AstraKernel Documentation

A Minimal Kernel for QEMU's VersatileAB/PB (Cortex-A8) Platform

Written in Modern C and ARM Assembly

Version 0.1

By Chris Dedman

Last edited: January 8, 2026

This page is intentionally left blank.

Contents

Preface

This documentation serves as a practical guide to the AstraKernel project, a minimal operating system kernel written in modern C and ARM assembly. Designed to run on QEMU's VersatileAB (Cortex-A8) emulated platform, AstraKernel is intended to provide a clear and approachable introduction to the fundamentals of kernel development. This project also reflects my personal journey in learning about 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 used 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 emphasizes modularity and extensibility, allowing developers to add new features or modify existing ones as they learn. This makes it ideal for educational purposes and experimentation.

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 or extending the kernel or simply experimenting 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

Special 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 programs the exception vector base, sets up mode-specific stacks (SVC/IRQ/FIQ/ABT/UND), initializes `.data`, clears `.bss`, and ensures a clean environment for the kernel's entry point. Below is a trimmed excerpt of the current startup sequence:

```
    .section .vectors
    .global _start
_start:
    // Switch to SVC, mask IRQ/FIQ
    MRS     R0, cpsr
    BIC     R0, R0, #0x1F
    ORR     R0, R0, #0x13
    ORR     R0, R0, #(1 << 7)
    ORR     R0, R0, #(1 << 6)
    MSR     cpsr_c, R0

    // Point VBAR at the vector table
    LDR     R0, =vectors_base
    MCR     P15, 0, R0, C12, C0, 0
    ISB

    // Set mode stacks (SVC/IRQ/FIQ/ABT/UND)
    LDR     sp, =__stack_top__
    // ... switch modes and set stack pointers ...

    // Copy .data from LMA to VMA
    LDR     R0, =__data_load
    LDR     R1, =__data_start
    LDR     R2, =__data_end
```

```
25    // ... copy loop ...
26
27    // Clear .bss
28    LDR     R0, =__bss_start
29    LDR     R1, =__bss_end
30    MOV     R2, #0
31    // ... zero loop ...
32
33    BL      kernel_main
```

Listing 1.1: Bootstrap excerpt 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 `src/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 Cortex-A8 processor with ARMv7-A, 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 VersatileAB (Cortex-A8) platform for kernel development and testing.
  - Ensure your QEMU installation supports the `versatileab` 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: src/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 $@ $(KFLAGS)

# 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.

C. Build Modes

Overview

The Makefile provides a couple of build-time options to enable tests and instrumentation. These are passed via `KFLAGS`.

- `USE_KTESTS`: enable the kernel test suite (see `src/kernel/tests/test_memory.c`).
- `KLOG_USE_TICKS`: include tick counts in `KLOG` output and start the timer/IRQs early in `kernel_main()`.

Examples

```
# Run kernel tests
make KFLAGS="-DUSE_KTESTS"

# Enable tick timestamps in logs
make KFLAGS="-DKLOG_USE_TICKS"
```

Listing 1.4: Build modes.

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

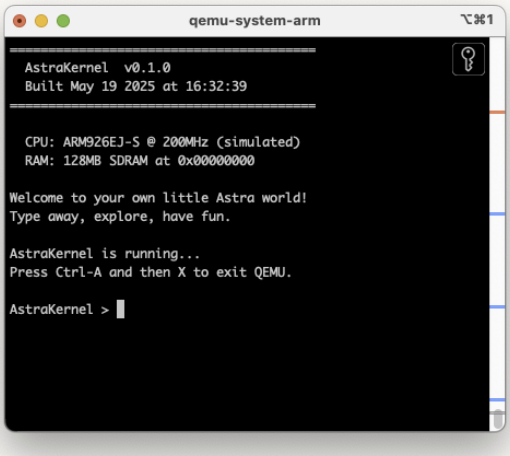

Figure 1.1: AstraKernel booted in QEMU.

D. Codebase's Overview

- `src/kernel/`: core kernel code (boot, interrupts, memory, panic, errno).
- `src/user/`: basic user-facing utilities (printf, string, datetime, clear).
- `src/kernel/tests/`: kernel tests (currently `kmalloc`).
- `include/`: public headers for kernel and user modules.
- `kernel.ld`: linker script and memory layout (vectors, text, rodata, data, bss, heap, stacks).
- `Makefile`: build rules and flags (including `KFLAGS`).
- `doc/`: LaTeX source for the kernel manual; `docs/` contains generated docs.
- `docs/pages/`: short markdown notes (panic, memory).
- `literature/`: hardware reference PDFs for VIC, timers, and ARM toolchain.
- `Dockerfile` and `Dockerfile.dev`: containerized build flows.

