Starman OS — Scope of Work: POSIX Compatibility

A proposal for running existing Linux and Unix software on Starman

Prepared for: Andrew G. Watters
Prepared by: Grok Bot
Baseline: Starman 0.1 (x86-64, pure assembly)
Date: November 26, 2026
Version: v0.1 draft (for review)
Status: Proposal. Nothing here is committed or scheduled.

1. Executive summary

Recommendation. Do not try to hand-write a POSIX library and port every program to a new native API. Make the Starman kernel speak the **Linux x86-64 system-call interface** (the "Linux ABI") and run real Linux binaries unmodified. Start with static programs built against musl libc (a static "hello world", then BusyBox), then add dynamic linking so glibc-based binaries run. This is the route taken by the FreeBSD Linux emulation layer, illumos LX zones, gVisor, and small hobby kernels such as Kerla, and it gives the largest amount of existing software for the least porting work.

Keep the boot, interrupt-entry, context-switch and system-call entry code in **assembly**, since that is the part of Starman that makes it distinctive. Write the larger subsystems (memory manager, VFS, filesystems, TCP/IP, drivers) in **freestanding C**. A pure-assembly kernel is possible, but my estimate is that it would take roughly three to five times as many lines of source and about two to three times as long, and it could not reuse the C libraries that make Phases 3 and 8 cheaper.

My rough estimate for everything up to "Python, ssh, xterm and twm running" is **85–135 person-weeks** of focused work by one experienced developer, or about **75,000–135,000 lines** of kernel code. These are planning estimates, not measurements (Section 8). Firefox is a stretch goal with no committed estimate. Full POSIX *certification* is explicitly **not** a goal. The immediate next step is **Phase 2: memory management** (physical and virtual memory managers, per-process address spaces, a kernel heap), about 3–5 person-weeks, which Section 11 details.

2. Goals and non-goals

2.1 Goals

- **Run existing software.** In practice this means: a Linux x86-64 ELF executable, built by someone else for Linux and copied onto a Starman disk image, starts, runs and produces the same output it would on Linux, for the programs on the target ladder (Section 7). It does *not* mean "every Linux program."
- **POSIX behaviour where programs depend on it:** processes (`fork`, `execve`, `wait`), file descriptors, pipes, signals, terminals (termios, ptys), threads, sockets, `poll`/`select`/`epoll`, and the usual file and directory calls.
- **Stay small, readable and Starman-flavoured:** the boot path and the lowest layers remain assembly and remain well commented.
- **Run first in VMware Workstation** (Andrew's environment) and QEMU (for automated tests), on a small, fixed set of virtual hardware (Section 10).
- **Be testable:** every phase ends with acceptance tests that can be run headlessly and logged over the serial port.

2.2 Non-goals (stated plainly)

- **No POSIX certification.** Formal branding (the Open Group UNIX® or IEEE POSIX® programs) needs a passing conformance run, a formal application and a license, and those programs are a business process as much as an engineering one. Linux itself is not certified either. I am not proposing it. Starman would be described as "Linux-ABI compatible" and "mostly POSIX-conforming," never "POSIX certified."
- **No full Linux compatibility.** The Linux interface is several hundred system calls plus ioctls, `/proc`, `/sys`, netlink, namespaces, cgroups, seccomp, `io_uring` and more. Starman will implement the subset that the target software actually uses and return `ENOSYS` for the rest.
- **No Linux drivers, no GPU acceleration, no Wi-Fi,** no hardware beyond what VMware presents, and no suspend or power management beyond what ACPI gives for power-off.
- **No security hardening claims.** Basic user/kernel separation yes; a hardened multi-user system, no.
- **Not a Linux distribution.** Starman provides a kernel and a minimal root filesystem. Anything beyond that is bring-your-own binaries.

3. Key architectural decision

3.1 The options

Option	What it means	Advantages	Disadvantages
A. Native POSIX API	Starman defines its own syscall numbers and structures. A libc (musl or mlibc, ported) wraps them. Software is recompiled for Starman.	Free to design a clean, small interface. Smaller kernel surface. Same approach as SerenityOS.	Every program must be ported and rebuilt. Needs a cross-toolchain targeting Starman before anything runs. Large programs (Python, Firefox) become porting projects. No prebuilt binaries.
B. Linux ABI	The kernel implements Linux x86-64 syscall numbers, struct layouts, errno values, signal frames, ELF auxiliary vectors and enough of `/proc`. Unmodified Linux binaries run.	Immediate access to existing binaries and any Linux toolchain. Test with real Linux test suites. Proven by FreeBSD, illumos LX, gVisor, Kerla and others.	The interface is large and moves. Behaviour must match Linux's quirks, not only the standard. The design is constrained by Linux semantics (but see the hybrid).
C. Hybrid (recommended)	The kernel's native interface *is* the Linux ABI subset (B), while Starman's own userland tools are built with musl for that same ABI, plus optional Starman-only system calls in a reserved number range.	All of B's advantages, and Starman can still add its own features (for example, introspection calls) without breaking compatibility.	Needs discipline to keep the Starman-only extensions separate and documented.

