Low-Level Programming on Linux (x86-64)

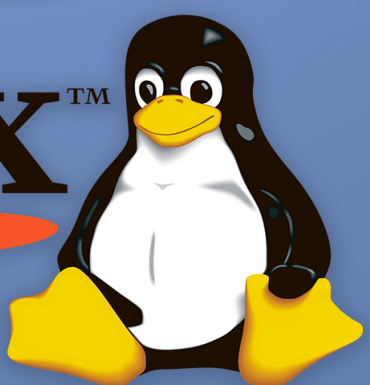
Minimal Runtime (No libc)

redhat

debian

10

Linux™



Prepared by Ayman Alheraki

https://simplifycpp.org

Low-Level Programming on Linux (x86-64) :

Minimal Runtime (No libc)

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

The purpose of this volume is to strip user-space execution on Linux down to its bare minimum—to expose, with absolute precision, what the kernel expects from a program at the instant it begins executing in user mode, and what the program must provide in return to execute correctly. This booklet demonstrates that beneath layers of toolchains, runtime libraries, startup files, dynamic loaders, and language abstractions, there exists a simple, well-defined, deterministic contract: the System V AMD64 ABI, the ELF object format, and the Linux system call interface.

Almost every program written in C or C++ relies silently on infrastructure that is rarely examined in detail. The compiler inserts hidden helpers. The linker pulls in startup objects. The dynamic loader resolves symbols. The C runtime initializes global state, executes static constructors, registers destructors, prepares thread-local storage, and finally transfers control into `main()`. All of this occurs before a single line of user code executes. For most developers, this machinery remains invisible—until it must be replaced, customized, or removed.

This book replaces it.

By constructing a complete user-space runtime in pure x86-64 assembly, using GAS with Intel syntax—without libc, without dynamic linking, without an unwinder, without PLT or GOT indirection, and without compiler runtime helpers—we expose the true structure of program execution on Linux. We recover `argc`, `argv`,

and `envp` directly from the initial stack created by the kernel. We enforce ABI-mandated stack alignment manually. We traverse and execute `.init_array` and `.fini_array` explicitly by walking raw memory. We implement an `atexit`-style handler mechanism from first principles. We invoke `main()` using nothing more than register conventions and a `call` instruction. Program termination is performed exclusively via the `sys_exit` system call—the kernel's final and authoritative termination interface.

Throughout this booklet, assembly is not used as illustration; it is the implementation. Every initialization step, every cleanup stage, every stack transition, and every system call is performed explicitly. Nothing is delegated to libc or to automatically generated startup code. Every instruction executed by the CPU is visible, intentional, and subject to inspection.

To ground these concepts, consider the minimal program entry point as seen directly by the kernel:

```
.intel_syntax noprefix
.global _start

_start:
    # On entry:
    # RSP -> argc
    #        argv[0]
    #        ...
    #        argv[argc]
    #        NULL
    #        envp[0]
    #        ...
    #        NULL
```

```
mov rdi, [rsp]          # rdi = argc
lea rsi, [rsp + 8]      # rsi = &argv[0]

# Align stack to 16 bytes before calling main
and rsp, -16

call main               # int main(int argc, char** argv)

# Return value from main is in eax
mov edi, eax            # exit status
mov eax, 60             # sys_exit
syscall
```

No hidden prologue exists. No runtime prepares the environment. The kernel transfers control directly to `_start`, and everything that follows is the responsibility of the program itself.
The value of this approach lies in clarity. By removing layers of abstraction, we gain a direct and precise understanding of:

- how the kernel constructs a process's initial execution context;
- how the System V AMD64 ABI governs register usage, stack layout, and calling conventions;
- how ELF metadata directs initialization and finalization;
- how constructor and destructor arrays operate independently of libc;
- how to issue system calls without dependent libraries;

- and ultimately, how minimal the requirements for a functioning C or C++ program truly are.

Once a minimal runtime can be constructed by hand, every higher-level mechanism becomes optional rather than mandatory. libc becomes a convenience library, not a prerequisite. Linker-generated startup code becomes a design choice, not an unquestioned assumption. Program execution becomes fully understandable, fully transparent, and fully controlled by the developer.

This final booklet in the series completes a progression that began with CPU architecture, assembly fundamentals, process creation, dynamic linking, ELF inspection, kernel boundary rules, system call execution, and performance constraints. The runtime developed here serves both as a teaching instrument and as a foundation—a starting point for custom execution environments, embedded runtimes, specialized loaders, research kernels, or language-specific systems.

The chapters that follow are written with academic rigor and a strict focus on low-level correctness. All assembly examples use GAS with Intel syntax and target the x86-64 architecture. The objective is not minimalism for its own sake, but minimalism for correctness: to construct a runtime that behaves exactly as required by the kernel and the ABI—no more, and no less.

If you understand this runtime, you understand the essence of program execution on Linux. Everything else is engineering layered on top.

Stay Connected

For more discussions and valuable content about **Performance Optimization and SIMD**, I invite you to follow me on **LinkedIn**:

https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Ayman Alheraki

1 Program Startup State

1.1 Stack Layout

A Linux process entering user space without a runtime—no `crt1.o`, no `libc`, and no dynamic linker assistance—begins execution in a state defined entirely by the kernel and the x86-64 System V ABI. Execution starts at the ELF entry point with the stack already populated and aligned according to strict ABI rules. Understanding this initial stack image is essential for building a minimal runtime, because it is the only structured state available before constructing a custom environment, invoking system calls, or establishing a heap.

At entry, the CPU registers contain no guaranteed values except for the instruction pointer targeting `_start` and the stack pointer `rsp` referencing the first quadword containing the argument count. The kernel guarantees 16-byte alignment *before* transferring control; because `_start` is reached via a direct jump rather than a `call`, the ABI condition at entry is:

```
rsp % 16 == 8
```

This constraint is fundamental. Any subsequent `call` or ABI-conformant function entry must reestablish correct alignment explicitly.

