Chronos OS: Phases 13 & 14 Detailed Report

Implementation of the TCP/IP Protocol Stack (ARP, IPv4, DHCP)

Development Log

January 20, 2026

Abstract

Following the initialization of the RTL8139 network controller, Chronos OS required a software abstraction layer to interpret raw binary signals. This report details the implementation of a functional Network Stack from first principles. We address the challenges of handling Network Byte Order (Big Endian) on x86 architecture, the strict memory alignment requirements of network packets, and the implementation of the Internet Checksum algorithm. The stack successfully negotiates hardware addresses via ARP and acquires an IPv4 identity via DHCP.

Contents

1 Architectural Challenges

1.1 Memory Alignment and Padding

Network protocols define strict byte-offsets for packet fields. However, the Rust compiler (and C compilers) typically add "padding" bytes between struct fields to align data to CPU word boundaries for performance.

The Problem: A standard struct definition for a header would introduce gaps, causing the OS to read invalid data from the network buffer.
The Solution: We utilized the `#[repr(packed)]` attribute on all protocol structs. This forces the compiler to arrange fields contiguously in memory, matching the wire format exactly.

1.2 Endianness (Byte Order)

Network protocols standardize on **Big Endian** (Most Significant Byte first). The x86_64 architecture is **Little Endian** (Least Significant Byte first).

To read multi-byte fields (like `ethertype` or `ip_length`) correctly, we implemented the `ntohs` (Network to Host Short) helper function:

$$\text{Value}_{LE} = ((\text{Value}_{BE} \& 0xFF) \ll 8) \mid ((\text{Value}_{BE} \& 0xFF00) \gg 8)$$

2 Phase 13: Address Resolution Protocol (ARP)

2.1 Protocol Overview

ARP allows the OS to map a known IP address to an unknown physical MAC address.

- **Hardware Type:** 1 (Ethernet)
- **Protocol Type:** 0x0800 (IPv4)
- **Opcode:** 1 (Request), 2 (Reply)

2.2 Implementation

When the network driver receives a packet with EtherType `0x0806`, it dispatches the buffer to the ARP handler. We utilize ‘unsafe‘ pointer casting to overlay the ‘ArpPacket‘ struct onto the raw memory buffer.

Verification: Upon broadcasting a request for the Gateway IP (`10.0.2.2`), the OS successfully intercepted an ARP Reply, extracted the Gateway’s MAC address, and logged it to the console.

3 Phase 14: Dynamic Host Configuration (DHCP)

3.1 Packet Composition (The Onion Model)

Sending a DHCP Request requires encapsulating headers within headers. We constructed a 272-byte buffer structured as follows:

1. **Ethernet Header (14 bytes):** Dest: `FF:FF:FF:FF:FF:FF` (Broadcast).
2. **IPv4 Header (20 bytes):** Source: `0.0.0.0`, Dest: `255.255.255.255`.

3. **UDP Header (8 bytes):** Src Port: 68 (Client), Dest Port: 67 (Server).
4. **DHCP Payload (230+ bytes):** The actual request logic.

3.2 The Internet Checksum Algorithm

Initial transmission attempts failed because the virtual router dropped packets with an invalid IPv4 Checksum. We implemented the standard checksum algorithm required by RFC 791:

1. Treat the header as a sequence of 16-bit integers.
2. Sum all integers using One's Complement arithmetic.
3. Fold 32-bit carries back into the lower 16 bits.
4. Bitwise invert the result.

```rust
fn calc_ip_checksum(&self, header: &[u8]) -> u16 {
    let mut sum: u32 = 0;
    // 1. Sum words
    for i in (0..header.len()).step_by(2) {
        let word = ((header[i] as u32) << 8) | (header[i+1] as u32);
        sum = sum.wrapping_add(word);
    }
    // 2. Handle Carry
    while (sum >> 16) != 0 {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    // 3. Invert
    !sum as u16
}
```

Listing 1: IPv4 Checksum Logic

3.3 DHCP Options and Magic Cookie

To prove the packet is a valid DHCP request, we included the "Magic Cookie" (`0x63825363`) followed by **Option 53** (DHCP Message Type). We set the value to `1` (DISCOVER).

4 Verification and Results

The complete stack was tested in the QEMU environment. The execution flow was captured as follows:

- **Step 1 (TX):** The OS constructed a valid UDP/IP packet and instructed the RTL8139 to broadcast it.
- **Step 2 (Routing):** The virtual router verified the checksum and accepted the DHCP Discover.
- **Step 3 (RX):** The router broadcasted a DHCP Offer.
- **Step 4 (Parsing):** The OS caught the packet, identified UDP Port 68, parsed the DHCP struct, and extracted the offered IP.

```
[NET] Sending DHCP DISCOVER (With Checksum)...
[NET] DHCP OFFER RECEIVED!
   >>> ASSIGNED IP: 10.0.2.15 <<<
```

5 Conclusion

Chronos OS has successfully implemented the fundamental layers of the Internet Protocol Suite (Link, Internet, and Transport). The kernel now possesses a valid identity on the network (`10.0.2.15`). This infrastructure enables the future development of ICMP (Ping), TCP sockets, and file transfer capabilities.