification

CHECK: CPUID.80000007H:EDX[bit 8] == 1

IF NOT SET:
    TSC frequency may vary with power states
    System is non-compliant with this ABI
    Behavior: Halt with error

IF SET:
    TSC increments at constant rate
    Independent of P-states, C-states
    Safe for timing measurements
```

**Source:** AUDIT §7.1.4

### 6.1.3 TSC Calibration

TSC frequency MUST be calibrated at boot. Hardcoded frequencies are forbidden.

```
REQUIREMENT TSC-2: Calibration at Boot

TIMING: Before ExitBootServices (UEFI services available)

METHOD:
    1. Read TSC: start = rdtsc()
    2. Call UEFI Stall(1_000_000) — 1 second delay
    3. Read TSC: end = rdtsc()
    4. Frequency = end - start (ticks per second)
```

```
STORAGE:
    Store in BootHandoff.tsc_freq
    Pass to all timing-dependent code

ACCURACY:
    ±5% acceptable for network protocol timeouts
    UEFI Stall() accuracy varies by firmware
```

**Source:** REDESIGN §5.4, §7.6, AUDIT §2.5 TIME-2

## 6.2 Monotonicity Guarantee

### 6.2.1 Single-Core Monotonicity

```
GUARANTEE MONO-1: TSC Monotonicity (Single Core)

GIVEN:
    Invariant TSC verified (CPUID check)
    Single-core execution (no SMP)

THEN:
    For any two reads t1, t2 where t1 happens-before t2:
    t2 >= t1 (no backward jumps)

CAVEAT:
    Multi-core TSCs may be unsynchronized
    This ABI runs single-core only, so this is not a concern
```

**Source:** AUDIT §7.1.5

### 6.2.2 Wraparound Handling

```
GUARANTEE MONO-2: Wraparound Safety

TSC WIDTH: 64 bits
WRAP PERIOD: ~200 years at 3GHz

HANDLING:
    Use wrapping arithmetic for differences:
    elapsed = now.wrapping_sub(start)

    This correctly handles wrap (which won't occur in practice)
```

## 6.3 Resolution Guarantee

### 6.3.1 TSC Resolution

```
GUARANTEE RES-1: TSC Resolution

MINIMUM RESOLUTION: 1 TSC tick
TYPICAL RESOLUTION: ~0.3-0.5 nanoseconds (at 2-3 GHz)

FOR NETWORK PROTOCOLS:
    smoltcp requires millisecond timestamps
    1 ms = ~3,000,000 ticks at 3GHz
    Resolution is more than sufficient
```

### 6.3.2 Conversion Functions

```
CONVERSION TIME-1: TSC to Milliseconds

FORMULA:
    milliseconds = tsc_ticks / (tsc_freq / 1000)

    OR (avoiding overflow):
    milliseconds = tsc_ticks * 1000 / tsc_freq

IMPLEMENTATION:
    // Safe for tsc_ticks < 2^54 (centuries at 3GHz)
    fn tsc_to_ms(ticks: u64, freq: u64) -> u64 {
        ticks / (freq / 1000)
    }
```

```
CONVERSION TIME-2: Milliseconds to TSC

FORMULA:
    tsc_ticks = milliseconds * (tsc_freq / 1000)

IMPLEMENTATION:
    fn ms_to_tsc(ms: u64, freq: u64) -> u64 {
        ms * (freq / 1000)
    }
```

## 6.4 smoltcp Integration

### 6.4.1 Timestamp Requirements

smoltcp's `Instant` type uses milliseconds internally:

```
// smoltcp timestamp creation
let timestamp = smoltcp::time::Instant::from_millis(ms as i64);
```

**Contract requirement:**

- Rust reads TSC via `asm_tsc_read()`
- Rust converts to milliseconds using calibrated `tsc_freq`
- Rust passes milliseconds to smoltcp

### 6.4.2 Main Loop Timestamp Pattern

```
// CORRECT: Main loop timestamp usage
fn main_loop(tsc_freq: u64) {
    loop {
        // Read TSC once per iteration
        let now_tsc = unsafe { asm_tsc_read() };

        // Convert to smoltcp timestamp
        let now_ms = now_tsc / (tsc_freq / 1000);
        let timestamp = Instant::from_millis(now_ms as i64);

        // Pass to smoltcp
        iface.poll(timestamp, &mut device, &mut sockets);

        // Use same TSC value for timeout checks
        app_state.step(now_tsc, tsc_freq);
    }
}
```

### 6.4.3 Timer Precision for TCP

| TCP Timer | Typical Value | TSC Precision | Adequate? |
|---|---|---|---|
| RTO (min) | 200ms | 0.0003ms | Yes |
| RTO (max) | 120s | 0.0003ms | Yes |
| Keepalive | 2h | 0.0003ms | Yes |
| TIME-WAIT | 2×MSL (120s) | 0.0003ms | Yes |

All TCP timers have precision far exceeding requirements.

## 6.5 Progress Guarantees

### 6.5.1 Non-Blocking Function Guarantee

```
GUARANTEE PROG-1: Bounded Function Execution

ALL ASM interface functions (except asm_nic_reset) MUST:
    - Return in bounded time
    - Not spin-wait on external conditions
    - Not loop waiting for device state
```

```
BOUNDED TIME DEFINITIONS:
    asm_tsc_read:          < 100 cycles
    asm_tsc_read_serialized: < 300 cycles
    asm_vq_submit_*:       < 200 cycles
    asm_vq_poll_*:         < 100 cycles
    asm_vq_notify:         < 1000 cycles (MMIO latency)
    asm_bar_*:             < 100 cycles
    asm_mmio_*:            < 500 cycles
```

**Source:** REDESIGN §2.2 EXEC-3

### 6.5.2 Exception: Device Reset

