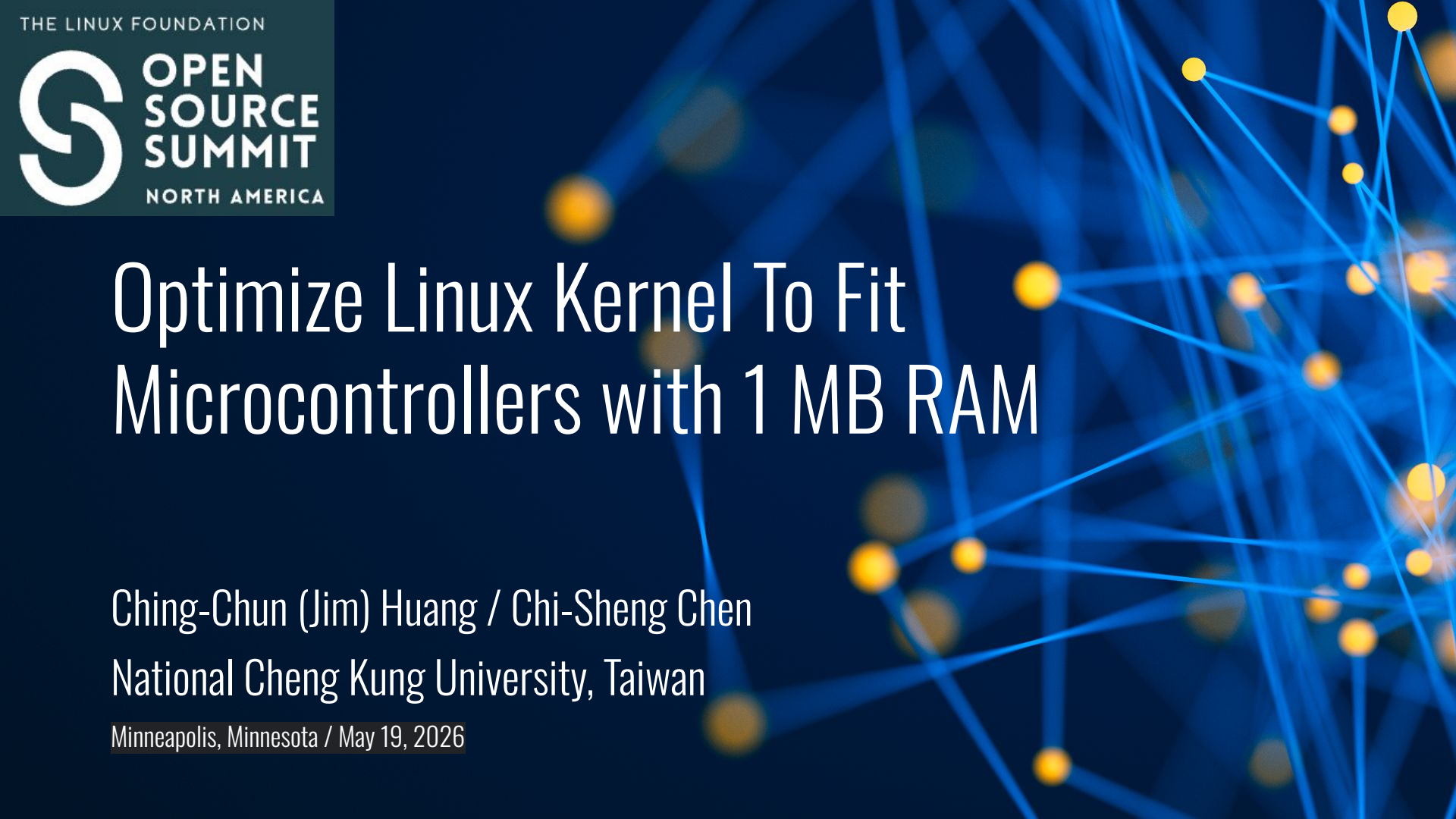
