AstraKernel Documentation

A Minimal Kernel for QEMU's VersatilePB (ARM926EJ-S) Platform

Written in Modern C and ARM Assembly

Version 0.1

By Chris Dedman

2025-05-19 ©SandboxScience

This page is intentionally left blank.

Contents

Preface

This documentation serves as a comprehensive guide to the AstraKernel project, a minimal operating system kernel written in modern C and ARM assembly. Designed to run on QEMU's VersatilePB (ARM926EJ-S) emulated platform, AstraKernel is intended to provide a clear and approachable introduction to the fundamental concepts of operating system design and development. This project also reflects my personal journey in learning about kernel development and systems programming.

This project was developed with a focus on clarity, simplicity, and educational value. Rather than attempting to recreate the complexity of established operating systems, AstraKernel's goal is to strip away unnecessary abstractions and present a clean, understandable codebase for anyone interested in the "bare metal" foundations of computing.

Through hands-on implementation of kernel bootstrapping, direct hardware communication, and basic user interaction, AstraKernel demonstrates how fundamental OS components come together. The project showcases how modern C best practices can be utilized in a systems programming context to create code that is maintainable, portable, and robust, while still being accessible to those new to kernel development. The design of this kernel emphasizes modularity and extensibility, allowing developers to easily add new features or modify existing ones. This makes it ideal for educational purposes, as it provides a clear structure that can be followed and built upon.

It is my hope that AstraKernel will not only serve as a foundation for those wishing to understand kernel development, but also inspire curiosity and confidence in exploring lower-level aspects of computer systems.

INFO:

> This documentation is a work in progress and may be updated as the project evolves. I welcome contributions, feedback, and suggestions for improvement. You can find the source code on GitHub: `https://github.com/sandbox-science/AstraKernel`

About This Project

Resources To guide my learning and support development of AstraKernel, I am using the following resources, which are particularly valuable for foundational and practical understanding of OS design:

- **Operating Systems: Three Easy Pieces** by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
- **The Little Book About OS Development** by Erik Helin and Adam Renberg

Contributions AstraKernel is an open source project, and I encourage contributions from anyone interested in improving, extending the kernel or simply experiment with it. I also encourage anyone to improve this documentation, as it is a work in progress. If you would like to contribute, please feel free to open an issue or pull request on the GitHub repository.

INFO:

> You can join the Matrix Server `https://matrix.to/#/#sandboxscience:matrix.org` for general SandBox Science discussion as well as project-specific discussion such as AstraKernel development.

Acknowledgments Specially thanks to the following individuals for their contributions:

- **shade5144** (`https://github.com/shade5144`) for contributing the initial version of the `printf` and `datetime` function as well as the documentation for them.

Disclaimer AstraKernel is currently in its early stages of development and is not intended for production use. It is primarily an educational and experimental project. The code is provided as-is, without any warranties or guarantees of any kind. Use it at your own risk.

Chapter 1

Introduction

I. Getting Started

AstraKernel begins its life in a small bootstrap routine, written in ARM assembly, that prepares the processor's state before passing control to the main C kernel. This bootstrap code is responsible for setting up the stack pointer, clearing the uninitialized data section (`.bss`), and ensuring a clean environment for the kernel's entry point. Below is the initial assembly code that executes at startup:

```
    .section .text
    .global _start

_start:
    // Set up the stack pointer
    LDR sp, =_estack
    BIC sp, sp, #7

    // Zero the .bss section
    LDR R0, =__bss_start   // Start address (symbol from linker script)
    LDR R1, =__bss_end     // End address (symbol from linker script)
    MOV R2, #0             // init zero-value for BSS clearing

zero_bss:
    // Check if we are done zeroing the BSS
    CMP R0, R1             // Compare current address to end
    BGE bss_done           // If done, skip zeroing
    STR R2, [R0], #4       // Store zero at [r0], increment r0 by 4
    B zero_bss

bss_done:
    // Call kernel_main function
    BL kernel_main

hang:
    // Halt if kernel_main returns (should not happen)
    B hang    // Infinite loop
```

Listing 1.1: Initial bootstrap code for AstraKernel.

This startup sequence is the essential first step for any kernel, ensuring the CPU is properly initialized and memory is in a known state before higher-level code takes over. Once these preparations are complete, the `kernel_main` function from `kernel/kernel.c` is called, marking the transition from low-level assembly to the C code that forms the core of AstraKernel.