1.1.1 The Canonical Stack Image at `_start`

The kernel constructs the initial stack in descending memory order. The layout is linear, null-terminated, and contains no padding:

- `argc`
- `argv[]` pointers (terminated by NULL)
- `envp[]` pointers (terminated by NULL)
- auxiliary vector pairs `(tag, value)`
- terminating `AT_NULL`

Every element is naturally aligned for 64-bit access. No metadata beyond this structure is provided. In a no-libc environment, the runtime must parse this memory manually to recover arguments, environment variables, and auxiliary vector entries such as `AT_PHDR`, `AT_PHNUM`, and `AT_ENTRY`.

1.1.2 Extracting `argc`, `argv`, and `envp` in Pure Assembly

A minimal runtime must derive all startup parameters directly from the stack. The following example demonstrates reading `argc` and `argv[]` at program entry and issuing a raw `write` system call without libc.

```
.intel_syntax noprefix
.global _start

.section .bss
.align 8
argc_val:    .quad 0
```

```
argv_base:  .quad 0

.section .text
_start:
    mov rbx, rsp              # initial stack pointer
    mov rax, [rbx]            # argc
    mov [argc_val], rax

    lea rcx, [rbx + 8]        # &argv[0]
    mov [argv_base], rcx

    # write argv[0] to stdout

    mov rdi, 1                # stdout
    mov rsi, [rcx]            # argv[0]
    xor rdx, rdx              # length counter

len_loop:
    cmp byte ptr [rsi + rdx], 0
    je write_arg
    inc rdx
    jmp len_loop

write_arg:
    mov eax, 1                # sys_write
    syscall

    xor edi, edi
```

```
    mov eax, 60                 # sys_exit
    syscall
```

This code validates the kernel-defined stack contract and demonstrates that a program can execute fully without any runtime support beyond the kernel itself.

1.1.3 **Navigating the Environment and Auxiliary Vector**

After `argv[]` terminates, the next pointer marks the beginning of `envp[]`. It follows the same format: an array of pointers ending in NULL. Immediately after `envp[]` comes the auxiliary vector, organized as consecutive 16-byte pairs:

- a 64-bit tag
- a 64-bit value

The auxiliary vector provides critical metadata such as page size, program headers, load addresses, and entropy. A minimal runtime relies on entries like `AT_PHDR` and `AT_PHNUM` to introspect its own ELF layout.

```
.intel_syntax noprefix
.global parse_auxv

parse_auxv:
    mov rbx, rdi                # pointer to auxv start

aux_loop:
    mov rax, [rbx]              # tag
    test rax, rax               # AT_NULL
    je aux_done
```

```
    mov rcx, [rbx + 8]          # value
    # handle tag-specific logic here

    add rbx, 16
    jmp aux_loop

aux_done:
    ret
```

1.1.4 Stack Integrity and Alignment Discipline

Because no compiler-generated prologue exists, every function must preserve ABI alignment manually. Before any `call`:

```
# Required condition
rsp % 16 == 8
```

A robust startup sequence enforces alignment explicitly:

```
and rsp, -16
sub rsp, 8
```

Failure to uphold this invariant results in undefined behavior, particularly when SSE or AVX registers are used.

1.1.5 Why the Stack Layout Matters in a No-libc Environment

In a standard program, the C runtime parses the stack automatically. In a minimal runtime, the stack is the sole structured input. Argument handling, environment

parsing, auxiliary vector decoding, and loader construction all begin here. A precise understanding of the entry stack is therefore non-negotiable.

1.2 argc / argv / envp Retrieval

At the ELF entry point, the kernel provides no abstractions. The stack must be interpreted directly.

1.2.1 Recovering `argc`

```
.intel_syntax noprefix
.global _start

.section .bss
.align 8
argc_slot: .quad 0

.section .text
_start:
    mov rax, [rsp]
    mov [argc_slot], rax
```

1.2.2 Locating `argv[]`

```
lea rbx, [rsp + 8]      # &argv[0]
```

Iteration proceeds until a NULL pointer is encountered.

1.2.3 Retrieving `envp[]`

```
mov rcx, rbx
```

```
find_envp:
    mov rax, [rcx]
    test rax, rax
    je env_found
    add rcx, 8
    jmp find_envp

env_found:
    add rcx, 8          # &envp[0]
```

1.2.4 Passing Arguments to C

A minimal runtime must prepare registers according to the SysV ABI:

- rdi = argc
- rsi = argv
- rdx = envp

1.2.5 Calling `main` Without libc

```
.intel_syntax noprefix
.global _start
.extern main

.section .bss
argc_slot: .quad 0
argv_slot: .quad 0
envp_slot: .quad 0
```

```
.section .text
_start:
    mov rax, [rsp]
    mov [argc_slot], rax

    lea rcx, [rsp + 8]
    mov [argv_slot], rcx

    mov rbx, rcx
scan_envp:
    mov rdx, [rbx]
    test rdx, rdx
    je have_envp
    add rbx, 8
    jmp scan_envp

have_envp:
    add rbx, 8
    mov [envp_slot], rbx

    mov rdi, [argc_slot]
    mov rsi, [argv_slot]
    mov rdx, [envp_slot]

    and rsp, -16
    sub rsp, 8
```

```
    call main

    mov edi, eax
    mov eax, 60
    syscall
```

This code fully replaces `crt1.o` and the early libc startup layer. When linked with:

```
gcc -nostdlib -static start.s main.c -o program
```

the resulting binary behaves identically with respect to argument handling, while remaining completely independent of the C runtime.

1.2.6 **Rationale in a Minimal Runtime**

Manually passing arguments to C is the defining boundary between raw kernel execution and higher-level language semantics. Once this bridge is constructed correctly, all higher abstractions become optional rather than mandatory.

