OhneBS Operating System

Documentation Version 0.2.0

Marwin Zöpfel

July 2, 2025

Contents

1 Introduction and Concept

This document describes the architecture and functionality of the bare-metal operating system "OhneBS," version 0.2.0.

The term "bare-metal" signifies that this operating system runs directly on the Raspberry Pi hardware, without the assistance of an underlying OS like Linux. Every aspect, from controlling hardware pins to processing user input, must be implemented from scratch.

Project Objective

The primary goal of OhneBS is to learn the fundamental concepts of low-level hardware programming and operating system development, and to create a simple yet functional interactive system.

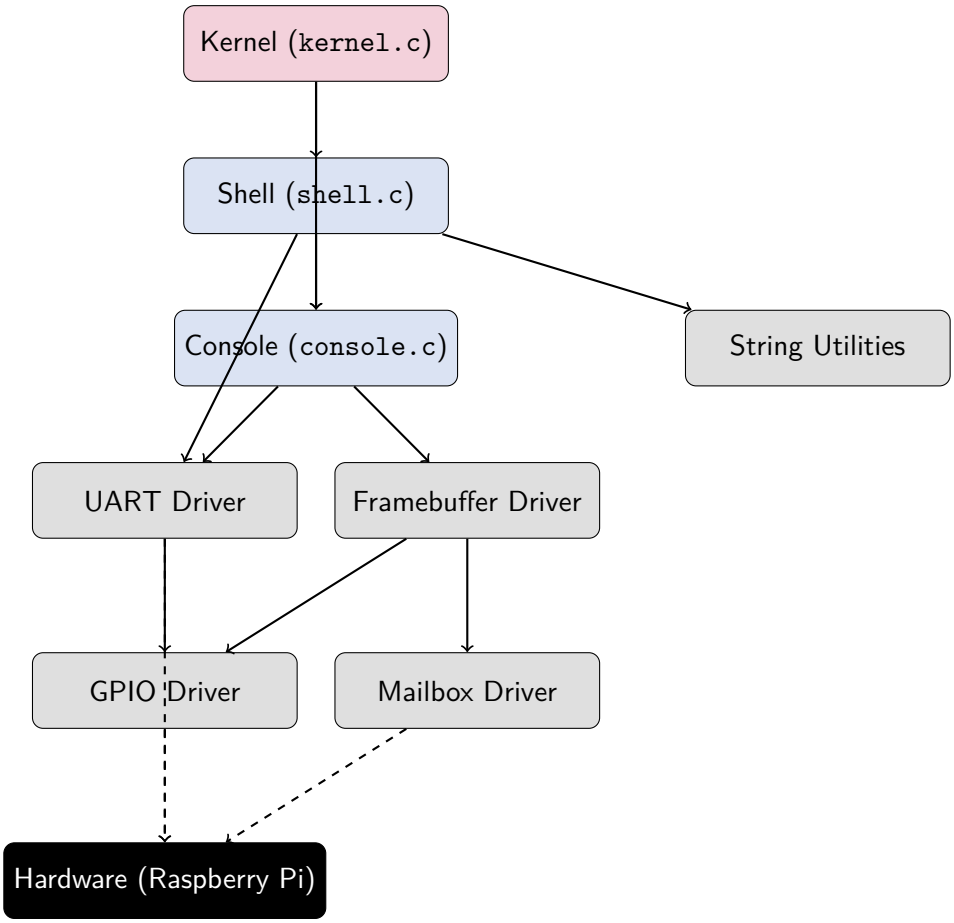
Version 0.2.0 implements a serial command-line interface (shell) that enables variable management and basic arithmetic. Additionally, it introduces basic graphics capabilities on the Raspberry Pi's framebuffer.

2 Architecture and Project Structure

OhneBS adheres to the principle of modularization. The architecture is divided into distinct layers to ensure a clean separation of responsibilities.

2.1 Architecture Diagram

The following diagram illustrates the dependencies between the software layers:



2.2 Directory Tree

The physical structure on the disk reflects the logical separation.

```
OhneBS/
├── build/
├── include/
│   ├── console.h
│   ├── fb.h
│   ├── gpio.h
│   ├── mb.h
│   ├── shell.h
│   ├── string_utils.h
│   └── uart.h
├── src/
│   ├── boot.S
│   ├── console.c
│   ├── fb.c
│   ├── gpio.c
│   ├── kernel.c
│   ├── mb.c
│   ├── shell.c
│   ├── string_utils.c
│   └── uart.c
├── LICENSE
├── link.ld
├── Makefile.gcc
└── Readme.md
```

3 Boot Process and Kernel Initialization

This section provides a detailed, step-by-step walkthrough of the entire process that brings OhneBS to life, from the moment the Raspberry Pi is powered on to the execution of the main kernel loop.