Chapter 2

Kernel Core

I. Exception Vector Table

Overview

AstraKernel installs an ARMv7-A exception vector table in the `.vectors` section and points `VBAR` at it during early boot. The vector table is aligned to 32 bytes and contains branch stubs for each exception type.

Vector Table Layout

```
1  /* 0x00 Reset        */   B   _start
2  /* 0x04 Undefined    */   B   undef_handler
3  /* 0x08 SWI/SVC      */   B   svc_entry
4  /* 0x0C PrefetchAbt  */   B   pabort_handler
5  /* 0x10 DataAbt      */   B   dabort_handler
6  /* 0x14 Reserved     */   B   reserved_handler
7  /* 0x18 IRQ          */   B   irq_entry
8  /* 0x1C FIQ          */   B   fiq_handler
```

Listing 2.1: Vector table layout (A32).

Vector Entries

- Reset: branches to `_start`.
- Undefined instruction, Prefetch/Data Abort, Reserved, FIQ: default handlers spin in a tight loop.
- SVC: prints a short message and returns.
- IRQ: jumps to a C handler in `src/kernel/interrupt.c`.

Implementation Notes

The boot code in `src/kernel/start.s` sets `VBAR` to the vector base, disables high vectors (SCTLR.V = 0), and installs mode-specific stacks for SVC/IRQ/FIQ/ABT/UND before entering

`kernel_main()`.

II. Boot Sequence

High-Level Flow

1. Set CPU mode to SVC and mask IRQ/FIQ.
2. Program `VBAR` to the vector base and disable high vectors.
3. Initialize stacks for SVC/IRQ/FIQ/ABT/UND.
4. Copy `.data` from load address to runtime address.
5. Zero `.bss`.
6. Jump to `kernel_main()`.

Mode Stacks

Each exception mode has its own stack to avoid clobbering the main kernel stack during fault handling. Stacks are reserved in `.bss` in `src/kernel/start.s`.

III. Interrupts and Timer

Overview

The kernel configures the SP804 timer (Timer0) and routes its interrupt through the PL190 VIC. The IRQ handler clears the timer interrupt and increments a global tick counter:

- `systicks` lives in `src/kernel/interrupt.c` and is incremented once per timer interrupt.
- `interrupts_init_timer0()` programs Timer0 in periodic mode and enables the VIC line.
- `irq_handler()` clears the timer interrupt and acknowledges the VIC by writing `VIC_VECTADDR = 0`.

Key Registers (VersatilePB)

- PL190 VIC base: `0x10140000`
- SP804 Timer0 base: `0x101E2000`
- Timer0 registers: `T0_LOAD`, `T0_CONTROL`, `T0_INTCLR`, `T0_MIS`

Initialization Sequence

1. Compute the reload value: `load = timer_clk_hz / tick_hz`.
2. Program Timer0 in periodic mode with interrupts enabled.
3. Enable Timer0/1 IRQ line in the VIC.
4. Unmask CPU IRQs via `irq_enable()`.

IV. Linker Layout and Memory Map

Overview

The linker script `kernel.ld` defines the full memory layout:

- `.vectors` at `0x00000000` (32-byte aligned).
- `.text` at `0x00010000` (64 KiB offset).
- `.rodata`, `.data`, `.bss` in ascending order.
- `.ptables` reserves 16 KiB for MMU page tables.
- Stack region at the top of RAM; heap below stacks.

Key Symbols

- `__text_start`, `__text_end`
- `__data_start`, `__data_end`, `__data_load`
- `__bss_start`, `__bss_end`
- `__stack_top__`, `__stack_bottom__`
- `__heap_start__`, `__heap_end__`, `__heap_size__`

V. Logging Macro

Overview

AstraKernel provides a minimal logging macro in `include/log.h`. It prefixes messages with a level tag and optionally includes the tick counter.

Usage

```
#include "log.h"

KLOG(KLOG_INFO, "kernel_main start");
KLOG(KLOG_WARN, "timer fallback");
KLOG(KLOG_ERROR, "bad ptr=%p", ptr);
```

Listing 2.2: Logging examples.

Tick Prefix

Define `KLOG_USE_TICKS` at compile time to include `systicks` in the log prefix. The kernel starts the timer and enables IRQs early in `kernel_main()` when this flag is set.

VI. Kernel Heap (kmalloc/kfree)

Overview

The kernel includes a simple first-fit allocator in `src/kernel/memory.c`. It manages a linked list of block headers stored inside the heap region.