2 Writing `_start`

2.1 Register and Stack Preparation

A minimal runtime begins at `_start`, not at `main`. The Linux kernel transfers control directly to the program's entry address with no stack frame, no saved registers, no prologue, and no ABI-managed environment. In this raw context, the runtime must immediately validate and prepare the register and stack state so that all subsequent logic—argument parsing, auxiliary vector inspection, and the eventual call to `main`—executes under the strict guarantees of the System V AMD64 ABI. Linux does not preserve general-purpose registers across the transition to user mode. Aside from the instruction pointer and the stack pointer, no register contains defined data. The initial stack, however, is fully defined: the quadword at `[rsp]` holds `argc`, followed by the canonical stack layout described earlier. The first responsibility of the runtime is therefore to stabilize access to this stack-provided information before performing any destructive modifications to `rsp`.

2.1.1 The ABI Constraint at Entry

The kernel guarantees a single invariant upon entry to `_start`:

```
rsp % 16 == 8
```

This misalignment is intentional. Because `_start` is entered via a direct jump rather than a `call`, the stack pointer is offset by eight bytes. When `_start` later performs a `call`, the pushed return address restores 16-byte alignment for the callee. All C functions—including `main`—must begin execution with `rsp % 16 == 0`. Violating this rule breaks ABI guarantees relied upon by vector instructions, stack-based locals, and compiler-generated prologues.
Immediately upon entry, the runtime must preserve the raw stack pointer:

```
.intel_syntax noprefix
.global _start

.section .text
_start:
    mov rbx, rsp        # preserve initial stack pointer
```

Any later realignment of `rsp` must not invalidate access to the kernel-provided stack layout.

2.1.2 **Register Clearing and Predictable State**

Linux provides no guarantees about the contents of general-purpose registers beyond `rsp`. Assuming registers contain zero or meaningful values introduces nondeterminism. A minimal runtime therefore explicitly clears registers used for counting, scanning, or indexing.
A common pattern is:

```
xor rax, rax
xor rcx, rcx
xor rdx, rdx
```

While not mandated by the ABI, explicit initialization prevents subtle bugs when implementing argument parsing, auxiliary vector scanning, or custom allocators.

2.1.3 **Stabilizing Access to the Kernel-Provided Stack Block**

Because `_start` must eventually realign the stack before calling any function, the original `rsp` cannot be relied upon indefinitely. To preserve a stable view of the startup block, the runtime stores key pointers in static storage.

```
.section .bss
.align 8
argc_ptr:    .quad 0
argv_ptr:    .quad 0
envp_ptr:    .quad 0
```

The runtime then extracts and stores these values before modifying the stack:

```
_start:
    mov rbx, rsp
    mov rax, [rbx]          # argc
    mov [argc_ptr], rax

    lea rcx, [rbx + 8]      # &argv[0]
    mov [argv_ptr], rcx
```

At this stage, `rsp` must remain untouched. Any premature stack adjustment would corrupt the kernel-defined structure.

2.1.4 Preparing the Stack for Internal Calls

Before invoking any helper routine—assembly or C—the runtime must restore ABI-compliant alignment. The most predictable sequence is:

```
and rsp, -16          # align stack downward
sub rsp, 8            # prepare ABI-required misalignment
```

This ensures:

```
rsp % 16 == 8
```

immediately before a `call`, so that the callee begins execution with proper 16-byte alignment.

2.1.5 Example: Minimal Register and Stack Preparation Scaffold

The following sequence demonstrates a correct `_start` preparation suitable for a minimal runtime:

```
.intel_syntax noprefix
.global _start

.section .bss
.align 8
argc_ptr:    .quad 0
argv_ptr:    .quad 0
envp_ptr:    .quad 0

.section .text
_start:
```

```
    mov rbx, rsp                    # preserve entry stack

    mov rax, [rbx]                  # argc
    mov [argc_ptr], rax

    lea rcx, [rbx + 8]              # argv[]
    mov [argv_ptr], rcx

    mov rdx, rcx                    # scan for envp
find_envp:
    mov rsi, [rdx]
    test rsi, rsi
    je  have_envp
    add rdx, 8
    jmp find_envp

have_envp:
    add rdx, 8                      # &envp[0]
    mov [envp_ptr], rdx

    xor r8, r8
    xor r9, r9

    and rsp, -16
    sub rsp, 8
```

Once these invariants are enforced, the runtime can safely enter later phases such as auxiliary vector decoding and control transfer into C.

2.1.6 **Importance in a No-libc Environment**

A minimal ELF without libc lacks:

- startup code from `crt1.o`
- automatic stack alignment
- implicit argument forwarding
- register sanitization
- destructor or termination handlers

Every ABI rule must therefore be enforced explicitly from the very first instruction. Failure at this stage results in undefined behavior that propagates into all higher-level code.

2.2 Calling `main()` Manually

The kernel never calls `main`. In a no-libc environment, `_start` must bridge the gap between the kernel-defined startup state and the C ABI.
The System V AMD64 ABI defines the effective signature of `main` as:

```
int main(int argc, char **argv, char **envp);
```

Accordingly, `_start` must load:

- `argc` into **rdi**
- `argv` into **rsi**
- `envp` into **rdx**

2.2.1 Preparing Arguments for `main()`

```
mov rdi, [argc_ptr]
mov rsi, [argv_ptr]
mov rdx, [envp_ptr]
```

This is the complete ABI contract for invoking `main()`.

2.2.2 Calling `main()`

```
.extern main

    and rsp, -16
    sub rsp, 8
    call main
```

Upon return, the result is placed in `eax`.

2.2.3 Exiting via `sys_exit`

In a no-libc runtime, termination must be performed using a raw system call.

```
mov edi, eax        # exit status
mov eax, 60         # sys_exit
syscall
```

This is the definitive termination mechanism on Linux x86-64.