3.1 The Black Box: The Firmware Stage

The very first moments of the boot process are controlled by proprietary firmware stored in the Raspberry Pi's on-chip ROM. This stage is a "black box" for our operating system, but it's crucial to understand what it does for us.

1. **Power On:** The ARM cores are initially off. The VideoCore IV GPU is the first component to start.
2. **First Stage Bootloader:** The GPU executes code from its internal ROM, which mounts the SD card and looks for the file `bootcode.bin`.
3. **Second Stage Bootloader:** `bootcode.bin` is loaded into the GPU's L2 cache and executed. Its primary job is to initialize the SDRAM, allowing the larger third-stage bootloader to be loaded.
4. **Third Stage Bootloader (`start.elf`):** This is the main GPU firmware. It reads the `config.txt` file to configure various hardware parameters (like clock speeds, HDMI settings, etc.). Its final and most important task for us is to look for our kernel image file, `kernel8.img`, load it into SDRAM at the physical address `0x80000`, and finally, "wake up" the primary ARM CPU core (Core 0) and instruct it to start executing code at that address.

The Handoff

The moment `start.elf` finishes and points the ARM core's Program Counter to `0x80000` is the magical handoff. From this point on, the proprietary firmware is out of the picture, and **the OhneBS code has full control** of the machine.

3.2 The Kernel Stage: From Assembly to C

Execution now begins at the very first instruction in our own code, defined by the `_start` label in the `boot.S` assembly file.

3.2.1 Step 1: Core Verification

The Raspberry Pi 400 has a quad-core CPU. The firmware only starts Core 0. It's our responsibility to handle the other cores. The first thing we do is check which core is executing our code.

```
1    // Read the Multiprocessor Affinity Register (MPIDR_EL1)
2    mrs     x1, mpidr_el1
3    // Isolate the last two bits, which contain the core ID (0-3)
4    and     x1, x1, #3
5    // If the result is zero (it's Core 0), branch to the main setup
6    cbz     x1, 2f
7
8    // We're not on the main core, so hang in an infinite wait loop
9    1:  wfe
10   b       1b
11   2:  // We're on the main core! Continue...
```

```
12
```

Listing 1: Checking the core ID in `boot.S`

This ensures that only the primary core proceeds to initialize the kernel. The other cores are put into a low-power sleep state using the ‘wfe‘ (Wait For Event) instruction.

3.2.2 Step 2: Stack Initialization

Before we can call any functions (especially C functions), we need a stack. The stack is a region of memory used for local variables, function parameters, and return addresses. We initialize the Stack Pointer register (‘sp‘) to point to the beginning of our kernel code. Since the stack on ARM grows downwards, this gives us the memory space *below* our code to use as a temporary stack.

```
1    // Set stack to start at the beginning of our code
2    ldr     x1, =_start
3    mov     sp, x1
4
```

Listing 2: Setting up the initial stack pointer.

3.2.3 Step 3: Clearing the BSS Section

In C, global or static variables that are not explicitly initialized are guaranteed to be zero. The compiler doesn't store these zeros in the image file. Instead, it groups all such variables into a memory region called the "BSS section" and provides two symbols: `__bss_start` and `__bss_size`. It is the operating system's job to manually write zeros to this entire memory region before any C code runs.

```
1    // Clean the BSS section
2    ldr     x1, =__bss_start     // Start address
3    ldr     w2, =__bss_size      // Size of the section
4    3:  cbz     w2, 4f               // Quit loop if size is zero
5    str     xzr, [x1], #8        // Store a 64-bit zero and increment address
6    sub     w2, w2, #1           // Decrement size counter (in 8-byte chunks)
7    cbnz    w2, 3b               // Loop if non-zero
8
```

Listing 3: Zeroing out the BSS section.

3.2.4 Step 4: The Jump to C (`kernel_main`)

With the low-level setup complete, the assembly code performs its final task: it calls our main C function, `kernel_main`. The ‘bl‘ (Branch with Link) instruction jumps to the C code and also stores the return address in a register, although our `kernel_main` function is designed to never return.

```
1    // Jump to our main() routine in C
2    4:  bl      kernel_main
3
```

Listing 4: Calling the C kernel.