3.2 Recommendation

Choose **C, the hybrid**, with B as its core. The decisive reason is the one in Andrew's request: *run existing software*. Under A, the first useful milestone (a shell) requires a native toolchain and libc port before any program runs. Under B, the first milestone needs only a Linux-built static binary and about two dozen system calls, and there is a strong ready-made supply of test programs. A kernel that follows the Linux ABI can later host a second personality if that is ever desired. The reverse is much harder.

Two further points favour B. First, Linux's own `COPYING` file states that user programs using the kernel through normal system calls are not, by that fact alone, derivative works of the kernel, and Starman would be reimplementing an interface, not copying kernel code (Section 9 covers licensing in more detail). Second, the userland to run is mostly written against POSIX *as implemented on Linux*, so tracking Linux's behaviour is more useful than tracking the written standard.

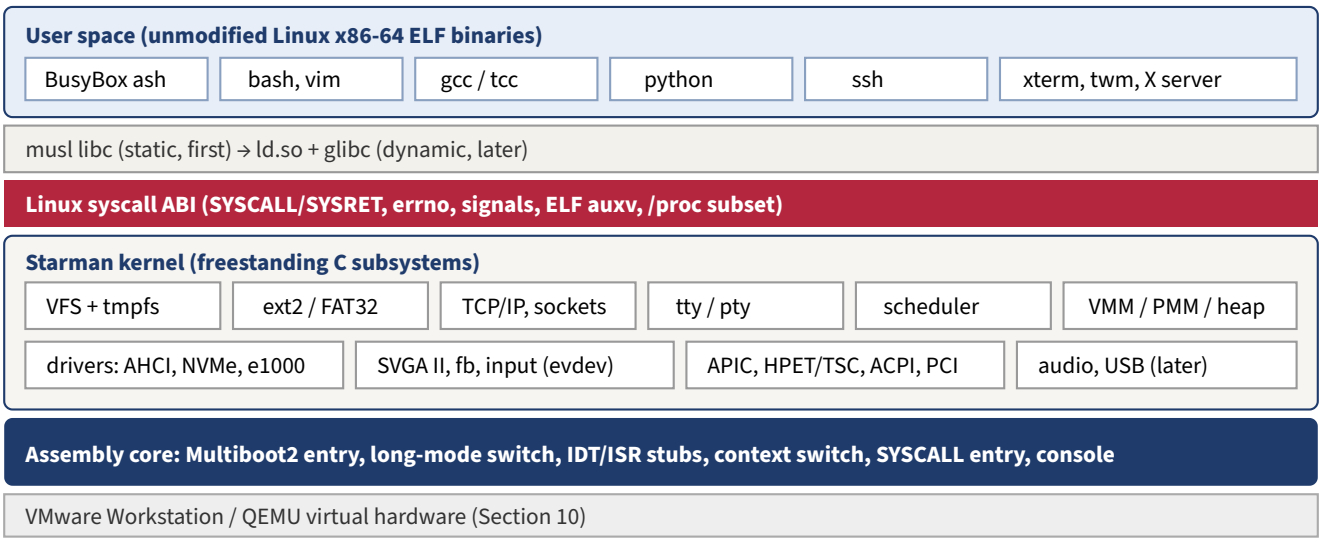


Figure 1. Proposed layering. The red band is the compatibility contract: everything above it is existing software, everything below it is Starman.

3.3 Assembly versus C

Starman 0.1 is 100% NASM and the project's character comes from that, so this is a real decision and not a technicality.

Recommendation: keep the *boot and lowest-level core in assembly*: Multiboot2 entry, long-mode switch, IDT and ISR stubs, context switch, the `SYSCALL` entry path, and the console and serial drivers already written. Allow *freestanding C* (`-ffreestanding -mno-red-zone -mcmodel=kernel`, no libc inside the kernel) for the subsystems where size and rate of change dominate: VFS, filesystems, networking, drivers. The two sides share a documented calling convention (System V AMD64), so assembly and C functions call each other directly. On RHEL 9, `gcc` and `binutils` are ordinary AppStream packages.