2.2.4 Complete Minimal Startup

```
.intel_syntax noprefix
.global _start
.extern main

.section .bss
.align 8
argc_ptr: .quad 0
argv_ptr: .quad 0
envp_ptr: .quad 0

.section .text
_start:
    mov rbx, rsp

    mov rax, [rbx]
    mov [argc_ptr], rax

    lea rcx, [rbx + 8]
```

```
    mov [argv_ptr], rcx

    mov rdx, rcx
scan_envp:
    mov rsi, [rdx]
    test rsi, rsi
    je have_envp
    add rdx, 8
    jmp scan_envp

have_envp:
    add rdx, 8
    mov [envp_ptr], rdx

    mov rdi, [argc_ptr]
    mov rsi, [argv_ptr]
    mov rdx, [envp_ptr]

    and rsp, -16
    sub rsp, 8

    call main

    mov edi, eax
    mov eax, 60
    syscall
```

2.2.5 **Significance**

This bridge between `_start` and `main` is the precise boundary between kernel-driven execution and the C language execution model. Every minimal runtime must construct this boundary manually. Once done correctly, all higher-level abstractions become optional rather than mandatory.

3 Initialization and Finalization

3.1 `.init_array` Execution

A minimal runtime that omits libc and all startup objects must explicitly execute global constructors before transferring control to `main()`. On modern ELF systems—including Linux on x86-64—the `.init_array` section is the canonical mechanism through which C++ static constructors, compiler-generated initializers, and user annotations are registered. In a conventional environment, this work is performed implicitly by the dynamic loader or by `crt1.o`. In a no-libc runtime, the responsibility belongs entirely to the program itself.
The `.init_array` section consists of a contiguous sequence of 64-bit function pointers. Each entry refers to a constructor with the signature:

```
void (*ctor)(void)
```

These functions must execute **before** the first line of user C or C++ code. The runtime must not pass arguments, must preserve stack alignment across calls, and must obey the System V AMD64 ABI.

3.1.1 Discovering `.init_array` Boundaries

The linker emits two symbols defining the bounds of `.init_array`:

```
__init_array_start
__init_array_end
```

These symbols represent raw memory addresses and must be declared as externals in assembly:

```
.extern __init_array_start
.extern __init_array_end
```

They are accessed using RIP-relative addressing to preserve position independence.

3.1.2 **Iterating Over the Constructor Array**

A minimal constructor runner loads the start and end addresses and walks the array linearly, calling each function pointer:

```
    lea rbx, [rip + __init_array_start]
    lea rcx, [rip + __init_array_end]

ctor_loop:
    cmp rbx, rcx
    je  ctor_done

    mov rax, [rbx]          # load constructor pointer
    call rax                # call void (*ctor)(void)

    add rbx, 8
    jmp ctor_loop

ctor_done:
```

This sequence:

- preserves ABI-required stack alignment,
- executes constructors in linker-defined order,
- assumes no callee-saved registers survive across calls.

3.1.3 Ensuring Stack Alignment During Constructor Execution

Before invoking any constructor, `_start` must already enforce:

```
rsp % 16 == 8
```

A standard precondition applied earlier in startup:

```
and rsp, -16
sub rsp, 8
```

This alignment must remain unchanged throughout constructor execution.

3.1.4 Complete Minimal `.init_array` Runner

```
.intel_syntax noprefix
.global run_init_array
.extern __init_array_start
.extern __init_array_end

.section .text
run_init_array:
    lea rbx, [rip + __init_array_start]
```

```
    lea rcx, [rip + __init_array_end]

.loop:
    cmp rbx, rcx
    je .done

    mov rax, [rbx]
    call rax

    add rbx, 8
    jmp .loop

.done:
    ret
```

This routine can be called directly from `_start` before `main()`.

3.1.5 Why `.init_array` Is Mandatory

Without executing `.init_array`, a no-libc binary violates C++ language guarantees:

- static objects remain uninitialized,
- global initialization expressions are skipped,
- toolchain-inserted hooks never execute.

Manual execution restores full language correctness while remaining runtime-independent.

3.2 `.fini_array` Execution

Finalization mirrors initialization. The `.fini_array` section contains destructors that must run **after** `main()` returns but **before** the process exits. In a no-libc runtime, this step must also be implemented manually.
Each entry has the signature:

```
void (*fini)(void)
```

Unlike constructors, destructors must execute in **reverse order**.

3.2.1 Locating `.fini_array` Boundaries

The linker provides:

```
__fini_array_start
__fini_array_end
```

Declared as:

```
.extern __fini_array_start
.extern __fini_array_end
```

3.2.2 Reverse-Order Execution

If entries are stored as:

```
A, B, C
```

they must execute as:

```
C → B → A
```

3.2.3 Reverse Iteration Implementation

```
.intel_syntax noprefix

    lea rbx, [rip + __fini_array_start]
    lea rcx, [rip + __fini_array_end]

    sub rcx, 8                # last entry

.loop:
    cmp rcx, rbx
    jb .done

    mov rax, [rcx]
    call rax

    sub rcx, 8
    jmp .loop

.done:
```

3.2.4 Encapsulated `.fini_array` Runner

```
.global run_fini_array
.extern __fini_array_start
.extern __fini_array_end

run_fini_array:
    lea rbx, [rip + __fini_array_start]
```

```
    lea rcx, [rip + __fini_array_end]

    sub rcx, 8

.loop:
    cmp rcx, rbx
    jb .done

    mov rax, [rcx]
    call rax

    sub rcx, 8
    jmp .loop

.done:
    ret
```

3.2.5 **Required Shutdown Order**

A correct no-libc shutdown sequence is:

```
call main
mov edi, eax
call run_fini_array
mov eax, 60
syscall
```

3.2.6 atexit-Style Handler List

Without libc, `atexit()` must be reimplemented if user-defined cleanup hooks are required. A minimal implementation maintains a LIFO array of function pointers.