3.2.5 Step 5: High-Level Initialization (`kernel.c`)

Control is now within the familiar world of C. The `kernel_main` function acts as the main orchestrator, initializing all the high-level drivers and services in a specific order.

```c
void kernel_main() {
  // 1. Initialize serial communication for debugging
  uart_init();

  // 2. Initialize the shell's internal state
  shell_init();

  // 3. Initialize the framebuffer via mailbox calls to the GPU
  fb_init();

  // 4. Print a welcome message to the console (now both screen and UART)
  console_puts("Welcome to OhneBS! v0.2.0\n");

  // ... (Drawing demo graphics) ...

  // 5. Enter the main kernel loop
  while (1) {
    // Continuously poll the shell for updates (e.g., new user input)
    shell_update();
  }
}

```

Listing 5: The entry point of the C kernel.

From this point on, the system is fully initialized and has entered its main operational loop, waiting for user input and processing commands. The boot process is complete.

4 The Interactive Shell and GPIO Driver

This section details the highest and lowest levels of the current software stack: the user-facing interactive shell that defines the system's functionality, and the fundamental GPIO driver that communicates with the physical hardware pins.

4.1 The Interactive Shell (`shell.c`)

The shell is the primary user interface of OhneBS. It provides a command-line interface (CLI) that allows a user to interact with and control the kernel.

4.1.1 Core Logic: The Polling Loop

The entire shell is driven by a single function, `shell_update()`, which is called continuously from the main kernel loop. This represents a **polling-based architecture**, where the system actively checks for input instead of waiting for an interrupt.

```c
// in kernel.c
void kernel_main() {
  // ... initializations ...

  console_puts("Welcome to OhneBS!\n> ");

  while (1) {
    // The CPU spends all its time repeatedly calling this function.
    shell_update();
  }
}

```

Listing 6: The main kernel loop, driving the shell.

Inside `shell_update()`, the `uart_read_byte()` function is called to check for new characters. If a character is available, it is processed; otherwise, the function returns immediately.

4.1.2 Input Processing and Command Execution

1. **Buffering:** Received characters are stored in a 256-byte static buffer, `input_buffer`. The current position is tracked by `input_buffer_pos`.
2. **Echoing:** Every printable character is immediately sent back to the 'console' module for display, providing instant user feedback.
3. **Editing:** The shell supports backspace ('0x08' or '0x7F') to delete the last character from the buffer and the screen.
4. **Execution:** When a carriage return ('r') is detected, a null terminator ('0') is appended to the buffer to create a valid C-string. This string is then passed to the `process_command()` function. After execution, the input buffer is cleared for the next command.

4.1.3 Command Parser (`process_command`)

The parser is a simple implementation that separates the first word of the input string to identify the command. The rest of the string is then passed as arguments.

Implemented Commands

- **`help`**: Displays a list of available commands.
- **`version`**: Prints the current version string of the OS.
- **`set <name> <value>`**: Creates or updates a named variable with a given integer value. The shell can store up to 50 variables.
- **`print <expression>`**: Evaluates a simple expression and prints the result. The current evaluator can handle two terms (numbers or variables) joined by a single '+' operator.

4.2 The GPIO Driver (`gpio.c`)

The GPIO (General Purpose Input/Output) driver is the most fundamental layer for interacting with the physical world outside the SoC. It provides direct control over the Raspberry Pi's header pins.

4.2.1 Design Philosophy: Generic Register Access

Instead of writing separate logic for each type of GPIO operation (set function, set pin, clear pin, set pull-up/down), the driver uses a single, powerful helper function: `gpio_call()`.

Understanding GPIO Registers

The BCM2711's GPIO functionality is controlled by writing to specific 32-bit registers. For example, the `GPFSELn` registers control the function of a pin (Input, Output, Alt0-5), with 3 bits dedicated to each pin. The `GPSETn` and `GPCLRn` registers set or clear pins by writing a '1' to the corresponding bit. The 'gpio_call' function abstracts the complex math needed to find the correct register and bit position for any given pin.

```
1    unsigned int gpio_call(unsigned int pin_number, unsigned int value,
2    unsigned int base, unsigned int field_size,
3    unsigned int field_max)
4    {
```

```
    unsigned int field_mask = (1 << field_size) - 1;

    // Calculate which register and which bits within that register to modify
    unsigned int num_fields = 32 / field_size;
    unsigned int reg = base + ((pin_number / num_fields) * 4);
    unsigned int shift = (pin_number % num_fields) * field_size;

    // Read-modify-write operation
    unsigned int curval = mmio_read(reg);
    curval &= ~(field_mask << shift); // Clear the relevant bits
    curval |= value << shift;         // Set the new value
    mmio_write(reg, curval);

    return 1;
}

```

Listing 7: The generic gpio_call function.