```
EXCEPTION PROG-2: Device Reset Timeout

asm_nic_reset() MAY block for bounded time:
    MAXIMUM: 100 milliseconds
    MECHANISM: Poll status register with TSC timeout

IMPLEMENTATION:
    Write 0 to status register
    Loop:
        Read status register
        If status == 0: success, return 0
        If TSC elapsed > 100ms: timeout, return 1
        Brief pause (rep nop)
```

### 6.5.3 Main Loop Progress

```
GUARANTEE PROG-3: Main Loop Iteration Bound

EACH main loop iteration MUST complete within:
    TARGET: 1 millisecond
    MAXIMUM: 5 milliseconds (under load)

MECHANISM:
    - RX poll budget: 32 packets max
    - TX submit budget: 32 packets max
    - smoltcp.poll(): bounded by socket count and packet count
    - App state: O(1) state machine step
```

**Source:** REDESIGN §7.1, AUDIT §2.8 LOOP-1

## 6.6 Timeout Implementation Pattern

### 6.6.1 Timeout Configuration

```rust
/// Timeout configuration derived from calibrated TSC
pub struct TimeoutConfig {
    tsc_freq: u64,
}

impl TimeoutConfig {
    pub fn new(tsc_freq: u64) -> Self {
        Self { tsc_freq }
    }

    /// Convert milliseconds to TSC ticks
    #[inline]
    pub fn ms_to_ticks(&self, ms: u64) -> u64 {
        ms * (self.tsc_freq / 1000)
    }

    // Protocol-specific timeouts
    pub fn dhcp_discover(&self) -> u64 { self.ms_to_ticks(5_000) }   // 5s
    pub fn dhcp_request(&self) -> u64 { self.ms_to_ticks(3_000) }    // 3s
    pub fn tcp_connect(&self) -> u64 { self.ms_to_ticks(30_000) }    // 30s
    pub fn dns_query(&self) -> u64 { self.ms_to_ticks(5_000) }       // 5s
    pub fn http_response(&self) -> u64 { self.ms_to_ticks(60_000) }  // 60s
}
```

### 6.6.2 Timeout Check Pattern

```rust
/// Check if timeout has elapsed (non-blocking)
#[inline]
fn is_timed_out(start_tsc: u64, timeout_ticks: u64) -> bool {
    let now = unsafe { asm_tsc_read() };
    now.wrapping_sub(start_tsc) > timeout_ticks
}

// Usage in state machine
impl DhcpState {
    fn step(&mut self, now_tsc: u64, timeouts: &TimeoutConfig) {
        match self {
            DhcpState::Discovering { start_tsc } => {
                if now_tsc.wrapping_sub(*start_tsc) >
timeouts.dhcp_discover() {
                    *self = DhcpState::Failed(Error::Timeout);
                }
                // ... check for response ...
            }
            // ...
        }
    }
}
```

## 6.7 Forbidden Patterns

### 6.7.1 Spin-Wait (FORBIDDEN)

```
// FORBIDDEN: Spin-wait loop
fn wait_for_tx_complete(state: &mut VqState) {
    loop {  // ← FORBIDDEN
        if let Some(_) = unsafe { asm_vq_poll_tx_complete(state) } {
            return;
        }
        // Spinning forever if device never completes
    }
}
```

### 6.7.2 Busy Delay (FORBIDDEN)

```
// FORBIDDEN: TSC-based busy delay
fn tsc_delay_us(us: u64) {
    let start = unsafe { asm_tsc_read() };
    let ticks = us * TSC_FREQ / 1_000_000;  // Also forbidden: hardcoded
freq
    while unsafe { asm_tsc_read() } - start < ticks {
        // Busy waiting - FORBIDDEN
    }
}
```

### 6.7.3 Hardcoded Frequencies (FORBIDDEN)

```
// FORBIDDEN: Hardcoded TSC frequency
const TSC_FREQ: u64 = 2_500_000_000;  // WRONG