A. Prerequisites

Before you can build and run AstraKernel, please ensure you have the following tools installed on your system:

- **ARM Cross-Compiler:** A cross-compiler targeting ARM is required to build the kernel. It is recommended to use `arm-none-eabi-gcc`, `arm-none-eabi-ld`, and `arm-none-eabi-objcopy` for ARM926EJ-S, which is the target architecture for AstraKernel.
  - Example installation: `arm-none-eabi-xxx` (available via package managers such as `brew`, `apt`, or direct download from ARM's website).
- **QEMU Emulator:** QEMU is used to emulate the ARM VersatilePB (ARM926EJ-S) platform for kernel development and testing.
  - Ensure your QEMU installation supports the `versatilepb` machine.
  - Example installation: `qemu-system-arm` via qemu https://www.qemu.org/download/.
- **Build Tools:** Standard build tools such as `make` are required to compile the kernel.
  - Example installation: `make` (available via package managers such as `brew`, `apt`, or direct download https://www.gnu.org/software/make/#download).

For best results, ensure all tools are up-to-date. Consult the official documentation of each tool for installation instructions on your operating system.

B. Building & Running AstraKernel

NOTE:

> The following instructions assume you have the necessary tools installed on your system as mentioned in the prerequisites.

This project uses `Make` to automate the build process. The configuration is located in the `Makefile` in the root directory of the kernel source code. To build and run the kernel, navigate to the root directory of the AstraKernel source code and execute the following command in your terminal:

```
make
```

Listing 1.2: Building AstraKernel.

This command invokes the Makefile, which automatically compiles the kernel source code, links the object files, and generates the final kernel binary. The output files are placed in the `build/` directory, and any previously compiled files there are removed to ensure a clean build environment. Finally, the commands also run `make qemu`, which launches the QEMU emulator with the built kernel image.

```
# Assembly start.o goes to build/
$(OUT_DIR)start.o: kernel/start.S
	@mkdir -p $(OUT_DIR)
	$(AS) -c $< -o $@

# Pattern rule for any .c -> build/*.o
$(OUT_DIR)%.o: %.c
	@mkdir -p $(OUT_DIR)
	$(CC) $(CFLAGS) -c $< -o $@

# Link everything
$(OUT_DIR)kernel.elf: $(OUT_DIR)start.o $(OBJS) kernel.ld
	$(LD) $(LDFLAGS) $(OUT_DIR)start.o $(OBJS) -o $@

# Generate the kernel binary from the ELF file
kernel.bin: $(OUT_DIR)kernel.elf
	$(OBJCOPY) -O binary $< $(OUT_DIR)$@
```

Listing 1.3: Makefile for AstraKernel.

INFO:

> You can run each `make` target on its own. Run `make kernel.bin` to compile the kernel binary, `make qemu` to launch the built kernel in QEMU, and `make clean` to remove all object files and the `kernel.bin` from the `build/` directory.

If the build is successful, you will see the output similar to the following:

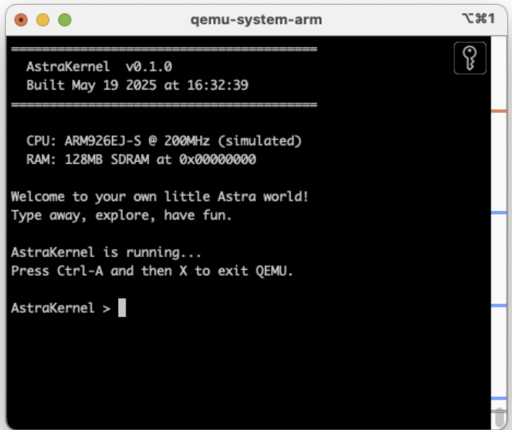

Figure 1.1: AstraKernel booted in QEMU.

C. Codebase's Overview

Coming soon...

Chapter 2

I/O & Time Services

I. UART Output API

Registers & Constants

- **UART0_DR** (Data Register), `0x101f1000`
- **UART0_FR** (Flag Register), `0x101f1018`
- **UART_FR_TXFF**, **UART_FR_RXFE**

A. `printf(char *s, ...)`

Description Sends a null-terminated format string over UART. If an incorrect datatype is given for a format specifier, the behavior is undefined. If a format specifier is given without a matching argument, it is simply skipped when the string is outputted. The following format specifiers are supported:

Supported Format Specifiers

- `%c`: Expects a single character.
- `%s`: Expects a null-terminated string.
- `%d`: For 32-bit signed integers in the range `-2147483648` to `2147483647`.
- `%ld`: For 64-bit signed integers in the range `-9223372036854775808` to `9223372036854775807`, excluding the range specified above, for **%d**.
- `%lu`: For 64-bit unsigned integers in the range `2147483648` to `18446744073709551615`.
- `%x`, `%X`: For 32-bit unsigned integers in the range `0` to `2147483647`, where the case of `x` determines the case of the hexadecimal digits (`a-f` or `A-F`).