> **If Andrew insists on pure assembly.** It is feasible, as MenuetOS and KolibriOS show, but the costs are real and I would want them accepted up front. (1) **Volume:** as an estimate, 3–5× the source lines for the same functionality, so on the order of several hundred thousand lines for this scope. (2) **Schedule:** an estimate of 2–3× longer. (3) **No reuse:** lwIP, ACPI libraries, filesystem code and the like are written in C and could not be linked in, so each would be written from scratch. (4) **Bug rate and review:** no type checking, so a wrong register or stack offset corrupts memory silently. (5) **Debugging** a scheduler or TCP stack in assembly is slower. Mitigations would be a strict macro layer for structured code and calling conventions, and a large in-kernel self-test suite. Under pure assembly I would trim the ladder: stop at BusyBox, bash and vim, and treat networking, X11 and Python as separate decisions.

Userland is unaffected by this choice: BusyBox, Python and the rest are C programs compiled on Linux, and they run on Starman regardless of what the kernel is written in.

4. Current baseline: Starman 0.1

Starman 0.1 was built and tested in QEMU on both legacy BIOS and UEFI firmware (a VMware test is pending). It is about 2,560 lines of NASM (about 2,280 of them hand-written; the font and logo files are generated). What exists today:

Area	Present in 0.1	Missing for POSIX work
Boot	GRUB 2 and Multiboot2, hybrid BIOS/UEFI ISO, flat binary at 1 MiB	Higher-half kernel, ELF64 image, kernel command line parsing
CPU mode	32-bit entry, CPUID and long-mode check, 64-bit GDT, identity-mapped 4 GiB with 2 MiB pages	TSS, user segments, NX/W^X, `SYSCALL` MSRs, per-CPU data

Area	Present in 0.1	Missing for POSIX work
Memory	Boot memory map parsed and displayed (`mem`)	Physical allocator, virtual memory manager, address spaces, heap
Interrupts	256-entry IDT, exception reporter (number, name, error code, RIP, CR2), remapped 8259 PIC	APIC/IOAPIC, MSI, recoverable page faults, IRQ routing
Timing	PIT at 100 Hz tick counter	Calibrated high-resolution clock, timers, wall clock
Input	PS/2 keyboard, US layout, Shift/Caps/Ctrl	Key repeat, arrows and function keys, mouse, tty line discipline, USB HID
Output	VGA text or linear framebuffer (8×16 font), COM1 mirror	Terminal emulation (VT100-class), `/dev/fb0`
Shell	Built-in monitor: help, clear, echo, ticks/uptime, mem, cpuid, info, version, banner, fault, reboot, halt	Replaced by a real userland shell; the monitor survives as a debug console
Everything else	No processes or scheduler, no user mode, no system calls, no filesystem, no storage, network, USB or ACPI. Single CPU only.	

Starman 0.1 is a solid first stage, but the distance to "runs a static musl binary" is still the whole of Phases 2 through 5.

5. Phased roadmap

Each phase has a deliverable and acceptance tests that must pass in QEMU (BIOS and UEFI) before the next phase is called done. Phases overlap in practice, but this order respects the real dependencies. Estimates are in Section 8.

Phase 1 — Boot and shell (Starman 0.1)

DONE

Complete; see Section 4.

Phase 2 — Memory management

Next: Section 11

Scope: physical memory manager (PMM) from the boot memory map; virtual memory manager (VMM) with 4-level page tables, NX and W^X; a direct map of RAM; per-process address spaces (create, clone, destroy); page-fault handler; kernel heap (`kmalloc`/`kfree`). Introduces the C build and the higher-half kernel.

Deliverable: kernel that boots with all allocators online and self-tests. **Acceptance:** allocate and free 100,000 pages and 100,000 heap objects with no leak or overlap (counters return to baseline); NX and read-only mappings fault as expected; works with 128 MiB, 512 MiB and 8 GiB of RAM, BIOS and UEFI.

Phase 3 — Interrupts, timing and ACPI

Scope: parse ACPI tables (RSDP, MADT, FADT, HPET); local APIC and IOAPIC; APIC timer calibrated against the PIT, HPET or TSC; monotonic clock and wall clock (RTC); ACPI power-off (so `halt` really turns the VM off) and reset; PCI enumeration (bus, device, function, BARs, MSI). An existing ACPI library written in C (for example ACPICA or uACPI) is recommended over writing an AML interpreter.

Deliverable: tickless-capable timer and a device list. **Acceptance:** `uptime` agrees with the host clock to within a stated tolerance over 10 minutes; `halt` powers off QEMU and VMware; PCI list matches the guest hardware.

Phase 4 — Processes, scheduler, preemption

Scope: kernel threads; task structures; context switch (assembly); a simple preemptive round-robin scheduler, then priorities; sleep, wake and wait queues; mutexes and spinlocks; per-CPU data. SMP (AP startup, per-CPU run queues, TLB shootdowns) is *explicitly deferred* until the single-CPU system is stable.