3.2.7 Handler Storage

```
.intel_syntax noprefix

.section .bss
.align 8
atexit_list:  .zero 8 * 64
atexit_count: .quad 0
```

3.2.8 Registration Interface

```
.global my_atexit

my_atexit:
    mov rax, [atexit_count]
    cmp rax, 64
    jae .full

    lea rcx, [atexit_list + rax*8]
    mov [rcx], rdi

    inc rax
    mov [atexit_count], rax
```

```
    xor eax, eax
    ret

.full:
    mov eax, 1
    ret
```

3.2.9 **Executing Handlers**

```
.global run_atexit_handlers

run_atexit_handlers:
    mov rax, [atexit_count]
    test rax, rax
    je .done

    dec rax

.loop:
    lea rcx, [atexit_list + rax*8]
    mov rdx, [rcx]
    call rdx

    test rax, rax
    je .done
    dec rax
    jmp .loop
```

```
.done:
    ret
```

3.2.10 Final Shutdown Integration

```
call main
mov edi, eax
call run_fini_array
call run_atexit_handlers
mov eax, 60
syscall
```

3.2.11 Significance

Executing `.init_array`, `.fini_array`, and atexit-style handlers restores full C and C++ object lifetime semantics in a no-libc environment. This distinction separates a merely *functional* binary from a *correct* one, while preserving complete control over execution and termination.

4 Linker Script

4.1 Section Placement

A minimal runtime without libc requires explicit and complete control over how the ELF image is laid out in memory. Default linker scripts supplied by `ld` assume the presence of libc, a dynamic loader, relocation processing, exception metadata, and runtime startup files. When building a standalone no-libc ELF, these assumptions are invalid.
The developer must therefore define explicitly:

- which sections exist,
- where they are placed in virtual memory,
- how they are aligned,
- and how runtime code locates them at execution time.

This control is achieved through a custom **linker script**. The most fundamental responsibility of such a script is **section placement**.
Section placement determines:

- physical and virtual addresses of code and data,

- loadable segment boundaries,
- visibility of constructor and destructor arrays,
- alignment guarantees relied upon by low-level assembly,
- correspondence between file offsets and runtime addresses.

Incorrect placement can render `_start`, `.init_array`, `.fini_array`, or global data unusable.

4.1.1 **Defining the Memory Layout**

At minimum, a no-libc ELF must define the placement of:

- `.text` — executable code
- `.rodata` — read-only constants
- `.data` — writable initialized data
- `.bss` — zero-initialized data
- `.init_array` — constructor pointers
- `.fini_array` — destructor pointers
- optional runtime-specific sections

Each section must satisfy both ELF loader constraints and CPU alignment rules. On x86-64, code is typically aligned to 16 bytes, while pointer arrays require 8-byte alignment.
A minimal layout begins at a clean page boundary:

```
SECTIONS
{
    # Base virtual address
    . = 0x400000;
```

This mirrors the canonical static, PIE-disabled mapping used by Linux.

4.1.2 Placing the Code Section (`.text`)

The ELF entry point must refer to the first instruction of `_start`. This is enforced by placing the entry symbol at the head of `.text`:

```
    .text ALIGN(16) :
    {
        *(.text._start)
        *(.text*)
    }
```

Assembly code intended for the entry point must explicitly use the matching section name:

```
.section .text._start
.global _start
_start:
    # minimal startup code
```

This guarantees deterministic entry placement.

4.1.3 Read-Only Data (`.rodata`) and RIP-Relative Access

x86-64 code relies heavily on RIP-relative addressing. To avoid dynamic relocations, `.rodata` must be placed close to `.text`:

```
.rodata ALIGN(16) :
{
    *(.rodata*)
    *(.gnu.linkonce.r*)
}
```

This ensures instructions such as:

```
lea rax, [rip + msg]
```

resolve statically without GOT or loader intervention.

4.1.4 Writable Data: `.data` and `.bss`

Writable sections must be mapped into a loadable segment with read/write permissions:

```
.data ALIGN(16) :
{
    *(.data*)
}

.bss ALIGN(16) (NOLOAD) :
{
    *(.bss*)
    *(COMMON)
}
```

The NOLOAD directive ensures that .bss occupies memory but does not consume file space, matching kernel expectations.

4.1.5 Constructor and Destructor Arrays

Manual execution of .init_array and .fini_array requires precise placement and visible boundaries:

```
.init_array ALIGN(8) :
{
    __init_array_start = .;
    *(.init_array*)
    __init_array_end = .;
}

.fini_array ALIGN(8) :
{
    __fini_array_start = .;
    *(.fini_array*)
    __fini_array_end = .;
}
```

These linker-defined symbols are consumed directly by the no-libc runtime via RIP-relative addressing.

4.1.6 Custom Runtime Sections

Minimal runtimes often introduce custom sections:

- .rtdata — runtime metadata (atexit lists)

- `.syscalls` — syscall stubs
- `.stubs` — C-callable assembly helpers

Example placement:

```
.rtdata ALIGN(8) :
{
    *(.rtdata*)
}
```

This preserves deterministic offsets and simple access patterns.

4.1.7 **Minimal Linker Script Example**

```
ENTRY(_start)

SECTIONS
{
    . = 0x400000;

    .text ALIGN(16) :
    {
        *(.text._start)
        *(.text*)
    }

    .rodata ALIGN(16) :
    {
        *(.rodata*)
```

```
    }

    .data ALIGN(16) :
    {
        *(.data*)
    }

    .bss ALIGN(16) (NOLOAD) :
    {
        *(.bss*)
        *(COMMON)
    }

    .init_array ALIGN(8) :
    {
        __init_array_start = .;
        *(.init_array*)
        __init_array_end = .;
    }

    .fini_array ALIGN(8) :
    {
        __fini_array_start = .;
        *(.fini_array*)
        __fini_array_end = .;
    }
}
```

This script produces a static ELF requiring no dynamic loader, no relocations, and no runtime libraries.