4.2.2 Public API Functions

The user-facing functions in `gpio.h` are simple wrappers around `gpio_call()`, providing a clean and readable API.

`gpio_function(pin, func)` Calls `gpio_call` with the base address of the `GPFSEL0` registers and a field size of 3 bits.

`gpio_set(pin, val)` Calls `gpio_call` with the base address of the `GPSET0` registers and a field size of 1 bit.

`gpio_pull(pin, val)` Calls `gpio_call` with the base address of the `GPPUPPDN0` registers and a field size of 2 bits to set pull-up/down states.

5 System Execution Flow and I/O Architecture

5.1 Core Philosophy: From Serial to Screen

The initial version of OhneBS relied exclusively on a serial UART connection for all input and output. While robust for debugging, this "headless" approach lacks the immediacy of a real computer system. The primary goal for version 0.2.0 was to enable graphical output on a standard HDMI monitor, providing a persistent, self-contained user interface.

The main challenge is that modern graphics hardware is incredibly complex. Writing a full driver for the HDMI interface from scratch is a monumental task. The Raspberry Pi, however, offers a clever shortcut: a powerful VideoCore IV GPU that handles the low-level hardware communication. The CPU can communicate with the GPU to request services, such as setting up a simple, linear framebuffer.

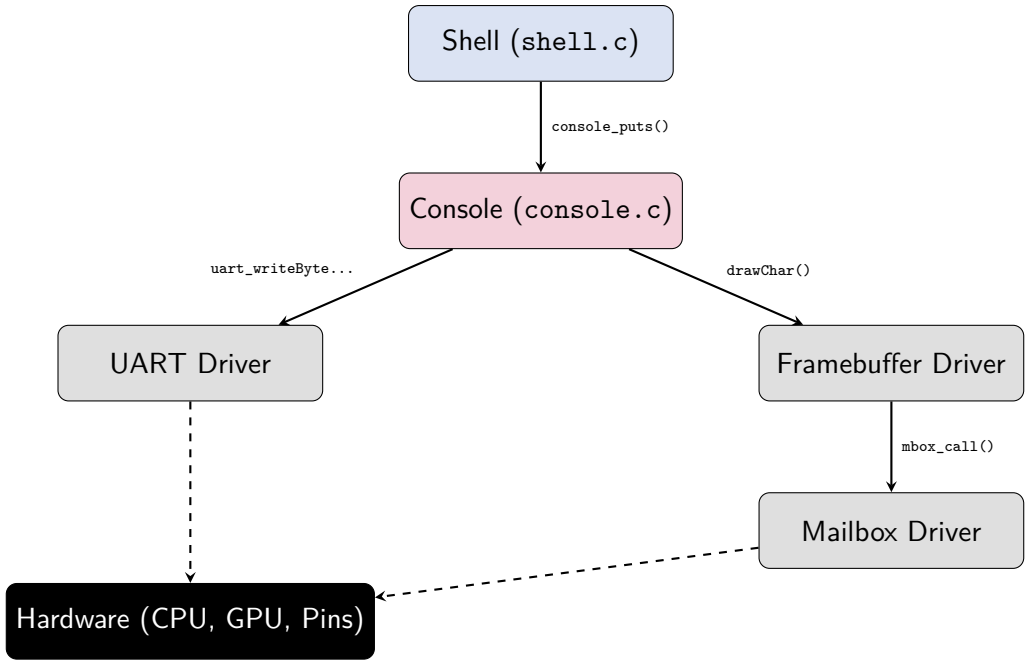
The Mailbox Interface

A **framebuffer** is a region of RAM that directly corresponds to the pixels on the screen. Writing a color value to a specific memory address in the framebuffer instantly changes the color of the corresponding pixel. To get the address and configuration of this memory region, our ARM CPU must communicate with the GPU. This is achieved via a **Mailbox**, a simple message-passing system.

5.2 The New I/O Data Flow

To support both the new framebuffer and the existing UART console, a new abstraction layer, the **Console Module**, was introduced. This module acts as a central hub for all text output. Higher-level components like the Shell no longer need to know *where* their output is going; they simply send it to the console.

The diagram below illustrates this improved data flow:



Execution Path:

1. The `shell` requests to print a string.
2. The `console` receives the request.
3. It forwards each character to **both** the `UART` driver and the `Framebuffer` driver.
4. The `Framebuffer` driver itself uses the `Mailbox` driver during its initialization.

6 Module Reference: Implementation Details

This chapter provides a deeper insight into the implementation of the new and updated modules.

6.1 Mailbox Module (`mb.c`)

This module implements the low-level protocol for communicating with the VideoCore GPU.