Deliverable: multiple kernel threads running concurrently under preemption. **Acceptance:** stress test with many threads incrementing locked counters gives exact totals; timer-driven preemption of a busy loop; no lost wakeups in 1 hour of soak test.

Phase 5 — User mode, system calls, ELF loader

Scope: TSS, ring-3 segments, `SYSCALL`/`SYSRET` entry (assembly), copy-to/from-user with fault handling, ELF64 loader (static executables first), initial stack with argv, envp and auxv, `exit`. First system-call set, about 40 calls: `read`, `write`, `readv`, `writev`, `open`/`openat`, `close`, `fstat`/`newfstatat`, `lseek`, `mmap`, `munmap`, `mprotect`, `brk`, `ioctl`, `arch_prctl`, `set_tid_address`, `exit`, `exit_group`, `getpid`, `uname`, `clock_gettime`, `getrandom`, `rt_sigaction` and `rt_sigprocmask` (stubs at first); the process-creation calls follow in Phase 6. A static musl "hello world" typically needs only a handful of these (setting up thread-local storage, `ioctl` for the terminal check, `writev`, `exit_group`), which makes it a good first target.

Deliverable: a statically linked musl `hello` binary, built on RHEL 9, runs from an initial in-memory image. **Acceptance:** prints the expected text and exit status; a deliberately crashing program is killed without crashing the kernel; a syscall trace log (serial) matches the expected sequence; invalid user pointers return `EFAULT`.

Phase 6 — VFS, files, terminals and the process API

Scope: a virtual filesystem layer (inodes, dentries, mounts); tmpfs and an initramfs unpacker (cpio "newc") so the ISO can carry a root filesystem; devfs (`/dev/null`, `zero`, `random`, `urandom`, `tty`, `console`); ttys and ptys with a termios line discipline (canonical mode, echo, job control signals); pipes and FIFOs; `fork`/`vfork`/`clone`, `execve`, `wait4`; signals with Linux signal frames; copy-on-write `mmap` and file-backed mappings; `futex` and threads via `clone`; `poll`, `select`, `epoll`; `fcntl`, `dup`, `getdents64`, `chdir`/`getcwd`, `stat` family, permissions and uid/gid basics.

Deliverable: BusyBox `ash` boots as `init`, with its standard applets. **Acceptance:** an interactive shell on the console and over serial; pipelines (`ls | grep | wc`), job control (Ctrl+C, Ctrl+Z), redirection, background jobs; the BusyBox test suite subset passes; selected LTP syscall tests (Section 8) pass.

Phase 7 — Storage and persistent filesystems

Scope: block layer and page/buffer cache; AHCI (SATA) driver first, since VMware offers a SATA controller, then NVMe; MBR and GPT partition parsing; **ext2** read-write, then **ext4 read-only** (extents, no journal replay writes); **FAT32** (useful for the UEFI system partition and interchange). ext4 *write* support is deliberately out of scope for this proposal because of data-loss risk.

Deliverable: root filesystem on a virtual disk, with `mount`, `umount`, `sync`. **Acceptance:** create, write and re-read files across reboot; `fsck` from a Linux guest reports the ext2 volume clean after a stress test; read-only ext4 volume made by Linux mounts and checksums match.

Phase 8 — Networking

Scope: driver for the Intel e1000 family (the default adapter in VMware and available in QEMU); later vmxnet3 and virtio-net; Ethernet, ARP, IPv4, ICMP, UDP, TCP, DHCP client, DNS resolver config; BSD sockets API (`socket`, `bind`, `listen`, `accept`, `connect`, `send*`/`recv*`, `setsockopt`, `getsockname`) including Unix-domain sockets (needed by X11). **Stack choice:** port lwIP (BSD license, small, proven, but its socket semantics and performance differ from Linux's) *or* write a minimal stack of Starman's own. I recommend lwIP behind the kernel's socket layer first, because a custom TCP is a substantial project in its own right (retransmission, congestion control, edge cases), and replace it later only if it limits real programs. IPv6 is out of scope for the first pass.

Deliverable: `ping`, `wget`/`nc` and a TCP echo server from BusyBox. **Acceptance:** DHCP lease obtained in VMware NAT mode; fetch a file over HTTP and compare its checksum; 1 GB local transfer with no corruption.

Phase 9 — Dynamic linking, glibc and /proc, /sys

Scope: support for the ELF interpreter (`PT_INTERP`, `ld.so` from musl or glibc, running in user space), correct `mmap` and `mprotect` semantics for shared objects, thread-local storage and `clone` flags used by pthreads, auxv entries, optionally a small vDSO (or let libc fall back to real system calls); a minimal `/proc` (`self/exe`, `self/maps`, `self/fd`, `cpuinfo`, `meminfo`, `mounts`, `stat`) and a minimal `/sys`. Many programs fail only because one of these files is missing, so this list will grow based on observed failures.