// CORRECT: Use calibrated value from BootHandoff
fn get_tsc_freq(handoff: &BootHandoff) -> u64 {
    handoff.tsc_freq  // Calibrated at boot
}
```

## 6.8 Time Guarantees Summary

| ID | Guarantee | Source |
|---|---|---|
| **TIME-G1** | TSC is sole time source | Contract §6.1 |
| **TIME-G2** | Invariant TSC verified at boot | AUDIT §7.1.4 |
| **TIME-G3** | TSC frequency calibrated, never hardcoded | REDESIGN §7.6 |
| **TIME-G4** | TSC monotonic on single core | AUDIT §7.1.5 |

| ID | Guarantee | Source |
|---|---|---|
| **TIME-G5** | All ASM functions return in bounded cycles | REDESIGN §2.2 |
| **TIME-G6** | No spin-waits or busy delays | REDESIGN §1.5 |
| **TIME-G7** | Main loop iteration < 5ms | REDESIGN §7.1 |
| **TIME-G8** | Timeout checks are non-blocking | Contract §6.6 |

---

# Chunk 7 — Error Semantics & Failure Modes

## 7.1 Error Classification

Errors in this ABI are classified into two categories with fundamentally different handling requirements:

### 7.1.1 Retryable Errors

Retryable errors indicate temporary conditions that may succeed on retry.

| Error | Cause | Caller Action |
|---|---|---|
| Queue full | All descriptors in use | Retry after collecting completions |
| No RX packet | RX queue empty | Normal; poll again next iteration |
| No TX completion | Device hasn't finished | Normal; poll again next iteration |
| DHCP timeout | No DHCP response | Retry with exponential backoff |
| TCP connection refused | Remote rejected | Report to app, may retry |
| DNS timeout | No DNS response | Try fallback server |

**Caller obligation:** May retry immediately or with backoff; must not loop indefinitely.

### 7.1.2 Fatal Errors

Fatal errors indicate unrecoverable conditions. The system cannot continue.

| Error | Cause | System Response |
|---|---|---|
| No NIC found | Hardware not present | Halt |
| VirtIO feature negotiation failed | Incompatible device | Halt |
| DMA allocation failed | No memory | Halt |
| ExitBootServices failed | UEFI error | Halt |
| Device reset timeout | Hardware unresponsive | Halt |
| Invariant TSC not available | Incompatible CPU | Halt |

| Error | Cause | System Response |
|---|---|---|
| Invalid BootHandoff | Corruption | Halt |

**System obligation:** Log error (if possible), halt processor.

**Source:** AUDIT §2.9 ERR-1

# 7.2 ASM Function Error Returns

## 7.2.1 Error Return Convention

ASM functions use the following error return convention:

| Return Value | Meaning |
|---|---|
| 0 | Success |
| 1 | Retryable error (queue full, not ready, etc.) |
| 0xFFFFFFFF | Sentinel for "no result" (polling functions) |
| Other | Reserved for future error codes |

## 7.2.2 Function-Specific Error Returns

**`asm_vq_submit_tx` / `asm_vq_submit_rx`**

| Return | Meaning | Caller Action |
|---|---|---|
| 0 | Success, buffer submitted | Buffer now DEVICE-OWNED |
| 1 | Queue full | Retry after poll_complete |

```
// Usage pattern
loop {
    for _ in 0..TX_BUDGET {
        match tx_queue.pop() {
            Some(pkt) => {
                let result = unsafe { asm_vq_submit_tx(state, pkt.idx,
pkt.len) };
                if result != 0 {
                    // Queue full - put packet back, stop for this
iteration
                    tx_queue.push_front(pkt);
                    break;
                }
            }
            None => break,
        }
    }
    // Poll completions before next iteration
}
```

`asm_vq_poll_tx_complete`

| Return | Meaning | Caller Action |
|---|---|---|
| 0..0xFFFE | Buffer index of completed TX | Return buffer to pool |
| 0xFFFFFFFF | No completion available | Normal; try again later |

`asm_vq_poll_rx`

| Return | Meaning | Caller Action |
|---|---|---|
| 1 | Packet received (result populated) | Process packet |
| 0 | No packet available | Normal; try again later |

`asm_nic_reset`

| Return | Meaning | Caller Action |
|---|---|---|
| 0 | Reset successful | Continue initialization |
| 1 | Reset timeout (100ms) | FATAL: Halt |

`asm_nic_read_mac`

| Return | Meaning | Caller Action |
|---|---|---|
| 0 | MAC read successfully | Use MAC from output buffer |
| 1 | MAC not available | Generate local MAC or fail |

## 7.3 Caller Obligations

### 7.3.1 Obligation on Retryable Error

```
OBLIGATION RETRY-1: Handling Retryable Errors

ON receiving retryable error:
    1. DO NOT loop immediately retrying
    2. DO continue with other main loop phases
    3. DO retry on next main loop iteration
    4. DO track retry count for eventual timeout

RATIONALE:
    Immediate retry loops violate PROG-1 (bounded execution)
    Main loop progress ensures completions are collected
```

### 7.3.2 Obligation on Fatal Error

```
OBLIGATION FATAL-1: Handling Fatal Errors

ON receiving fatal error:
    1. DO NOT retry
    2. DO NOT continue execution
    3. DO log error if output available (serial, framebuffer)
    4. DO halt processor (cli; hlt)

IMPLEMENTATION:
    fn fatal_error(code: u32, msg: &str) -> ! {
        // Attempt to log (may fail, that's okay)
        if let Some(output) = debug_output() {
            output.write(msg);
        }
        // Halt
        loop {
            unsafe { asm!("cli; hlt"); }
        }
    }
```

### 7.3.3 Obligation on Polling "Empty"

```
OBLIGATION POLL-1: Handling Empty Poll

ON receiving "no result" from poll function:
    1. This is NORMAL operation, not an error
    2. DO continue with other main loop phases
    3. DO poll again on next iteration
    4. DO NOT log or count as error
```

## 7.4 Error Propagation

### 7.4.1 ASM → Rust Propagation

ASM communicates errors to Rust via return values only:

```
RULE ERR-PROP-1: ASM Error Returns

ASM functions:
    - Return error codes in RAX
    - Do NOT modify errno or global state
    - Do NOT panic or halt (except internal assertions)
    - Do NOT call Rust error handlers

Rust responsibility:
    - Check return value after every ASM call
```

```
    - Convert to Rust Result<T, E> if desired
    - Decide on retry vs. fatal based on context
```

### 7.4.2 Rust → smoltcp Propagation

```
RULE ERR-PROP-2: Device Trait Error Handling

smoltcp Device trait methods return Option:
    receive() -> Option<(RxToken, TxToken)>
    transmit() -> Option<TxToken>

MAPPING:
    ASM poll_rx returns 0 → receive() returns None
    ASM poll_rx returns 1 → receive() returns Some(...)
    ASM submit_tx returns 1 (full) → transmit() returns None
    ASM submit_tx returns 0 → transmit() returns Some(...)