4.1.8 **Discarding Unused Sections**

A no-libc ELF must explicitly discard sections associated with dynamic linking, exception handling, and toolchain metadata.

```
/DISCARD/ :
{
    *(.eh_frame*)
    *(.gcc_except_table*)
    *(.note*)
    *(.comment)
    *(.interp)
    *(.dynamic)
    *(.dynsym)
    *(.dynstr)
    *(.gnu.hash)
    *(.hash)
    *(.plt*)
    *(.got*)
    *(.rel*)
    *(.rela*)
}
```

This guarantees that no runtime features are implied but unsupported.

4.1.9 Static Linking Considerations

A minimal runtime must be fully statically resolved. All symbols must be defined at link time, and no dynamic relocations may exist.
Required compiler and linker flags:

```
-static -nostdlib -nostartfiles -nodefaultlibs
```

Additionally, compiler-generated helpers must be suppressed:

```
-fno-stack-protector
-fno-exceptions
-fno-asynchronous-unwind-tables
-fno-builtin
```

Any remaining helpers must be implemented manually.

4.1.10 Static ELF Assembly Demonstration

```
.intel_syntax noprefix
.global _start

.section .text._start
_start:
    mov edi, 1
    lea rsi, [rip + msg]
    mov edx, 6
    mov eax, 1
    syscall

    mov edi, 0
```

```
    mov eax, 60
    syscall

.section .rodata
msg: .ascii "Hello\n"
```

This program links cleanly into a deterministic static ELF using the custom linker script.

4.1.11 **Summary**

In a minimal no-libc model:

- everything resolves at link time,
- no dynamic sections exist,
- all memory layout is explicit,
- all runtime behavior is predictable.

The linker script is therefore not an auxiliary artifact—it is the *structural definition* of the runtime itself.

5 Final Project

5.1 Complete Minimal C Runtime + Demonstration Program

A minimal C runtime is the smallest execution environment in which a C program can operate on Linux without relying on libc, dynamic loaders, compiler-inserted startup code, or implicit ABI scaffolding. This section presents a production-grade minimal runtime—written entirely in human-authored x86-64 assembly—together with a demonstration program showing how conventional C code executes correctly within this environment.
This final project consolidates all previous chapters:

- manual ELF section placement via a custom linker script,
- startup state recovery from the kernel-provided stack,
- explicit `.init_array` and `.fini_array` execution,
- atexit-style cleanup handling,
- explicit entry into and exit from `main()`.

The result is a fully transparent and deterministic user-space execution model in which every executed instruction is under developer control.

5.1.0.1 **Minimal Runtime Philosophy**

Traditional C programs depend on extensive hidden infrastructure:

- `crt1.o` / `crti.o` entry code,
- dynamic loader relocation processing,
- libc initialization and teardown,
- `__libc_start_main`,
- automatic constructor and destructor execution,
- TLS initialization,
- exception unwinding metadata,
- compiler-generated runtime helpers.

A minimal runtime must instead:

- decode `argc`, `argv`, and `envp` manually,
- enforce ABI stack alignment explicitly,
- execute `.init_array` and `.fini_array` itself,
- provide an explicit atexit-style handler mechanism,
- use raw system calls only,
- manage all entry and exit obligations directly.

This reduction exposes the true requirements of the C execution model and separates language semantics from runtime convenience.

5.1.0.2 **Runtime Execution Timeline**

The complete runtime sequence is:

1. Kernel transfers control to `_start`.
2. Runtime extracts `argc`, `argv`, `envp`.
3. Stack alignment is enforced.
4. `.init_array` constructors execute.
5. `main(argc, argv, envp)` executes.
6. Return value captured from `eax`.
7. `.fini_array` destructors execute.
8. atexit handlers execute.
9. Process terminates via `sys_exit`.

This mirrors the abstract C/C++ language model without libc.

5.1.0.3 **Goals of the Minimal Runtime**

- absolute transparency,
- deterministic behavior,
- complete independence from system libraries,
- ABI and language correctness,
- extreme compactness,

- instructional clarity,
- extensibility toward custom runtimes.

5.1.0.4 **Full Minimal Runtime Assembly**

```
.intel_syntax noprefix
.global _start
.extern main
.extern __init_array_start
.extern __init_array_end
.extern __fini_array_start
.extern __fini_array_end

.section .bss
.align 16
argc_slot:    .quad 0
argv_slot:    .quad 0
envp_slot:    .quad 0

atexit_list:  .zero 8 * 64
atexit_count: .quad 0

.section .text._start
_start:
    mov rbx, rsp                # preserve initial stack

    mov rax, [rbx]              # argc
    mov [argc_slot], rax
```

```
    lea rcx, [rbx + 8]          # argv[]
    mov [argv_slot], rcx

    mov rdx, rcx
find_envp:
    mov rsi, [rdx]
    test rsi, rsi
    je have_envp
    add rdx, 8
    jmp find_envp

have_envp:
    add rdx, 8
    mov [envp_slot], rdx

    and rsp, -16
    sub rsp, 8

    call run_init_array

    mov rdi, [argc_slot]
    mov rsi, [argv_slot]
    mov rdx, [envp_slot]
    call main

    mov edi, eax
```

```
    call run_fini_array
    call run_atexit_handlers

    mov eax, 60
    syscall

.global run_init_array
run_init_array:
    lea rbx, [rip + __init_array_start]
    lea rcx, [rip + __init_array_end]
.init_loop:
    cmp rbx, rcx
    je .done
    mov rax, [rbx]
    call rax
    add rbx, 8
    jmp .init_loop
.done:
    ret

.global run_fini_array
run_fini_array:
    lea rbx, [rip + __fini_array_start]
    lea rcx, [rip + __fini_array_end]
    sub rcx, 8
.fini_loop:
    cmp rcx, rbx
    jb .done
```

```
    mov rax, [rcx]
    call rax
    sub rcx, 8
    jmp .fini_loop
.done:
    ret

.global my_atexit
my_atexit:
    mov rax, [atexit_count]
    cmp rax, 64
    jae .full
    lea rcx, [atexit_list + rax*8]
    mov [rcx], rdi
    inc rax
    mov [atexit_count], rax
    xor eax, eax
    ret
.full:
    mov eax, 1
    ret

.global run_atexit_handlers
run_atexit_handlers:
    mov rax, [atexit_count]
    test rax, rax
    je .done
    dec rax
```

```
.loop:
    lea rcx, [atexit_list + rax*8]
    mov rdx, [rcx]
    call rdx
    test rax, rax
    je .done
    dec rax
    jmp .loop
.done:
    ret
```