Deliverable: a dynamically linked glibc program and Python (with extension modules) run. **Acceptance:** `ldd`-style tracing shows every library loaded; pthread, `dlopen` and TLS tests pass; Python's own test-suite subset runs.

Phase 10 — Graphics and input

Scope: `/dev/fb0` (the Linux fbdev ioctl interface over the Multiboot2 framebuffer first); evdev input devices (`/dev/input/event*`) fed from PS/2 keyboard and mouse; a VMware **SVGA II** driver for mode setting beyond what GRUB gives and for absolute pointer support (together with VMware's mouse protocol); then X11: first Xorg with the `fbdev` driver, or alternatively a small purpose-built X server, or an Xvfb-style server that renders to memory and a thin presenter to the framebuffer. X needs Unix sockets, shared memory (SysV or `mmap`), `select`/`poll` and signals, so most of the work is already done by Phases 6–9. 3D acceleration is out of scope (software rendering only).

Deliverable: graphical session with `xterm` and `twm`. **Acceptance:** window opens, keyboard and mouse work, a screenshot from QEMU matches a reference closely enough; `xterm` runs the shell for 30 minutes with no crash.

Phase 11 — Audio and USB (later)

Scope: Intel HD Audio (or the AC'97-style Ensoniq device VMware can present) with an OSS-style or ALSA-subset interface; USB host controllers (xHCI, plus EHCI/UHCI if needed), USB HID (keyboard, mouse) and mass storage. Needed for real hardware and for the SSL 18 interface that motivated the project, but not for VMware-only goals, so it comes last.

Acceptance: play a WAV file through the VM's audio device; a USB keyboard and a USB disk work in VMware.

6. Starman's design choices that matter early

- **Address-space layout:** higher-half kernel at `0xFFFFFFFF80000000`, a direct physical map in the upper half, user space in the lower half. This is the layout Linux-ABI programs and tools expect.
- **One syscall dispatch table** indexed by the Linux x86-64 numbers, with unimplemented entries returning `-ENOSYS` *and logging the number to the serial console*. That log is the single most useful tool for deciding what to implement next.

- **Behaviour first, completeness later:** prefer a correct, small implementation of each call to a broad, buggy one.
- **Clean-room discipline:** the Linux ABI is implemented from documentation, man pages and observed behaviour (`strace` on real Linux), not by copying kernel source (Section 9).
- **Reserved Starman range:** any Starman-specific system calls live in a reserved number range and are documented separately.
- **Serial-first debugging:** every subsystem logs over COM1 so VMware runs can be captured with a serial-to-file setting.

7. Target software ladder

Each rung is a concrete "it works" demonstration and pulls in the next set of kernel features. Rungs 1 to 3 are the committed core; rungs 4 to 8 are the target; rung 9 is a stretch goal.

#	Program	What it demonstrates / needs	Phase	Notes
1	**hello** (static, musl)	ELF loading, ~10–40 syscalls, console tty	5	First milestone. Built on RHEL 9 with `musl-gcc` or a musl cross-toolchain.
2	**BusyBox** (static, musl)	fork/exec/wait, pipes, termios, signals, initramfs, `/dev`	6	Gives a shell and ~300 small applets. BusyBox is GPLv2.
3	**GNU coreutils, bash**	More syscalls, job control, ptys, `/proc` hits, locale basics	6, 9	Static builds first. GPLv3.
4	**tcc, then gcc/binutils/make** (self-hosting)	Heavy `mmap`, large files, writable filesystem, many processes	7, 9	tcc (small) first. Self-hosting means Starman can rebuild programs for itself.
5	**vim**	terminfo, termios raw mode, `select`/`poll`, signals	6	With ncurses or termcap.
6	**Python**	Threads, `futex`, `mmap`, `dlopen`, time, large stdlib, `/proc`	9	Static or dynamic; extension modules need dynamic linking.
7	**OpenSSH** (client, then server)	Sockets, `getrandom`, ptys, `select`, fork, privilege separation	8	Needs a working network and entropy source.
8	**xterm, twm** on X11	`/dev/fb0`, evdev, Unix sockets, shared memory, X server	10	Fonts and X libraries are a large dependency set.
9	**Firefox** (stretch)	Multi-process, many threads, `futex`, `epoll`, `memfd`/shared memory, glibc, GTK stack, software rendering	9, 10	Not committed: the long tail of Linux behaviours is large and the dependency tree is huge. Treat as research.