smoltcp handles None gracefully:
    - Stops TX for this poll() call
    - Retries on next poll() call
```

## 7.5 Panic / Halt Rules

### 7.5.1 ASM Panic Behavior

```
RULE PANIC-ASM-1: ASM Must Not Panic

ASM code MUST NOT:
    - Call Rust panic machinery
    - Trigger CPU exceptions (except unrecoverable)
    - Enter infinite loops without timeout

ASM code MAY:
    - Return error codes
    - Execute hlt on truly unrecoverable conditions
    - Trigger debug breakpoint (int3) in debug builds
```

### 7.5.2 Rust Panic Behavior

```
RULE PANIC-RUST-1: Rust Panic Handling

In this no_std environment:
    - Default panic handler halts
    - No unwinding (panic = abort)
    - No recovery from panic
```

```
Panic triggers:
    - Array bounds check failure
    - Integer overflow (debug builds)
    - Explicit panic!() calls
    - Unwrap on None/Err

RECOMMENDATION:
    - Use checked operations
    - Handle all Results explicitly
    - Avoid unwrap()/expect() in critical paths
```

### 7.5.3 Halt Implementation

```
RULE HALT-1: Processor Halt Sequence

ON unrecoverable error:

    ; Disable interrupts (already disabled, but defensive)
    cli

    ; Halt loop (handles spurious wake)
    .halt_loop:
        hlt
        jmp .halt_loop

RATIONALE:
    - cli ensures no interrupt wake
    - hlt stops processor execution
    - Loop handles NMI or SMI wake
```

## 7.6 Resource Leak Prevention

### 7.6.1 Leak-Free Error Paths

```
OBLIGATION LEAK-1: No Resource Leaks on Error

ON any error path, ensure:
    [ ] All DRIVER-OWNED buffers either:
        - Freed to pool, OR
        - Submitted to queue (becoming DEVICE-OWNED)
    [ ] All descriptors either:
        - In free list, OR
        - In available ring (pending)
    [ ] No dangling pointers stored

MECHANISM:
    - Use RAII patterns in Rust (Drop trait)
    - Track all allocations
    - Error paths must clean up
```

**Source:** AUDIT §2.9 ERR-2

### 7.6.2 Example: TX Error Recovery

```
fn try_transmit(driver: &mut VirtioDriver, packet: &[u8]) -> Result<(),
TxError> {
    // Allocate buffer (may fail)
    let buffer_idx = driver.alloc_tx_buffer()
        .ok_or(TxError::NoBuffer)?;

    // At this point, buffer is DRIVER-OWNED and we own it

    // Prepare buffer
    let buffer = driver.get_tx_buffer(buffer_idx);
    buffer[..12].fill(0);  // VirtIO header
    buffer[12..12+packet.len()].copy_from_slice(packet);

    // Submit (may fail)
    let result = unsafe {
        asm_vq_submit_tx(driver.tx_state, buffer_idx, (12 + packet.len())
as u16)
    };

    if result != 0 {
        // Queue full - we still own the buffer, must free it
        driver.free_tx_buffer(buffer_idx);
        return Err(TxError::QueueFull);
    }

    // Success - buffer is now DEVICE-OWNED
    Ok(())
}
```

## 7.7 Timeout as Error

### 7.7.1 Timeout Detection

```
RULE TIMEOUT-1: Timeout Is Retryable Error

Protocol timeouts (DHCP, TCP, DNS) are retryable errors:
    - State machine transitions to "Failed" or "Retry" state
    - May retry with backoff
    - Eventually becomes fatal if max retries exceeded

Hardware timeouts (device reset) are fatal errors:
    - Device is non-responsive
    - Cannot continue
    - Halt system
```

### 7.7.2 Timeout Backoff

```rust
/// Exponential backoff for retryable timeouts
struct RetryState {
    attempts: u32,
    max_attempts: u32,
    base_timeout_ms: u64,
    max_timeout_ms: u64,
}

impl RetryState {
    fn next_timeout(&mut self, timeouts: &TimeoutConfig) -> Option<u64> {
        if self.attempts >= self.max_attempts {
            return None;  // Give up
        }

        // Exponential backoff: base * 2^attempts, capped at max
        let timeout_ms = self.base_timeout_ms
            .saturating_mul(1 << self.attempts)
            .min(self.max_timeout_ms);

