Chronos OS: Phase 3 Report

Visual Output: Framebuffers, Text Rendering, and Safety

Development Log

January 19, 2026

Abstract

Phase 3 of the Chronos OS development focused on transforming the kernel's output capabilities. Moving beyond simple color bars, we implemented a full bitmap text rendering engine to display alphanumeric data. This required integrating a raster font library, developing a custom software blitter, and solving critical Rust memory safety challenges regarding raw pointers and global concurrency.

Contents

1 Objective

While Phase 1 established a graphical framebuffer, the OS lacked the ability to communicate textual information (logs, status updates, prompts). Phase 3 aimed to implement a `print!` style capability without access to the standard library or an underlying OS.

2 The Graphics Stack

2.1 Framebuffer Architecture

We utilize a Linear Framebuffer provided by the Limine bootloader.

- **Format:** 32-bit TrueColor (0x00RRGGBB).
- **Access:** Direct memory mapping via raw pointers (`*mut u32`).
- **Resolution:** Dynamic (typically 1280x800 or 1920x1080).

2.2 Font Rasterization

We incorporated the `noto-sans-mono-bitmap` crate. This provides pre-rasterized bitmap data for characters. Our engine takes this binary data (brightness values) and "blits" (copies) it pixel-by-pixel into the framebuffer, respecting the text color and cursor position.

3 The Writer Implementation

3.1 Cursor Tracking

To emulate a terminal, the `Writer` struct maintains specific state:

- `cursor_x` and `cursor_y`: Current draw position.
- `width` and `height`: Screen boundaries.

Logic was implemented to handle automatic line wrapping and newline (`\n`) characters.

3.2 Rust Safety Challenges

A major technical hurdle was making the Framebuffer accessible globally (so both the kernel loop and interrupt handlers can print).

The Problem: The Framebuffer is accessed via a raw pointer (`*mut u32`). Rust's safety model treats raw pointers as not thread-safe (`!Send` and `!Sync`). Attempting to put a struct containing a raw pointer into a static Mutex results in a compiler error.

The Solution: We manually implemented the safety traits, essentially signing a contract with the compiler that we guarantee safety via the Mutex lock.

```rust
pub struct Writer {
    pub video_ptr: *mut u32,
    // ...
}

// SAFETY: We guarantee that Writer is only accessed
// through a Mutex, preventing data races.
unsafe impl Send for Writer {}
unsafe impl Sync for Writer {}
```

Listing 1: Implementing Send and Sync for Raw Pointers

4 System Integration

4.1 The Visual Layout

The screen space was partitioned to support the new features:

1. **Top Zone (0-150px):** Dedicated Console Output. Protected from background animations.
2. **Middle Zone (150-200px):** The ”Fuel Gauge” (Time Budget visualizer).
3. **Bottom Zone (200px+):** The ”Void” (Background heartbeat animation).

4.2 Results

The system successfully prints:

- Boot initialization logs.
- Hardware status (”[OK] CPU Online”).
- Dynamic feedback (printing dots on keypress).

5 Conclusion

Phase 3 has successfully given Chronos a voice. The OS can now report its internal state to the user in human-readable text. This infrastructure is the foundation for the future implementation of a Shell, a filesystem inspector, and panic reporting.