8. Testing strategy, and effort estimates

8.1 Testing

- **Two environments.** QEMU for automated, headless runs (`-serial file:`, the monitor's `sendkey` and `screendump`, the `isa-debug-exit` device so a test can report pass/fail to the host script, and `-s -S` with GDB for debugging); VMware Workstation for release acceptance on the real target. Both BIOS and UEFI are exercised in QEMU (OVMF), and VMware is run with both firmware types.
- **Serial logs as the source of truth.** Each test boots, runs scripted commands, and compares the serial log to expected output. A CI script returns pass/fail per phase.
- **In-kernel self-tests** (allocators, scheduler, VFS) selectable from the GRUB command line.
- **Conformance suites, in subsets.** The **Linux Test Project (LTP)** syscall tests, chosen call by call as each is implemented. The **Open POSIX Test Suite** for POSIX interfaces (signals, threads, timers, message queues), where large parts will deliberately fail early. **libc-test** from the musl project, which checks libc behaviour against the

kernel. **BusyBox's own tests** and the test-suites of Python and coreutils as the ladder advances. Tests that fail for an interface Starman does not claim to provide are recorded as *expected failures*, not silently dropped.

- **Differential testing.** Run the same binary on real Linux under `strace` and on Starman with syscall logging, and compare the call sequences and results.
- **Regression gate.** No phase is closed with a failing test from an earlier phase.

8.2 Effort estimates

> **Read this first.** These are my planning estimates, not measurements. They assume one experienced developer working full time, kernel code in C with assembly for the core, reuse of existing permissively licensed libraries, and AI-assisted coding with human review. They are ranges because the long tail (hardware quirks, Linux behaviours that real programs depend on) is the main source of uncertainty. I would plan on the upper end of each range. "Lines" means kernel source lines excluding third-party libraries.

Table 1. Estimated effort by phase (estimates)

Phase	Person-weeks	Kernel lines	Main uncertainty
2 Memory management	*3–5*	*3,000–5,000*	Higher-half transition; UEFI memory-map oddities
3 Interrupts, timing, ACPI, PCI	*4–6*	*4,000–7,000*	ACPI on real firmware; timer calibration
4 Processes, scheduler	*4–6*	*3,000–5,000*	Locking and race conditions
5 User mode, syscalls, ELF	*4–6*	*4,000–6,000*	Getting the entry/exit path exactly right
6 VFS, tty/pty, fork/exec, signals, threads	*14–22*	*15,000–25,000*	Signals, job control, `mmap` and `futex` semantics
7 Storage, ext2, ext4 (ro), FAT32	*8–12*	*8,000–14,000*	Cache coherency, filesystem edge cases
8 Networking and sockets	*10–16*	*10,000–25,000*	lwIP versus own stack; Linux socket semantics
9 Dynamic linking, /proc, /sys	*8–12*	*8,000–14,000*	Open-ended list of missing behaviours
10 Graphics and input (X11)	*8–14*	*8,000–15,000*	X server choice; SVGA II documentation gaps
11 Audio and USB	*10–16*	*10,000–18,000*	xHCI complexity
Cross-cutting: integration, ladder bring-up, tests, docs	*12–20*	—	Whatever the real programs break on
Total (through Phase 11)	***85–135***	***75,000–135,000***	About 1.7–2.7 person-years

Through the end of Phase 6 (a BusyBox shell on the console), my estimate is 33–51 person-weeks including a share of the cross-cutting work. Firefox is excluded from the table.

8.3 Reference points

I use these only as general context and I am not quoting figures from them.

- **SerenityOS** is a from-scratch Unix-like system (own kernel, libc and desktop) with a native POSIX-flavoured API. It shows what option A looks like: very capable, but built over years by a large community, with software ported one program at a time.
- **Managarm** is a microkernel-based system that aims to run unmodified Linux-style userland, with a Linux-compatible interface provided by a POSIX subsystem and the `mlibc` C library. It is a close cousin of this proposal and has run X11 and Wayland clients.
- **The FreeBSD Linux compatibility layer and illumos LX zones** translate Linux system calls inside a Unix kernel so Linux binaries run. They show option B is long-lived and practical, but also that the effort never fully ends as Linux grows.
- **gVisor** (Google) reimplements a large part of the Linux system-call interface in user space and publishes which calls are supported. It shows the breadth of work required and that a partial implementation is workable for a defined set of software.

- **Kerla** is a small Rust kernel aiming at Linux binary compatibility, built by a very small team. It is the closest in scale to Starman's and shows that a BusyBox-level result is realistic for one developer, while the late rungs are much harder.