6.1.1 Mechanism

The communication relies on a shared memory buffer and a set of hardware registers.

1. A message buffer is prepared in RAM. This buffer must be 16-byte aligned because the GPU only receives the upper 28 bits of the address.
2. The memory address of this buffer, combined with a channel number, is written to the `MBOX_WRITE` register.
3. The CPU then polls the `MBOX_STATUS` register, waiting for the GPU to signal that it has placed a response in the mailbox.
4. The CPU reads the `MBOX_READ` register to confirm the response is for its original message and then processes the data in the shared memory buffer.

```
unsigned int mbox_call(unsigned char ch)
{
  // Combine the 28-bit address of our buffer with the 4-bit channel
  unsigned int r = ((unsigned int)((long) &mbox) &~ 0xF) | (ch & 0xF);

  // Wait until the mailbox is not full
  while (mmio_read(MBOX_STATUS) & MBOX_FULL);

  // Write our message address to the mailbox
  mmio_write(MBOX_WRITE, r);

  while (1) {
    // Wait until the mailbox is not empty (has a reply)
    while (mmio_read(MBOX_STATUS) & MBOX_EMPTY);

    // Check if the reply is for our message
    if (r == mmio_read(MBOX_READ)) {
      // Return success if the GPU response code is positive
      return mbox[1] == MBOX_RESPONSE;
    }
  }
  return 0; // Should not be reached
}

```

Listing 8: The core of the synchronous mailbox call.

6.2 Framebuffer Module (`fb.c`)

This driver uses the Mailbox interface to initialize a linear framebuffer and provides primitive functions for drawing.

6.2.1 Initialization (`fb_init`)

The `fb_init` function is a prime example of the mailbox interface in action. It constructs a long message in the global ‘mbox‘ array, filling it with various "tags" to request services from the GPU.

Key Mailbox Tags Used

- `MBOX_TAG_SETPHYWH`: Requests a specific physical screen resolution (e.g., 1920x1080).
- `MBOX_TAG_SETDEPTH`: Requests a specific color depth (32 bits per pixel).
- `MBOX_TAG_GETFB`: The most important tag. It asks the GPU to allocate the framebuffer memory and return its address and size.
- `MBOX_TAG_GETPITCH`: Asks for the "pitch," which is the number of bytes per row on the screen. This is crucial for calculating pixel offsets correctly.

After a successful `mbox_call`, the GPU-provided physical address is converted to an ARM-accessible address, and the screen parameters (width, height, pitch) are stored for later use.

6.2.2 Drawing Pixels (`drawPixel`)

This is the most fundamental drawing function. All other graphics primitives (‘drawLine‘, ‘drawRect‘, etc.) are built upon it. It calculates the memory offset for a given (x, y) coordinate and writes a 32-bit color value to that location.

```
1    void drawPixel(int x, int y, unsigned char attr)
2    {
3      // The pitch is the number of bytes to get to the next row.
4      // Each pixel is 4 bytes wide (32-bit color).
5      int offs = (y * pitch) + (x * 4);
6
7      // Write the 32-bit color value directly to the framebuffer memory.
8      // 'vgapal' is an array that maps the 4-bit color attribute to a 32-bit
   color.
9      *((unsigned int*)(fb + offs)) = vgapal[attr & 0x0f];
10   }
11
```

Listing 9: Calculating the memory offset for a pixel.

6.3 Console Abstraction Module (`console.c`)

This new module provides a unified text output interface, decoupling the shell from the physical display hardware.

6.3.1 Dual Output

The core of this module is the `console_putc` function. It acts as a multiplexer, sending every single character to two destinations:

1. **To the UART Driver:** via `uart_writeByteBlocking(c)`. This ensures that all output is still visible on the serial debug console.
2. **To the Framebuffer Driver:** via `drawChar(...)`. This renders the character graphically on the screen.

This dual-output approach is invaluable for debugging; even if the graphical output fails, the serial log remains functional.

6.3.2 Cursor Management and Scrolling

The console maintains static variables `current_x` and `current_y` to track the cursor position on the screen. It handles special characters:

- `\n` (Newline): Triggers the `console_newline()` helper function.
- `\r` (Carriage Return): Resets `current_x` to 0.
- `\b` (Backspace): Moves the cursor back and overwrites the character with a background-colored rectangle.

⚠ Current Scrolling Implementation

The scrolling mechanism in version 0.2.0 is very basic. When the cursor reaches the bottom of the screen, the entire framebuffer is cleared, and the cursor is reset to the top-left corner (0,0). A future version should implement a more efficient ‘memcpy‘-style operation to scroll the existing content up, providing a smoother user experience.