        self.attempts += 1;
        Some(timeouts.ms_to_ticks(timeout_ms))
    }
}
```

## 7.8 Error Codes Summary

| Code | Name | Category | Meaning |
|---|---|---|---|
| 0 | SUCCESS | — | Operation completed |
| 1 | QUEUE_FULL | Retryable | Virtqueue has no free descriptors |
| 1 | TIMEOUT | Retryable/Fatal | Operation timed out |
| 1 | NOT_AVAILABLE | Retryable | Resource not ready |
| 0xFFFFFFFF | NO_RESULT | — | Poll found nothing (normal) |

## 7.9 Error Invariants Summary

| ID | Invariant | Consequence of Violation |
|---|---|---|
| **ERR-INV-1** | Retryable errors must not cause infinite retry loops | System hang |
| **ERR-INV-2** | Fatal errors must halt system | Continued corruption |
| **ERR-INV-3** | All error paths must free owned resources | Memory/descriptor leak |
| **ERR-INV-4** | ASM returns error codes, does not halt | Rust loses control |

| ID | Invariant | Consequence of Violation |
|---|---|---|
| **ERR-INV-5** | Rust checks all ASM return values | Silent failures |

# Chunk 8 — Safety, Undefined Behavior & Misuse

## 8.1 Undefined Behavior Definition

**Undefined Behavior (UB)** in this contract means the system may exhibit any behavior, including but not limited to:

- Data corruption
- Security vulnerabilities
- System hang
- Incorrect results
- Apparently correct operation (most dangerous)

**Philosophy:** This contract adopts a "hostile misuse" stance. Any deviation from the contract is UB, even if it "seems to work" in testing.

## 8.2 Explicit UB Conditions

### 8.2.1 Calling Convention Violations

| Violation | Why It's UB |
|---|---|
| Wrong parameter registers | Function receives garbage values |
| Missing shadow space | Stack corruption |
| Misaligned stack | Potential crash, corruption |
| Wrong return register | Caller receives garbage |
| Clobbering non-volatile registers | Caller state destroyed |
| Non-zero direction flag | String operations corrupted |

### 8.2.2 Ownership Violations

| Violation | Why It's UB |
|---|---|
| Reading DEVICE-OWNED buffer | Data race with DMA engine |
| Writing DEVICE-OWNED buffer | Data race with DMA engine |
| Double-free of buffer | Allocator corruption |
| Use-after-free | Dangling pointer access |
| Submitting FREE buffer | Invalid descriptor |

| Violation | Why It's UB |
|---|---|
| Submitting already-submitted buffer | Descriptor reuse conflict |

### 8.2.3 Memory Ordering Violations

| Violation | Why It's UB |
|---|---|
| Missing sfence before avail.idx | Device sees stale descriptors |
| Missing mfence before notify | Notify races with data |
| Missing lfence after used.idx | Reading stale completion |
| Compiler reordering of DMA access | Device sees wrong data |
| Using WB memory without clflush | Cache incoherence |

### 8.2.4 Pointer Violations

| Violation | Why It's UB |
|---|---|
| Null pointer to ASM function | Dereferencing null |
| Misaligned pointer | Alignment fault |
| Out-of-bounds pointer | Memory corruption |
| Pointer to freed memory | Use-after-free |
| Pointer outside DMA region | Invalid DMA address |

### 8.2.5 State Violations

| Violation | Why It's UB |
|---|---|
| Using NIC before initialization | Hardware in unknown state |
| Using queues before setup | Invalid ring pointers |
| Calling EBS functions post-EBS | Firmware not available |
| Enabling interrupts | No handler installed |
| Running on AP core | TSC unsynchronized, races |

### 8.2.6 Timing Violations

| Violation | Why It's UB |
|---|---|
| Hardcoded TSC frequency | Wrong timeout calculations |
| Using TSC without invariant check | Frequency may vary |
| Infinite spin-wait | System hang |

| Violation | Why It's UB |
|---|---|
| Main loop > 5ms | Protocol timeouts, poor performance |

## 8.3 Rust `unsafe` Boundaries

### 8.3.1 Required `unsafe` Blocks

All ASM function calls require `unsafe`:

```
// ALL ASM calls must be in unsafe blocks
let tsc = unsafe { asm_tsc_read() };

let result = unsafe {
    asm_vq_submit_tx(state, idx, len)
};

unsafe {
    asm_bar_mfence();
}
```

**Rationale:** The compiler cannot verify:

- Pointer validity
- Ownership rules
- Memory ordering
- State preconditions

### 8.3.2 `unsafe` Contract Documentation

Each `unsafe` block SHOULD document what invariants the caller guarantees:

```
// SAFETY:
// - state points to valid, initialized VirtqueueState
// - buffer_idx < queue_size
// - Buffer at buffer_idx is DRIVER-OWNED
// - Buffer content fully written (including 12-byte header)
let result = unsafe {
    asm_vq_submit_tx(state, buffer_idx, length)
};
```

### 8.3.3 `unsafe` Trait Implementations

The smoltcp Device trait implementation requires unsafe operations:

```
impl smoltcp::phy::Device for VirtioDevice {
    type RxToken<'a> = VirtioRxToken<'a>;
```

```
    type TxToken<'a> = VirtioTxToken<'a>;

    fn receive(&mut self, _timestamp: Instant) ->
Option<(Self::RxToken<'_>, Self::TxToken<'_>)> {
        let mut result = RxResult::default();

        // SAFETY:
        // - self.rx_state is valid VirtqueueState
        // - result is valid output buffer
        let status = unsafe {
            asm_vq_poll_rx(&mut self.rx_state, &mut result)
        };

        if status == 0 {
            return None;
        }

        // Buffer at result.buffer_index is now DRIVER-OWNED
        Some((
            VirtioRxToken {
                device: self,
                buffer_idx: result.buffer_index,
                length: result.length,
            },
            VirtioTxToken { device: self },
        ))
    }

    // ... transmit() similarly
}
```

## 8.4 Contract Violation Consequences

### 8.4.1 Severity Levels

| Level | Example | Consequence |
|---|---|---|
| **Catastrophic** | DMA to wrong address | System corruption, potential security hole |
| **Severe** | Ownership violation | Data corruption, packet loss |
| **Moderate** | Missing barrier | Intermittent corruption |
| **Minor** | Wrong error handling | Suboptimal behavior |

### 8.4.2 Detection Difficulty

| Violation Type | Detection | Notes |
|---|---|---|
| Calling convention | Immediate crash or corruption | Usually obvious |
| Ownership | Intermittent corruption | Hard to reproduce |

| Violation Type | Detection | Notes |
|---|---|---|
| Memory ordering | Rare corruption | May only appear under load |
| Pointer validity | Crash or silent corruption | May work in testing |
| Timing | Timeouts, performance issues | May seem like network issues |

### 8.4.3 QEMU vs. Real Hardware

**Warning:** Many violations that "work" in QEMU will fail on real hardware:

| Violation | QEMU Behavior | Real Hardware |
|---|---|---|
| Missing barriers | Usually works | Corruption |
| WB memory for DMA | Works (coherent emulation) | Corruption |
| Timing assumptions | May work | Fails under load |
| Ownership races | Hidden by synchronous emulation | Data corruption |

## 8.5 Defensive Programming Requirements

### 8.5.1 Pointer Validation (Debug Builds)