5.1.0.5 Demonstration C Program

```
extern int my_atexit(void (*fn)(void));
long write_sys(int fd, const char *buf, long len);

void cleanup(void) {
    write_sys(1, "cleanup\n", 8);
}

int main(int argc, char **argv, char **envp) {
    my_atexit(cleanup);
    write_sys(1, "Hello from main\n", 16);
    return 5;
}
```

5.1.0.6 Minimal Syscall Wrapper

```
long write_sys(int fd, const char *buf, long len) {
    long ret;
    asm volatile (
        "mov $1, %%rax\n\t"
```

```
        "syscall"
        : "=a"(ret)
        : "D"(fd), "S"(buf), "d"(len)
        : "rcx", "r11", "memory"
    );
    return ret;
}
```

5.1.0.7 **Runtime Execution Trace**

Output:

```
Hello from main
cleanup
```

Exit status:

```
$ echo $?
5
```

5.1.0.8 **Significance**

This runtime:

- is fully correct and ABI-compliant,
- supports C++ static initialization,
- handles deterministic shutdown,
- requires no runtime libraries,
- forms the foundation for advanced systems work.

5.1.0.9 **Next Extensions**

- custom heap allocator,
- signal handling,
- formatted output,
- minimal C library construction,
- custom loaders,
- optional exception support.

This project demonstrates that a correct C runtime on Linux can be built from first principles with absolute transparency.

Conclusion

A minimal runtime represents the purest expression of user-space execution on Linux. By deliberately removing every external dependency — libc, dynamic loaders, compiler-generated startup code, implicit initialization and teardown logic, symbol resolution infrastructure, exception metadata, and runtime helpers — this book exposed the structural foundation of what it means for a process to execute under the System V AMD64 ABI. What remains is a runtime governed entirely by explicit, deterministic assembly code rather than opaque layers of tooling.

Through the manual construction of `_start`, all mechanisms commonly assumed to be part of "the language" were replaced with mechanisms directly mandated by the operating system and the processor architecture. Recovering the kernel-supplied stack layout, enforcing ABI-compliant stack alignment, invoking `.init_array` constructors in strict order, calling `main()` using precise register conventions, executing `.fini_array` destructors, running atexit-style handlers, and terminating the process via `sys_exit` — each of these operations constitutes a minimal and essential component of the contract between Linux and a well-formed ELF executable. Nothing more is required, and nothing less is correct.

This book demonstrated that neither C nor C++ inherently requires a standard library or hidden runtime machinery. They require only that ABI rules are respected, memory is accessed deterministically, and the execution environment presented to

`main()` is coherent and well-defined. Everything beyond this — heap management, I/O abstractions, threading models, exception systems, and high-level libraries — is a matter of policy layered atop these fundamentals. A minimal runtime isolates these essentials, allowing higher layers to be designed intentionally rather than inherited implicitly.

The assembly presented throughout this volume illustrated not only correctness, but discipline. Every instruction was intentional: RIP-relative addressing preserved static layout stability, explicit stack alignment enforced architectural guarantees, and direct syscalls replaced large dependencies with compact, predictable sequences. By manually managing constructor and destructor arrays, the runtime restored the language's global object lifetime semantics without relying on libc's orchestration. The custom atexit mechanism extended these guarantees to user-defined cleanup routines, completing a fully freestanding initialization and finalization model.

The resulting runtime is small enough to fit within a single page of assembly, yet complete enough to execute real C programs, support static C++ objects, perform deterministic shutdown, and terminate with correct semantics. It is not merely an academic exercise; it is a reference model for understanding how user code transitions from kernel entry to fully running software — and how control is ultimately returned to the kernel.

More importantly, mastering such a runtime equips the systems programmer with insights that extend far beyond minimalism itself. It clarifies how modern compilers assume specific invariants, how linkers construct the execution image, how the kernel transfers control into user space, how ELF metadata governs initialization flow, and how language-level abstractions depend on precise low-level conventions. These insights are foundational for writing loaders, building custom operating environments, developing embedded systems, implementing language runtimes,

and auditing complex binaries.

This tenth book concludes the core series by enabling the reader to construct a complete, correct, and efficient C runtime by hand — from the first instruction executed to the final system call. It is an invitation to build higher-level systems consciously, with full understanding of what lies beneath. With this knowledge, libc, startup files, and conventional runtime behavior no longer appear monolithic or opaque; they become replaceable components and optional tools rather than unquestioned assumptions.

Minimalism here is not reduction for its own sake; it is clarity.

It is the architectural baseline upon which all higher abstractions can be built with confidence.

With this foundation established, the reader is now equipped to design new runtime environments, experiment with unconventional execution models, build custom loaders, or even create new programming languages — always with a precise understanding of the contract between the kernel, the ABI, and the executable's own logic.