Behavior

- `kmalloc_init(start, end)` sets up a single free block over the heap range.
- `kmalloc(size)` returns an aligned block; it splits larger free blocks when possible.
- `kfree(ptr)` marks a block free and merges adjacent free blocks.

Heap Layout

The heap begins at `__heap_start__` (end of `.bss`) and ends at `__heap_end__` (below the reserved kernel stacks), as defined in `kernel.ld`.

VII. Panic and Error Codes

Overview

Critical failures call `kernel_panic()` in `src/kernel/panic.c`. This prints a panic message (with an error code string) and halts the CPU using an infinite `wfi` loop with IRQ and FIQ masked.

Error Codes

Kernel error codes are defined in `include/errno.h` and include `KERR_NOMEM`, `KERR_NO_SPACE`, and `KERR_INVAL`.

Errno Helpers

The error module provides a tiny helper API:

- `kerr_is_ok(e)` returns true when the error code is `KERR_OK`.
- `kerr_is_err(e)` returns true when the error code is negative.
- `error_str(e)` returns a static string describing the code.

VIII. Tests

Overview

The kernel includes simple allocator tests in `src/kernel/tests/test_memory.c`. To run them, build with:

```
1 make KFLAGS="-DUSE_KTESTS"
```

Listing 2.3: Running kernel tests.

When enabled, the test suite runs in `kernel_main()` before returning to the interactive prompt.

Chapter 3

Error Handling

I. Error Codes

Overview

AstraKernel uses a small set of negative error codes defined in `include/errno.h`. These values are intended for internal kernel APIs that may fail during early boot or runtime.

Defined Codes

- `KERR_OK` (0): success.
- `KERR_NOT_FOUND` (-1): resource not found.
- `KERR_NOMEM` (-2): out of memory.
- `KERR_NO_SPACE` (-3): no space available.
- `KERR_INVAL` (-4): invalid request or parameter.

II. Helpers

Overview

The errno module provides small helper functions for checking and displaying error codes:

- `kerr_is_ok(e)` returns true if `e == KERR_OK`.
- `kerr_is_err(e)` returns true if `e < 0`.
- `error_str(e)` returns a static string for the code.

Example

```
kerror_t err = KERR_NOMEM;
if (kerr_is_err(err))
{
    printf("error: %s\n", error_str(err));
}
```

Listing 3.1: Error code formatting example.

Chapter 4

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, the behavior is undefined. The following format specifiers are supported:

Supported Format Specifiers

- `%c`: Expects a single character.
- `%s`: Expects a null-terminated string.
- `%d`: Signed integers (`int`).
- `%u`: Unsigned integers (`unsigned int`).
- `%x`, `%X`: Unsigned integers printed in hexadecimal; case depends on `x`/`X`.
- `%ld`: Signed `long` integers.
- `%lu`: Unsigned `long` integers.
- `%lx`, `%lX`: Unsigned `long` printed in hexadecimal.

- `%p`: Pointer, printed as `0x` followed by hex digits.
- `%%`: Outputs a literal `%`.
- Unknown specifier: the sequence is printed verbatim as `%x` (where `x` is the unknown specifier).

On ARMv7-A, `int` and `long` are 32-bit, so `%d` and `%ld` are the same width (likewise for `%u` and `%lu`). 64-bit integer formatting (e.g., `%lld` or `%llu`) is not implemented.

Examples

```
// 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 pointers
printf("UART base: %p\n", (void *)0x101F1000);

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

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

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

// Unknown specifiers are passed through
printf("%q\n");
// Output: %q
```

Listing 4.1: 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 4.2: 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.

B. `strlen(const char *str)`

Purpose

Calculates the length of a null-terminated string.

Overview

This function counts the number of characters in a null-terminated string, excluding the null terminator itself. It returns the length of the string as an unsigned integer. This function uses pointer arithmetic to traverse the string, performing mathematical operations directly on the pointer values.

Return Values

- The number of characters in the string, not including the null terminator.
- 0 if the string is empty (i.e., the first character is the null terminator).

Behavior

The function iterates through the input string until the null terminator is found, returning the number of characters preceding it. If the input pointer is not a valid null-terminated string, the behavior is undefined.

Examples

```
size_t result = strlen("abc"); // Expect 3
printf("Expect len 3 -> %d\n", result);

result = strlen(""); // Expect 0
printf("Expect len 0 -> %d\n", result);
```

Listing 4.3: String Length Example

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

```
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 4.4: Structure definitions for time and date values.

The number of seconds since epoch is retrieved from PL031'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 4.5: 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.