```
#[cfg(debug_assertions)]
fn validate_vq_state(state: *const VirtqueueState) {
    assert!(!state.is_null(), "VirtqueueState pointer is null");
    assert!(state as usize % 64 == 0, "VirtqueueState not cache-aligned");

    let state_ref = unsafe { &*state };
    assert!(state_ref.queue_size.is_power_of_two(), "Queue size not power
of 2");
    assert!(state_ref.queue_size <= 32768, "Queue size exceeds max");
}
```

### 8.5.2 Bounds Checking

```
fn get_buffer(&self, idx: u16) -> &mut [u8] {
    assert!((idx as usize) < self.buffer_count, "Buffer index out of
bounds");
    // ...
}

fn submit_tx(&mut self, idx: u16, len: u16) -> Result<(), Error> {
    if idx as usize >= self.queue_size as usize {
        return Err(Error::InvalidIndex);
    }
    if len as usize > self.buffer_size {
        return Err(Error::BufferTooLarge);
```

```
    }
    // ...
}
```

### 8.5.3 State Assertions

```
impl VirtioDriver {
    fn assert_initialized(&self) {
        debug_assert!(self.initialized, "Driver used before
initialization");
        debug_assert!(self.status & STATUS_DRIVER_OK != 0, "Device not
ready");
    }

    pub fn submit_tx(&mut self, ...) -> Result<...> {
        self.assert_initialized();
        // ...
    }
}
```

## 8.6 Misuse Scenarios

### 8.6.1 Scenario: Accessing DEVICE-OWNED Buffer

```
// MISUSE: Reading buffer after submit
let idx = driver.alloc_tx_buffer();
let buffer = driver.get_buffer(idx);
buffer.copy_from_slice(&packet);

unsafe { asm_vq_submit_tx(state, idx, len) };

// BUG: Buffer is now DEVICE-OWNED
let first_byte = buffer[0];  // UB! Data race with DMA
```

**Consequence:** If device is simultaneously transmitting, read may see partial/corrupted data.

### 8.6.2 Scenario: Missing Memory Barrier

```
// MISUSE: Manual descriptor update without barriers
unsafe {
    (*desc).addr = buffer_addr;
    (*desc).len = packet_len;
    (*desc).flags = 0;

    // BUG: Missing sfence
```

```
    (*avail).ring[idx] = desc_idx;
    (*avail).idx += 1;

    // BUG: Missing mfence

    asm_mmio_write16(notify_addr, queue_idx);
}
```

**Consequence:** Device may see stale descriptor, transmit garbage or crash.

### 8.6.3 Scenario: Spin-Wait in Callback

```
impl TxToken for VirtioTxToken<'_> {
    fn consume<R, F>(self, len: usize, f: F) -> R
    where F: FnOnce(&mut [u8]) -> R
    {
        let idx = self.driver.alloc_tx_buffer();
        let buffer = self.driver.get_buffer(idx);
        buffer[..12].fill(0);
        let result = f(&mut buffer[12..12+len]);

        unsafe { asm_vq_submit_tx(self.driver.tx_state, idx, (12+len) as
u16) };

        // BUG: Waiting for completion inside callback
        loop {
            if unsafe { asm_vq_poll_tx_complete(self.driver.tx_state) } !=
0xFFFFFFFF {
                break;
            }
        }  // Infinite loop if device slow/stuck

        result
    }
}
```

**Consequence:** System hangs, violates progress guarantee.

### 8.6.4 Scenario: Hardcoded TSC Frequency

```
// MISUSE: Hardcoded frequency
const TSC_FREQ: u64 = 2_500_000_000;  // "It's always 2.5GHz, right?"

fn timeout_5_seconds() -> u64 {
    5 * TSC_FREQ  // WRONG on any other CPU
}
```

**Consequence:** On 3.5GHz CPU, timeout is 3.5 seconds. On 2.0GHz CPU, timeout is 6.25 seconds.

## 8.7 Formal Precondition/Postcondition Summary

### 8.7.1 `asm_vq_submit_tx`

```
PRECONDITIONS:
    P1: vq_state != NULL
    P2: vq_state points to valid, initialized VirtqueueState
    P3: buffer_index < vq_state.queue_size
    P4: Buffer[buffer_index] is DRIVER-OWNED
    P5: Buffer content is fully written
    P6: buffer_length ≤ vq_state.buffer_size
    P7: buffer_length ≥ 12 (minimum VirtIO header)

POSTCONDITIONS (if return == 0):
    Q1: Buffer[buffer_index] is DEVICE-OWNED
    Q2: Descriptor submitted to available ring
    Q3: Device notified (if appropriate)

POSTCONDITIONS (if return == 1):
    Q4: Buffer[buffer_index] is still DRIVER-OWNED
    Q5: No state changed
```

### 8.7.2 `asm_vq_poll_rx`

```
PRECONDITIONS:
    P1: vq_state != NULL
    P2: vq_state points to valid, initialized VirtqueueState
    P3: result != NULL
    P4: result points to valid RxResult

POSTCONDITIONS (if return == 1):
    Q1: result.buffer_index contains valid index
    Q2: Buffer[result.buffer_index] is DRIVER-OWNED
    Q3: Buffer contains packet data (12B header + frame)
    Q4: result.length contains actual bytes