References

The development of a fully manual, no-libc runtime requires precise and authoritative knowledge of processor architecture, executable formats, linking models, and system call conventions. The references listed below constitute the foundational technical material necessary to design, implement, and verify a runtime environment at the level demonstrated in this booklet. They include processor manuals, ABI specifications, ELF documentation, linker and toolchain references, and classic systems programming texts that directly inform low-level assembly development, linker script construction, and ABI correctness.
These sources do *not* focus on high-level abstractions or libc usage. Instead, they describe the underlying mechanisms that allow a minimal runtime to exist independently of any standard library or runtime framework.

Architecture and Processor Manuals

- **AMD64 Architecture Programmer's Manual (Volumes 1–5)**

  The definitive specification of the x86-64 architecture, covering instruction semantics, register usage, calling conventions, stack alignment rules, system-level behavior, and RIP-relative addressing. Essential for understanding ABI constraints, leaf-function behavior, register preservation, and alignment

guarantees required for hand-written assembly.

- **Intel® 64 and IA-32 Architectures Software Developer's Manual**

  A complementary reference detailing instruction encoding, system call interfaces, memory model behavior, and low-level execution assumptions. Critical for constructing predictable startup and execution sequences in `_start`.

Linux Kernel and User-Space ABI References

- **System V Application Binary Interface AMD64 Architecture Supplement**

  Defines the canonical calling conventions for C and C++ on x86-64 Linux, including register usage, stack alignment requirements, red-zone rules, and application startup assumptions. This document forms the core contract between `_start`, `main()`, and ABI-compliant user code.

- **Linux Kernel System Call Interface Documentation**

  Describes raw system calls, register-based parameter passing, return value conventions, error signaling, and architectural considerations. This is the primary reference for implementing direct syscalls such as `sys_exit` and `sys_write` without libc.

- **Linux man-pages (Sections 2 and 7)**

  Provide authoritative definitions of Linux system calls and ABI-related behavior, including process startup, environment handling, signal delivery, and ELF loading rules. Although concise, these pages define the canonical runtime environment for user-space processes.

ELF Format, Linking, and Loader Behavior

- **Executable and Linking Format (ELF) Specification**

  Defines ELF headers, program headers, section semantics, symbol resolution rules, and loader behavior. Fundamental for understanding how `_start` is invoked, how `.init_array` and `.fini_array` are represented, and how static linking produces a self-contained executable image.

- **Linkers and Loaders (John R. Levine)**

  A detailed technical treatment of linkers, loaders, relocation models, symbol resolution, and object file formats. This work explains why minimal runtimes must override default startup objects, control section layout explicitly, and eliminate dynamic linking metadata.

- **GNU ld Linker Manual**

  Documents linker script syntax, section placement rules, PHDR definitions, symbol assignment, `PROVIDE` directives, and `DISCARD` usage. All section placement and elimination strategies used in this booklet rely on capabilities described in this manual.

- **Binutils Documentation (objdump, readelf, nm)**

  Authoritative reference for toolchain utilities used to inspect, verify, and validate ELF binaries. Essential for confirming the presence and layout of `.text`, `.data`, `.bss`, `.init_array`, `.fini_array`, and the absence of dynamic sections in a no-libc binary.

Compiler and Toolchain References

- **GCC Internals and Command-Line Documentation**

  Describes how the compiler emits startup code, built-in function calls, exception tables, PLT stubs, stack protection, and unwind metadata. Required for suppressing unwanted code generation using options such as `-nostdlib`, `-fno-exceptions`, and `-fno-stack-protector` when building a minimal runtime.

- **Clang/LLVM Documentation**

  Provides insight into backend code generation, assembly emission, builtin lowering, and target-specific behavior. Useful for ensuring that compiler output remains compatible with a strict no-libc, no-PLT environment.

- **DWARF Debugging Standard**

  Although not used directly in this runtime, understanding DWARF unwind semantics helps prevent accidental inclusion of `.eh_frame` and related exception metadata that must be removed in a minimal design.

Systems Programming and Low-Level Texts

- **Advanced Programming in the UNIX Environment (W. Richard Stevens)**

  A classic reference on UNIX process lifecycles, environment variables, file descriptors, and system call behavior. Although libc-oriented, the concepts map directly to the primitives manually implemented in this runtime.

- **The Linux Programming Interface (Michael Kerrisk)**

  A comprehensive reference on kernel and user-space interaction. Particularly relevant for syscall behavior, memory management, environment setup, and process control when reimplementing these features without libc.

- **Computer Systems: A Programmer's Perspective**

  Provides conceptual grounding in stack frames, calling conventions, binary formats, linking, and program execution. Strong supporting material for readers constructing their own minimal runtime.

Source-Level and Implementation References

- **Linux Kernel Source Tree**

  In particular, `fs/binfmt_elf.c` and `arch/x86/entry/`. These sources define ELF loading, initial stack construction, argument vector setup, auxiliary vector population, and syscall entry paths. They reveal the exact execution context in which `_start` is entered.

- **glibc and musl Source Trees**

  Not used directly in this runtime, but valuable for contrast and study. These implementations demonstrate how large runtimes handle constructors, destructors, TLS initialization, and system call abstractions — mechanisms reimplemented manually in this booklet.

- **GNU Assembler (GAS) Documentation**

  Defines assembler directives, section assignment rules, alignment constraints, symbol visibility, and low-level assembler behavior consistent with the Intel syntax used throughout this work.

Scope and Intent of These References

The references above collectively support:

- writing `_start` entirely by hand,
- replacing all libc-driven behavior with explicit assembly,
- executing `.init_array` and `.fini_array` manually,
- performing ABI-compliant calls into `main()`,
- constructing linker scripts with fully predictable layouts,
- and verifying ELF correctness through direct binary inspection.

None of these materials depends on non-kernel libraries or high-level frameworks. Instead, they describe the *true substrate* from which all user-space execution on Linux is constructed.