MorpheusX Blocking Patterns Refactor Guide

Core Problem

MorpheusX is a **single-threaded, cooperative, poll-driven system** with:

- No scheduler
- No preemption
- No interrupts relied upon
- No async runtime

Fundamental invariant: Every function must return in bounded time. Progress happens only across calls, never within them.

Any function that waits is a **dead system**.

Violations Catalog

Category 1: `tsc_delay_us()` - Busy-Wait Primitive

The function itself is a violation. It spins CPU cycles waiting for time to pass.

```
// network/src/device/pci.rs:248
pub fn tsc_delay_us(us: u32) {
    while unsafe { read_tsc() }.wrapping_sub(start) < cycles {
        core::hint::spin_loop();  // ❌ BLOCKING
    }
}
```

All call sites are violations:

File	Line	Context	Severity
`network/src/device/virtio.rs`	347	TX completion poll loop	CRITICAL
`network/src/client/native.rs`	202	`wait_for_network()`	CRITICAL
`network/src/client/native.rs`	261	DNS resolution loop	CRITICAL
`network/src/client/native.rs`	352	TCP connect wait	CRITICAL
`network/src/client/native.rs`	380	`send_all()` loop	CRITICAL
`network/src/client/native.rs`	407	`recv()` loop	CRITICAL
`core/src/net/init.rs`	205	DHCP wait loop	CRITICAL
`bootloader/.../network_check.rs`	109	1.5s delay	HIGH
`bootloader/.../network_init.rs`	109	1.5s delay	HIGH

File	Line	Context	Severity
`bootloader/.../ui.rs`	509	500ms delay	HIGH
`bootloader/.../ui.rs`	797	15s delay	HIGH

Resolution: Delete `tsc_delay_us()`. Replace all uses with state machine patterns.

Category 2: Blocking Loops in `NativeHttpClient`

These functions block until completion or timeout:

`wait_for_network()` - Line 192

```
while !self.iface.has_ip() {      // ✖ LOOP
    self.poll();
    if timeout { return Err(); }
    tsc_delay_us(1000);           // ✖ BUSY-WAIT
}
```

Refactor to:

```
pub fn poll_network_ready(&mut self) -> PollResult<()> {
    self.poll();
    if self.iface.has_ip() { return PollResult::Ready(()); }
    PollResult::Pending
}
// Caller checks timeout at step boundary
```

`resolve_host()` - Line 247

```
loop {                            // ✖ LOOP
    self.iface.poll(now);
    match self.iface.get_dns_result(handle) {
        Ok(Some(ip)) => return Ok(ip);
        Ok(None) => {
            if timeout { return Err(); }
            tsc_delay_us(1000);   // ✖ BUSY-WAIT
        }
    }
}
```

`wait_for_connection()` - Line 335

```
while !self.iface.tcp_is_connected(handle) {  // ✖ LOOP
    self.poll();
    if timeout { return Err(); }
    tsc_delay_us(1000);            // ✖ BUSY-WAIT
}
```

`send_all()` - Line 362

```
while sent < data.len() {          // ✖ LOOP
    self.poll();
    if can_send { sent += send(); }
    if timeout { return Err(); }
    tsc_delay_us(100);             // ✖ BUSY-WAIT
}
```

`recv()` - Line 391

```
loop {                             // ✖ LOOP
    self.poll();
    if can_recv { return recv(); }
    if timeout { return Err(); }
    tsc_delay_us(1000);            // ✖ BUSY-WAIT
}
```

`read_headers()` - Line 420

```
loop {                             // ✖ LOOP
    let n = self.recv(&mut buffer)?;  // Calls blocking recv()
    if found_header_end { return; }
}
```

`read_remaining_body()` - Line 461

```
loop {                             // ✖ LOOP
    match self.recv(&mut buffer) { ... }  // Calls blocking recv()
}
```

`stream_response_body()` - Line 503

```
loop {                            // ✖ LOOP
    match self.recv(&mut buffer) { ... }  // Calls blocking recv()
}
```

`do_request_with_redirects()` - Line 549

```
loop {                            // ✖ LOOP
    let response = self.do_request(&request)?;  // Calls multiple blockers
    if !redirect { return; }
}
```

Category 3: VirtIO Transmit Blocking

Location: `network/src/device/virtio.rs:322`

```
fn transmit(&mut self, packet: &[u8]) -> Result<()> {
    let token = self.inner.transmit_begin(&tx_buf)?;

    loop {                                    // ✖ LOOP
        if let Some(t) = self.inner.poll_transmit() {
            if t == token { return Ok(()); }
        }
        if timeout { return Err(); }
        tsc_delay_us(10);                     // ✖ BUSY-WAIT
    }
}
```

Why this exists: VirtIO TX is async. Buffer submitted → device processes → completion notification.

Why blocking is wrong here: smoltcp's `TxToken::consume()` calls this synchronously.