POSTCONDITIONS (if return == 0):
    Q5: result contents are undefined
    Q6: No buffers changed ownership
```

## 8.8 Safety Invariants Summary

| ID | Invariant | Category |
|---|---|---|
| **SAFE-1** | All ASM calls in `unsafe` blocks | Rust safety |
| **SAFE-2** | Pointer validity documented in `// SAFETY:` | Documentation |
| **SAFE-3** | No access to DEVICE-OWNED buffers | Ownership |

| ID | Invariant | Category |
|---|---|---|
| **SAFE-4** | All barriers placed per contract | Memory ordering |
| **SAFE-5** | No spin-waits or unbounded loops | Progress |
| **SAFE-6** | TSC frequency from calibration only | Timing |
| **SAFE-7** | All return values checked | Error handling |
| **SAFE-8** | Debug assertions for invariants | Defensive |

---

# Chunk 9 — Final Contract Review & Freeze Statement

## 9.1 Internal Consistency Verification

This section verifies that all chunks of this contract are internally consistent and do not contradict each other.

### 9.1.1 Cross-Chunk Consistency Matrix

| Chunk | References | Consistent With | Verified |
|---|---|---|---|
| 1. Scope | REDESIGN, AUDIT | Chunks 2-8 | ✓ |
| 2. ABI | Intel SDM, MS ABI | Chunk 3 (interface) | ✓ |
| 3. Interface | Chunk 2 (ABI), Chunk 4 (ownership) | Chunks 5, 6 | ✓ |
| 4. Ownership | REDESIGN §3.3, AUDIT §2.1 | Chunk 3, 5 | ✓ |
| 5. Ordering | AUDIT §5.4, Intel SDM | Chunk 3, 4 | ✓ |
| 6. Time | REDESIGN §7, AUDIT §2.5 | Chunk 1, 3 | ✓ |
| 7. Errors | AUDIT §2.9 | Chunk 3, 4, 8 | ✓ |
| 8. Safety | All chunks | Chunks 1-7 | ✓ |

### 9.1.2 Invariant Consistency

All invariants defined across chunks have been checked for:

- No contradictions between invariants
- No impossible-to-satisfy combinations
- Complete coverage of the ABI surface

**Invariant Count Summary:**

| Category | Count |
|---|---|
| Execution (EXEC-INV-*) | 5 |
| Memory (MEM-INV-*) | 6 |

| Category | Count |
|---|---|
| Timing (TIME-INV-*) | 4 |
| VirtIO (VIO-INV-*) | 6 |
| Barrier (BAR-INV-*) | 4 |
| Lifetime (LIFE-*) | 6 |
| Time Guarantee (TIME-G*) | 8 |
| Error (ERR-INV-*) | 5 |
| Safety (SAFE-*) | 8 |
| **Total** | **52** |

### 9.1.3 Source Document Traceability

Every normative statement in this contract traces to:

- NETWORK_STACK_REDESIGN.md (v1.0 Frozen), OR
- NETWORK_STACK_AUDIT.md (v1.1), OR
- Intel SDM / VirtIO Spec / UEFI Spec (for hardware/protocol details)

**No statement introduces new architectural requirements not present in source documents.**

## 9.2 Completeness Checklist

### 9.2.1 ABI Coverage

| Aspect | Documented | Section |
|---|---|---|
| Calling convention | ✓ | Chunk 2 |
| Parameter passing | ✓ | §2.1.1 |
| Return values | ✓ | §2.1.2 |
| Register preservation | ✓ | §2.2 |
| Stack requirements | ✓ | §2.1.3, §2.1.4 |
| Clobber rules | ✓ | §2.4 |

### 9.2.2 Interface Coverage

| Function | Documented | Section |
|---|---|---|
| asm_tsc_read | ✓ | §3.3.1 |
| asm_tsc_read_serialized | ✓ | §3.3.1 |
| asm_vq_submit_tx | ✓ | §3.3.2 |

| Function | Documented | Section |
|---|---|---|
| asm_vq_poll_tx_complete | ✓ | §3.3.2 |
| asm_vq_submit_rx | ✓ | §3.3.2 |
| asm_vq_poll_rx | ✓ | §3.3.2 |
| asm_vq_notify | ✓ | §3.3.2 |
| asm_bar_sfence | ✓ | §3.3.3 |
| asm_bar_lfence | ✓ | §3.3.3 |
| asm_bar_mfence | ✓ | §3.3.3 |
| asm_mmio_read32 | ✓ | §3.3.4 |
| asm_mmio_write32 | ✓ | §3.3.4 |
| asm_mmio_read16 | ✓ | §3.3.4 |
| asm_mmio_write16 | ✓ | §3.3.4 |
| asm_nic_reset | ✓ | §3.3.5 |
| asm_nic_set_status | ✓ | §3.3.5 |
| asm_nic_get_status | ✓ | §3.3.5 |
| asm_nic_read_features | ✓ | §3.3.5 |
| asm_nic_write_features | ✓ | §3.3.5 |
| asm_nic_read_mac | ✓ | §3.3.5 |

### 9.2.3 Semantic Coverage

| Aspect | Documented | Section |
|---|---|---|
| Ownership states | ✓ | Chunk 4 |
| Ownership transitions | ✓ | §4.3 |
| Buffer lifecycles | ✓ | §4.4, §4.5 |
| Memory barriers | ✓ | Chunk 5 |
| Barrier placement | ✓ | §5.3 |
| Visibility guarantees | ✓ | §5.5 |
| Time source | ✓ | Chunk 6 |
| Progress guarantees | ✓ | §6.5 |
| Error classification | ✓ | Chunk 7 |
| Error returns | ✓ | §7.2 |

| Aspect | Documented | Section |
|---|---|---|
| UB conditions | ✓ | Chunk 8 |
| Safety boundaries | ✓ | §8.3 |

## 9.3 Known Limitations

This contract acknowledges the following limitations:

### 9.3.1 Scope Limitations

| Limitation | Rationale |
|---|---|
| VirtIO only | Other NICs require separate contracts |
| Single-core only | Multi-core requires synchronization extensions |
| No interrupts | Interrupt support requires IDT, ISR contracts |
| IPv4 only | IPv6 requires additional protocol work |
| No TLS | Cryptography out of scope for v1 |

### 9.3.2 Implementation Guidance Not Provided

This contract does NOT specify:

- Internal ASM register allocation
- Internal Rust data structure layouts (beyond shared structs)
- Optimization strategies
- Debug/logging mechanisms
- Test methodologies

These are implementation concerns, not ABI concerns.

## 9.4 Versioning Policy

### 9.4.1 Version Number Scheme