- `%lx`, `%lX`: For 64-bit unsigned integers printed in hexadecimal format. Has the same range as **%lu**. The case of `x` determines the case of the hexadecimal digits (`a-f` or `A-F`).
- `%%`: Outputs a literal `%`.
- `%`: If the format specifier is not followed by a valid character, prints `(null)`.

All the ranges specified above are inclusive. Also, note that integers in the range `-2147483648` to `2147483647` are passed as 32 bit integers and any integers not part of this range are passed as 64 bit integers by default. If desired, integers of this range can be cast as `long long` or `unsigned long long` for use with the format specifiers prefixed by the character `l`.

Examples

```c
// Printing long long integers
printf("%lu %ld %ld\n", 18446744073709551615, -9223372036854775808,
9223372036854775807);

// Printing 32-bit signed integers
printf("%d %d\n", 2147483647, -2147483648);

// Printing 32-bit unsigned integers
printf("%x %x %X %X\n", 2147483647, 1234, 2147483647, 1234);
// Output: 7fffffff 4d2 7FFFFFFF 4D2

// Printing unsigned long long integers in hex
printf("%lX %lx\n", 0x123456789ABCDEF0, 9223372036854775809);
// Output: 123456789ABCDEF0 8000000000000001

// Printing a character
printf("Name: %c\n", 'b');

// Printing a string
printf("Hello %s\n", "World");

// Printing a '%'
printf("100%%\n");

// Printing a null character
printf("%\n");
// Output: (null)
```

Listing 2.1: Examples of printf usage in AstraKernel.

II. String Manipulation API

A. `strcmp(const char *str_1, const char *str_2)`

Purpose Compares two null-terminated strings, character by character.

Overview This function compares the characters of two strings (`ASCII values`), `str_1` and `str_2`, one by one. It returns:

Return Values

- 0 if both strings are equal,
- -1 if the first differing character in `str_1` is less than that in `str_2`,
- 1 if the first differing character in `str_1` is greater than that in `str_2`.

Behavior The comparison stops at the first difference or the null terminator. Case sensitivity is observed. Behavior is undefined if either pointer is not a valid null-terminated string.

Examples

```c
int result = strcmp("abc", "abc"); // Expect 0
printf("Expect 0 -> %d\n", result);

result = strcmp("abc", "abd"); // Expect -1
printf("Expect -1 -> %d\n", result);

result = strcmp("abc", "ABC"); // Expect 1
printf("Expect 1 -> %d\n", result);
```

Listing 2.2: String Comparison Example

NOTE:

> In the future, we could returning the difference between the first differing characters, which would allow for more detailed comparisons. This would enable users to understand how far apart the strings are in terms of character values.

III. Timekeeping API

Provide calendar dates and clock time based on the on-chip RTC.

Overview

- Reads a 32-bit seconds-since-1970 counter (RTC register at ‘0x101e8000‘).
- Exposes two APIs:
  - `getdate(dateval *date)`: Fetches current date in days, months, and years.
  - `gettime(timeval *time)`: Fetches current time in hours, minutes, and seconds.

Data Structure

```c
typedef struct
{
    uint32_t hrs;
    uint32_t mins;
    uint32_t secs;
} timeval;

typedef struct
{
    uint32_t day;
    uint32_t month;
    uint32_t year;
} dateval;
```

Listing 2.3: Structure definitions for time and date values.

The number of seconds since epoch is retrieved from 926EJ-S's Real Time Clock Data Register(**RTCDR**). This is a 32 bit register, and therefore subject to the Year 2038 Problem.

A. `uint32_t getdate(dateval *date_struct)`

Populate the provided `*date_struct` with the current date in days, months, and years.

Parameters

- `date_struct` pointer to a `dateval` structure; if `NULL`, only the raw **RTC** counter is returned.

Return Value

- Returns number of seconds since epoch (`RTCDR`) as read from the hardware register.

B. `uint32_t gettime(timeval *time_struct)`

Populate `*time_struct` with the current click time in hours, minutes, and seconds.

Parameters

- `time_struct` pointer to a `timeval` structure; if `NULL`, only the raw **RTC** counter is returned.

Return Value

- Returns number of seconds since epoch (`RTCDR`) as read from the hardware register.

Examples

```
gettime(&time_struct);
printf("Current time(GMT): %d:%d:%d\n", time_struct.hrs, time_struct.mins, time_struct.secs);

getdate(&date_struct);
printf("Current date(MM-DD-YYYY): %d-%d-%d\n", date_struct.month, date_struct.day, date_struct.year);
```

Listing 2.4: Examples of getdate and gettime usage in AstraKernel.

Notes

- Assumes `uint32_t` is 4-bytes (compile-time check via `_Static_assert`).
- The time and date values are based on the system's RTC, which should be set correctly at boot time.
- The API does not handle time zones or daylight saving time adjustments; it provides GMT time only.