9. Risks and licensing

9.1 Risks

Risk	Likelihood	Effect and mitigation
Scope growth ("one more syscall")	High	The long tail is the real cost. Mitigate with the ladder (each rung defined by a program), the `ENOSYS`-logging policy, and a written non-goals list.
Concurrency bugs (scheduler, locks, later SMP)	High	Intermittent hangs and corruption. Defer SMP; add lock-ordering rules, assertions, and long soak tests.
VMware-specific behaviour not seen in QEMU	Medium	Test in VMware at each phase boundary, not only at the end; keep the target hardware list fixed.
Data loss from writable filesystems	Medium	ext2 write only on disposable disk images; ext4 stays read-only; always test on virtual disks.
Missing documentation (SVGA II, vmxnet3, some VMware behaviours)	Medium	Start with well-documented devices (e1000, AHCI, VGA/VBE framebuffer); treat VMware-specific drivers as optional upgrades.
Firefox-class software	High	Treated as a stretch goal with no commitment and no schedule.
Single-developer and tool risk	Medium	Keep code well commented, keep the build reproducible, and keep releases (ISO plus source zip) at every phase.
Security flaws in a hobby kernel	Certain	Run it in a VM only, never on a production machine or network, until a review is done.
Assembly-only constraint	If chosen	See the cost box in Section 3.3.

9.2 Licensing and legal notes

This section is a technical summary for Andrew's attention, not legal advice, and I would expect him to apply his own judgment on each point.

Component	License (as I understand it)	Consequence for Starman
Starman kernel (own code)	**Not yet chosen**	A license should be chosen before importing third-party code in Phase 2 or 3. A permissive license (MIT or BSD) keeps options open; GPLv2 would be needed if any Linux kernel code or drivers are ever incorporated.
musl libc	MIT	Free to ship in binaries and build chains; keep the copyright notice.
glibc	LGPL 2.1+	Fine for dynamically linked programs run on Starman; if it is redistributed on the ISO, its license terms and a source offer apply.
BusyBox, bash, coreutils, gcc, binutils	GPLv2 or GPLv3	These run as separate programs, so they do not affect the kernel's license, but an ISO that *contains* them is a redistribution with source-offer obligations. Keep an inventory and ship sources or a written offer.
lwIP	BSD (3-clause)	Compatible with a proprietary or permissive kernel.
ACPICA / uACPI	Permissive or dual-licensed (check the exact terms of the version chosen)	Usable; verify before importing.

Component	License (as I understand it)	Consequence for Starman
Linux kernel drivers and code	GPLv2	**Do not copy.** Doing so would require the combined kernel to be GPLv2. This is the main reason for clean-room ABI work and for using BSD-licensed or documentation-based drivers.
FreeBSD or other BSD-licensed drivers	BSD, generally (per-file headers vary)	Acceptable sources for e1000 or AHCI ideas; check each file header.
Python, OpenSSH, xterm/twm, Vim, Firefox	PSF, BSD, MIT-style, Vim license, MPL 2.0	Run as separate programs; redistribution rules differ and should be listed in the release inventory.

- **Reimplementing an interface.** The Linux system-call interface is reimplemented, not copied. Linux's own `COPYING` notes that normal system calls by user programs do not make them derivative works of the kernel, and the Supreme Court's 2021 decision in *Google v. Oracle* treated reuse of API declarations as fair use on its facts. Andrew will know better than I how far to rely on that; the practical safeguard is the clean-room rule in Section 6.
- **Names and marks.** UNIX® is a registered trademark of The Open Group, and POSIX® is a registered trademark of the IEEE. Starman should be described as "Linux-ABI compatible" or "POSIX-style" unless it is formally certified. "Linux" is also a mark, to be used descriptively.
- **AI-assisted code.** If any of the code is machine-assisted, keep notes of provenance and review, particularly for any block that resembles known open source code.

10. VMware virtual hardware to target first

VMware Workstation presents a conventional PC. The table lists what to target first. Device IDs and options differ between Workstation versions, so the plan is to confirm each item by booting a Linux live guest in the same VM and running `lspci -nn` before writing a driver.