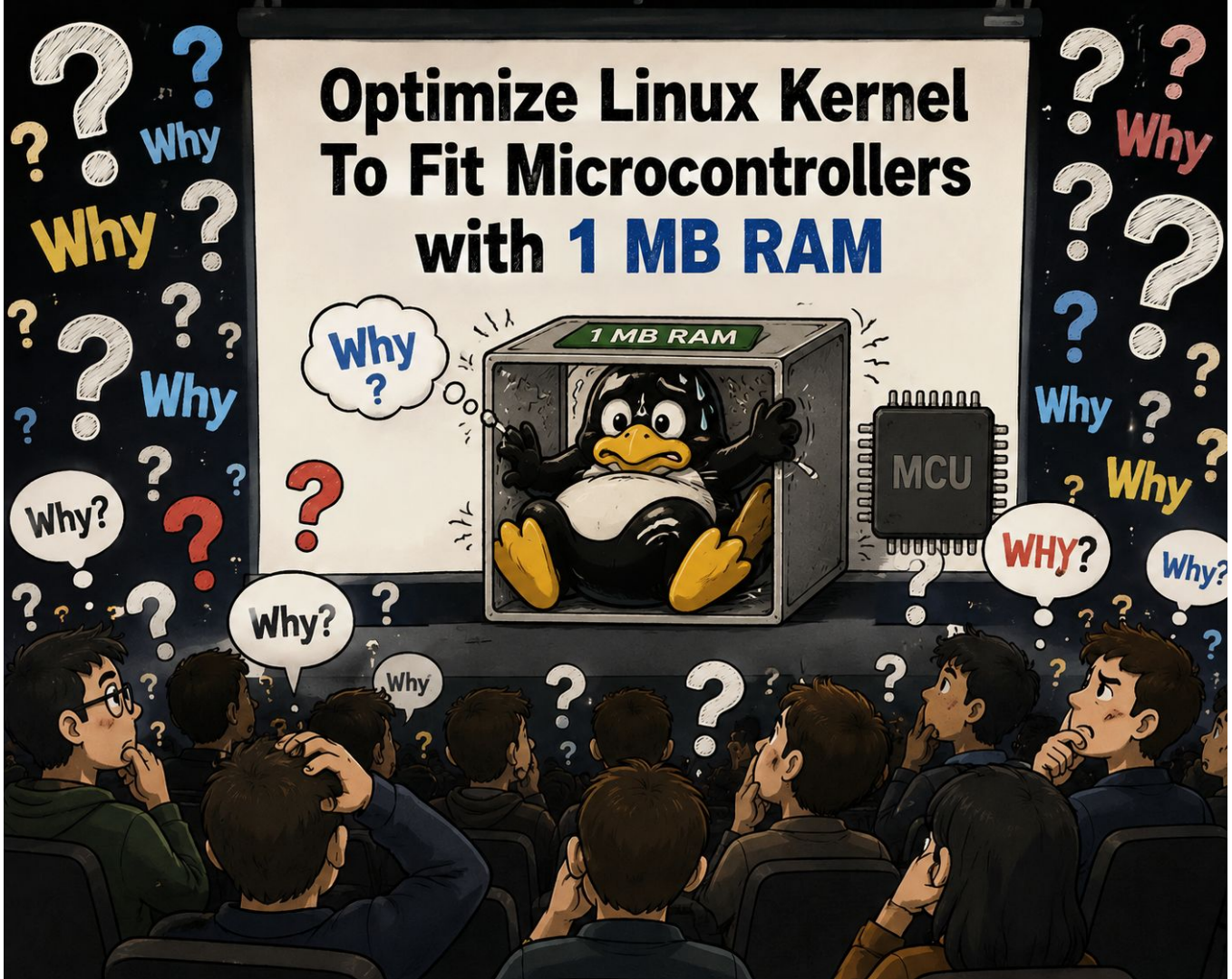