Resolution options:

1. **Fire-and-forget**: Since we use `vec![]` for each TX, we don't need to wait for completion. Submit and return immediately. Completions are collected opportunistically.
2. **Track pending**: Maintain pending TX list, poll completions in main step function.

Category 4: DHCP Wait Loop in init.rs

Location: `core/src/net/init.rs:200`

```
loop {
    client.poll();
    poll_display();
```

```
        tsc_delay_us(1000);                     // ✖ BUSY-WAIT

        if client.ip_address().is_some() { break; }
        if timeout { return Err(DhcpTimeout); }
    }
```

Why this exists: Network init wants to block until DHCP completes.

Resolution: Return `NetworkInitResult` with state, let caller poll:

```
pub enum NetworkInitState {
    Initializing,
    WaitingForDhcp { client: NativeHttpClient, start_time: u64 },
    Ready(NetworkInitResult),
    Failed(NetInitError),
}

pub fn poll_init(state: &mut NetworkInitState, now: u64) -> bool {
    match state {
        WaitingForDhcp { client, start_time } => {
            client.poll();
            if client.ip_address().is_some() {
                *state = Ready(...);
                return true;
            }
            if now - *start_time > TIMEOUT {
                *state = Failed(DhcpTimeout);
                return true;
            }
            false  // Still pending
        }
        ...
    }
}
```

Category 5: Serial Output Busy-Wait

Location: `network/src/lib.rs:113` (added during debugging)

```
loop {
    let status = port_read(0x3fd);
    if status & 0x20 != 0 { break; }  // ✖ UNBOUNDED
}
```

Already partially fixed with retry limit. But should be non-blocking entirely:

```
// Try once, skip if port not ready
if port_read(0x3fd) & 0x20 != 0 {
    port_write(0x3f8, byte);
}
```

Correct Patterns

Pattern 1: State Machine Step

```
enum HttpState {
    Idle,
    Resolving { host: String, start: u64 },
    Connecting { ip: Ipv4Addr, port: u16, start: u64 },
    SendingRequest { sent: usize, start: u64 },
    ReadingHeaders { buffer: Vec<u8>, start: u64 },
    ReadingBody { ... },
    Done(Response),
    Failed(Error),
}

impl HttpClient {
    /// Advance state by one step. Returns true if terminal state reached.
    pub fn step(&mut self, now: u64) -> bool {
        match &mut self.state {
            HttpState::Resolving { host, start } => {
                self.iface.poll(now);
                if let Some(ip) = self.dns_result() {
                    self.state = HttpState::Connecting { ip, port: 80,
start: now };
                } else if now - *start > DNS_TIMEOUT {
                    self.state = HttpState::Failed(Timeout);
                }
                false
            }
            HttpState::Done(_) | HttpState::Failed(_) => true,
            ...
        }
    }
}
```

Pattern 2: Timeout as Observation

```
// ✖ WRONG - blocks
while condition {
    if elapsed > timeout { return Err(Timeout); }
    delay(1ms);
}
```

```
// ✅ CORRECT - observes
pub fn step(&mut self, now: u64) -> StepResult {
    if now - self.start > self.timeout {
        return StepResult::Timeout;
    }
    if condition_met() {
        return StepResult::Done;
    }
    StepResult::Pending
}
```

Pattern 3: Fire-and-Forget TX

```
fn transmit(&mut self, packet: &[u8]) -> Result<()> {
    if !self.inner.can_send() {
        // Collect any completed transmits first
        while let Some(_) = self.inner.poll_transmit() {}
        if !self.inner.can_send() {
            return Err(QueueFull);
        }
    }

    let tx_buf = vec![...];  // Owned buffer
    self.inner.transmit_begin(&tx_buf)?;

    // Don't wait for completion - buffer is owned by device until complete
    // Completion will be collected on next transmit or explicit poll
    self.pending_tx.push(tx_buf);
    Ok(())
}
```

Refactor Priority

1. **CRITICAL**: `VirtioNetDevice::transmit()` - This is called by smoltcp on every TX
2. **CRITICAL**: `NativeHttpClient` blocking methods - All network operations use these
3. **HIGH**: `NetworkInit::initialize_with_poll()` DHCP loop
4. **MEDIUM**: Delete `tsc_delay_us()` entirely
5. **LOW**: Bootloader UI delays (can be converted to frame-based timing)

Testing Strategy

After refactor:

1. Top-level loop calls `net.step()` once per iteration
2. Log state transitions
3. Verify DHCP completes within expected polls (not wall-time)

4. Verify TX never blocks main loop for more than one device poll
5. Verify HTTP request completes via state machine progression

---

Summary

What	Why	How
Delete `tsc_delay_us()`	Busy-wait kills cooperative model	Remove, refactor callers
Split blocking methods into state machines	Functions must return in bounded time	`step()` pattern
Fire-and-forget TX	TX completion can be async	Track pending, poll opportunistically
Timeout = observation	Time informs, doesn't control	Check at step boundary

Invariant to enforce: No function may contain a loop that depends on external state changing. All loops must be bounded by input size only.