```
Version: MAJOR.MINOR

MAJOR: Incremented when:
    - Any breaking change to function signatures
    - Any breaking change to struct layouts
    - Any change to calling convention
    - Any change that requires both ASM and Rust changes

MINOR: Incremented when:
    - New functions added (backward compatible)
    - Documentation clarifications
    - Bug fixes that don't change interface
```

### 9.4.2 Compatibility Rules

```
RULE COMPAT-1: Major Version Compatibility

Code compiled against contract version X.* is INCOMPATIBLE with
code compiled against version Y.* where X ≠ Y.

RULE COMPAT-2: Minor Version Compatibility

Code compiled against contract version X.A is FORWARD COMPATIBLE with
code compiled against version X.B where B > A.
(New features may not be available, but existing features work.)

RULE COMPAT-3: Linking Requirement

ASM object files and Rust object files MUST be compiled against
the same major version of this contract.
```

### 9.4.3 Deprecation Policy

```
POLICY DEP-1: Function Deprecation

Functions may be deprecated but not removed within a major version.
Deprecation is indicated by:
    - Documentation note
    - Compiler warning (if tooling supports)

Deprecated functions are removed in the next major version.
```

## 9.5 Contract Freeze Declaration

### 9.5.1 Freeze Statement

```
╔═══════════════════════════════════════════════════════════════════════════════
═══╗
║
║
║                    CONTRACT FREEZE DECLARATION
║
║
║
║  Document:    NETWORK_ASM_RUST_ABI_CONTRACT.md
║
║  Version:     1.0
║
║  Date:        2026-01-09
```

```
╠═══════════════════════════════════════════════════════════════════════════════
╠═══╣
╠   Status:      FROZEN
╠
╠
╠   This document represents the frozen v1.0 ABI contract between ASM and
╠
╠   Rust for the MorpheusX post-ExitBootServices networking runtime.
╠
╠
╠   All implementations MUST conform to this contract.
╠
╠   Any deviation constitutes undefined behavior.
╠
╠
╠   Modifications to this contract require:
╠
╠     1. Formal review against source documents
╠
╠     2. Impact analysis on existing implementations
╠
╠     3. Version increment per §9.4
╠
╠     4. Explicit sign-off
╠
╠
╠
╚═══════════════════════════════════════════════════════════════════════════════
═══╝
```

### 9.5.2 Authority Chain

```
This contract is authoritative for:
    → ASM ↔ Rust interface behavior

This contract derives authority from:
    → NETWORK_STACK_REDESIGN.md (v1.0 Frozen)
    → NETWORK_STACK_AUDIT.md (v1.1)

These documents derive authority from:
    → VirtIO Specification 1.2 (OASIS)
    → UEFI Specification 2.10
    → Intel 64 and IA-32 Architectures SDM
    → Microsoft x64 ABI documentation
```

### 9.5.3 Conformance Statement

An implementation conforms to this contract if and only if:

1. All ASM functions match the signatures in Chunk 3
2. All ASM functions implement the specified behavior
3. All ASM functions place barriers as specified in Chunk 5
4. All Rust code calls ASM functions with valid preconditions
5. All Rust code handles return values per Chunk 7
6. All ownership rules from Chunk 4 are respected
7. All timing constraints from Chunk 6 are met
8. No undefined behavior conditions from Chunk 8 are triggered

## 9.6 Future Work (Out of Scope)

The following are explicitly deferred to future contract versions:

| Item | Target Version | Notes |
|---|---|---|
| Multi-core support | v2.0 | Requires synchronization primitives |
| Interrupt support | v2.0 | Requires IDT, ISR interface |
| Additional NIC drivers | v1.x | Additive, minor version |
| Hardware offload support | v2.0 | Changes buffer semantics |
| IPv6 support | v1.x | smoltcp supports, additive |

## 9.7 Document End

---

```
═══════════════════════════════════════════════════════════════════════════
                        END OF CONTRACT DOCUMENT

  Document:    MorpheusX ASM ↔ Rust ABI Contract
  Version:     1.0 (FROZEN)
  Date:        2026-01-09
  Chunks:      9 of 9 complete

  Source Documents:
    - NETWORK_STACK_REDESIGN.md v1.0 (Frozen)
    - NETWORK_STACK_AUDIT.md v1.1

  Next Steps (NOT part of this document):
    - Binding generation (extern "win64")
    - Rust unsafe annotation pass
    - ASM enforcement scaffolding

═══════════════════════════════════════════════════════════════════════════
```