Optimize Linux Kernel To Fit Microcontrollers with 1 MB RAM

Ching-Chun (Jim) Huang / Chi-Sheng Chen

National Cheng Kung University, Taiwan

Minneapolis, Minnesota / May 19, 2026


Optimize Linux Kernel
To Fit Microcontrollers
with 1 MB RAM
Why
Why
Why
Why
Why?
Why?
Why
Why
Why
Why?
WHY?
Why?
1 MB RAM
MCU

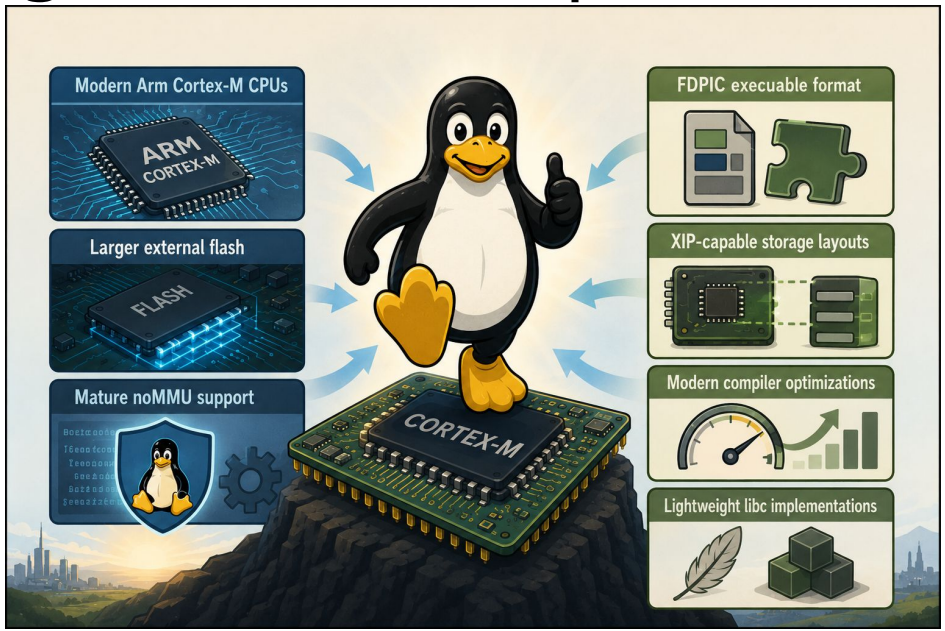
Part I: Why Linux on Microcontrollers?

Traditional Concerns About Linux on MCUs

- Determinism and latency
  - hard real-time scheduling
  - interrupt latency
  - bounded execution behavior
- Memory footprint
  - kernel residency
  - userspace residency
  - runtime duplication
- Hardware enablement
  - BSP maintenance
  - peripheral drivers
  - vendor fragmentation
- Deployment model
  - firmware update complexity
  - monolithic application images
  - limited software reuse
- Legal and ecosystem concerns
  - GPL compliance
  - long-term maintenance
  - fragmented middleware stacks

Linux-noMMU has become significantly more practical

- modern Arm Cortex-M CPUs
- larger external flash
- mature noMMU support
- FDPIC executable format
- XIP-capable storage layouts
- modern compiler optimizations
- lightweight libc implementations



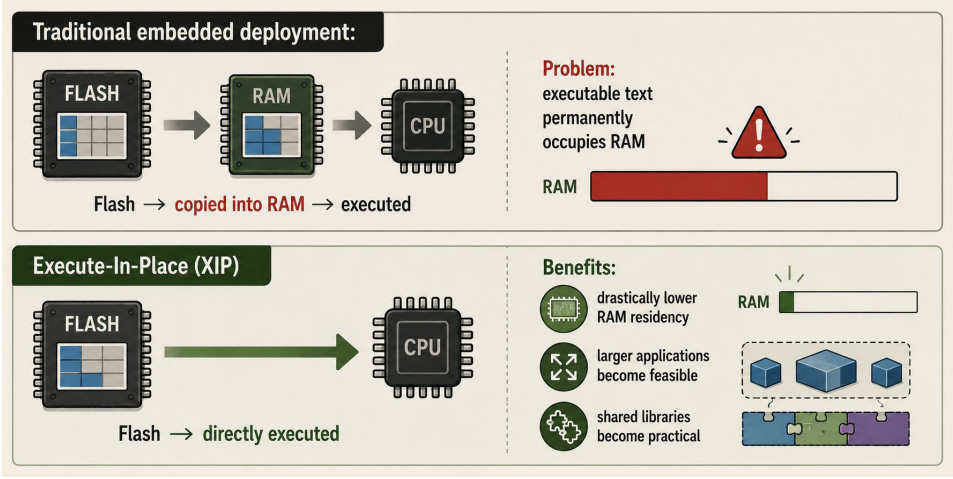
⇒ Running Linux on MCUs is no longer merely experimental

Linux Enables Software Features Rarely Seen on MCUs

Capability	Traditional RTOS	Linux noMMU
POSIX userspace	Partial	Native
Dynamic linking	Rare	FDPIC
Shared libraries	Rare	Native
Application XIP	Difficult	Supported
Reusable software stack	Limited	Strong
Multi-process execution	Weak	Native
Standard toolchain	Partial	Full ELF ecosystem
Portable userspace software	Limited	Direct