Function	First target	Later / optional	Phase
Firmware	BIOS and UEFI (GRUB handles both)	Secure Boot (needs a signed loader; not planned)	1
CPU, timers, interrupts	x86-64, local APIC, IOAPIC, HPET/PIT/TSC, ACPI	Multiple vCPUs (SMP)	3, 4
Disk	**SATA (AHCI)** virtual disk	NVMe; LSI Logic or PVSCSI (a VMware paravirtual SCSI controller)	7
Network	**Intel e1000** (Workstation's default for many guest types; e1000e also available)	vmxnet3 (VMware paravirtual NIC, faster, less documented)	8
Display	Framebuffer from GRUB (UEFI GOP or VBE), already working in 0.1	**VMware SVGA II** driver (mode setting, absolute pointer)	10
Keyboard, mouse	**PS/2** (present and working today)	USB HID; VMware absolute-pointer protocol for a seamless mouse	10, 11
Serial	COM1 to file (already used by 0.1 for logs)	Named pipe for live capture	1
Sound	None at first	Intel HD Audio, or the Ensoniq-style card VMware can present	11
USB	None at first	xHCI/EHCI controller, USB mass storage	11

The same list is available in QEMU (`-device e1000`, `-device ich9-ahci`, `-vga std`), so QEMU can be used for fast, scripted testing before each VMware check.

11. Proposed immediate next step: Phase 2 in detail

Objective: give Starman a real memory subsystem, and in doing so move to the build structure the rest of the project needs. Estimated effort: *3–5 person-weeks*.

11.1 Work packages

WP	Title	Content	Estimate
2.0	Build migration	Keep the NASM boot stub. Add freestanding C (`-ffreestanding -mno-red-zone -mcmodel=kernel`), a linker script and an ELF64 kernel image loaded by GRUB's Multiboot2 support; move the kernel to the higher half (`0xFFFFFFFF80000000`); keep the current features working and the Makefile building on RHEL 9 (`gcc`, `binutils`, `nasm`).	*0.5–1 pw*
2.1	Physical memory manager	Bitmap (or buddy) allocator built from the Multiboot2 memory map. Reserve the kernel image, boot info, low memory, firmware and framebuffer regions. API: `pmm_alloc`, `pmm_free`, contiguous allocation, statistics. A `pmm` shell command prints total, used and free pages.	*0.5 pw*
2.2	Virtual memory manager	4-level page-table code: `map`, `unmap`, `protect`, `invlpg`; a direct map of all RAM with 2 MiB or 1 GiB pages; NX enabled, kernel text read-only and executable, data non-executable; guard pages below stacks; map the framebuffer and device memory explicitly (the identity map goes away); page-fault handler with CR2 diagnostics.	*0.5–1.5 pw*
2.3	Address spaces	`as_create`, `as_clone` (copy-on-write groundwork), `as_destroy`: a new top-level page table that shares the kernel half. Not yet used by processes, but unit-tested.	*0.5 pw*
2.4	Kernel heap	`kmalloc`, `kcalloc`, `krealloc`, `kfree`: size-class slabs over page allocations, large allocations mapped directly, optional debug mode with red zones, poison patterns and leak counters.	*0.5–1 pw*
2.5	Tests and tooling	Parse the GRUB command line (`selftest`, `loglevel`); in-kernel self-tests; shell commands `pmm`, `vmm`, `heap`, `memtest`; a host-side script that boots QEMU, runs the tests and checks the serial log; updated README and release ISO.	*0.5 pw*

11.2 Acceptance tests

1. Boots on QEMU BIOS and UEFI and in VMware with 128 MiB, 512 MiB and 8 GiB of RAM, reporting the same usable RAM as the boot map to within the memory the kernel itself reserves.
2. Allocate and free 100,000 physical pages in random order: no double allocation, and the free count returns to its starting value.
3. Create, map, touch and destroy 1,000 address spaces: no leaked pages, and the kernel mappings stay identical in each.
4. Write to a read-only page and execute from a non-executable page: each produces a clean, diagnosable page-fault report (address, error code, RIP) and does not triple-fault.
5. Heap soak: 1,000,000 random-size allocations and frees with pattern verification and zero leaks reported in debug mode.
6. All Starman 0.1 features (shell commands, banner, keyboard, serial, `reboot`, `halt`) still work after the migration.

11.3 Decisions needed from Andrew before starting

1. **Language policy:** assembly core plus freestanding C (recommended), or pure assembly with the reduced ladder in Section 3.3.
2. **License** for the Starman source (Section 9.2).
3. **Priority:** is the first goal a BusyBox shell (Phases 2–6), or something else (for example, audio for the SSL 18 work, which would move Phase 11 forward)?

4. **Hardware baseline:** confirm VMware Workstation on RHEL 9 as the primary target, and whether real hardware is expected.
5. **Review points:** whether he wants a short review and a VMware test after every phase (recommended), with a release ISO and source zip each time.

If Andrew approves, the first deliverable would be Starman 0.2: the same shell, with the new memory subsystem and the commands to inspect it.

About this document. Prepared on Nov 26, 2026 as a v0.1 draft. All effort and size figures are estimates. Descriptions of other projects are general, and license and legal remarks are summaries for review, not legal advice. Package names and virtual-hardware identifiers should be verified on the actual target systems before use.