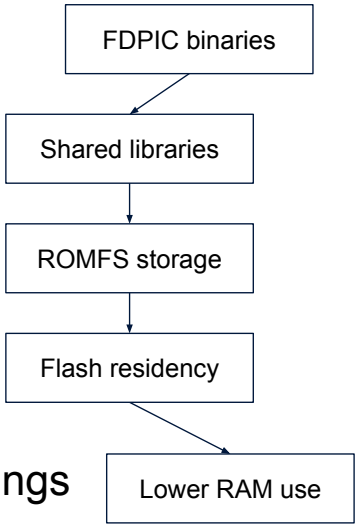
Why XIP Matters

- Traditional embedded deployment:
  - Flash → copied into RAM → executed
- Problem: executable text permanently occupies RAM
- Execute-In-Place (XIP)
  - Flash → directly executed
- Benefits:
  - drastically lower RAM residency
  - larger applications become feasible
  - shared libraries become practical



XIP Is More Than "*Execute From Flash*"

- XIP alone only saves executable text copies.
  - The real gains appear when the entire software stack is designed around flash residency.
- Traditional flat binaries:
  - require contiguous RAM allocation
  - duplicate executable text
  - difficult to share code
- FDPIC:
  - supports direct execution from flash
  - supports relocation without MMU
  - enables dynamic linking enabled shared executable mappings
- ⇒ The same executable library pages are shared across processes directly from flash.



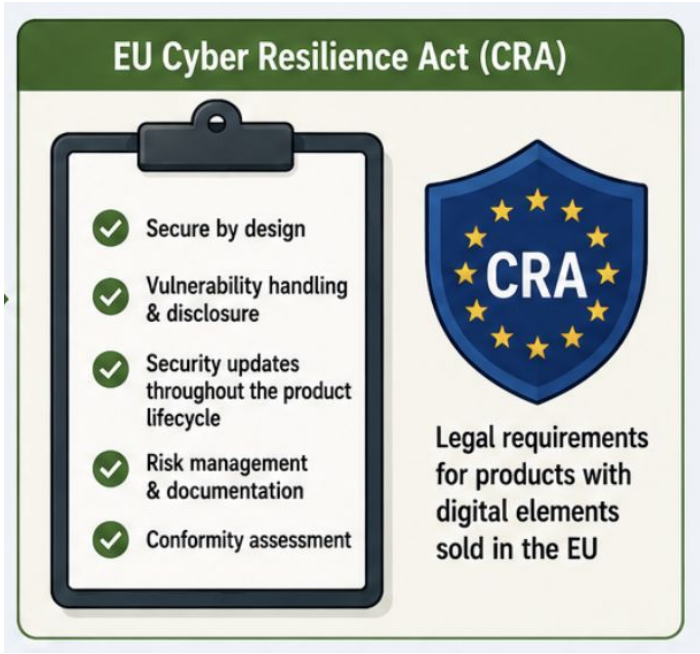
XIP changes flash from passive storage into active execution memory

- Kernel-side
  - finer-grained .data reduction
  - read-only structure migration
  - constification
  - compressed writable sections
  - lazy subsystem initialization
- Storage-side
  - finer-grained .data reduction
  - read-only structure migration
  - constification
  - compressed writable sections
  - lazy subsystem initialization
- Userspace-side
  - split writable hot paths
  - reduce relocation count
  - smaller dynamic loader
  - library specialization
  - function-section garbage collection

EU Cyber Resilience Act (CRA)

- Rethink Traditional RTOS Deployment Model
  - single monolithic firmware image = RTOS + drivers + middleware + crypto + application
- Characteristics:
  - statically linked
  - tightly coupled
  - vendor-specific
  - difficult to audit independently
- ⇒ Why reusable userspace architectures matter for long-term security maintenance

  ⇒ Security fixes require rebuilding and redistributing the entire firmware image (CRA nightmare)



EU CRA Changes the Engineering Requirements

- CRA increasingly requires:
  - long-term vulnerability management
  - traceable software components
  - timely security updates
  - dependency visibility
  - maintainable software lifecycle

Dynamic linking converts vulnerability remediation from “firmware replacement” into “component maintenance”.

- Implication:
  - Embedded software architectures now directly affect regulatory compliance cost.

Static RTOS deployment

App A + libc + TLS
App B + libc + TLS
App C + libc + TLS

If OpenSSL / mbedTLS vulnerability appears:

- every firmware image must be rebuilt
- every image must be revalidated
- every product variant may diverge

Linux/noMMU deployment

Applications
↓
shared libc.so/crypto/middleware

Security update:

- update shared library once
- all applications benefit immediately

Shared libraries create natural security boundaries

› Flash-Resident Shared Libraries Improve OTA Economics

Linux ecosystem already provides:

- CVE tracking
- upstream security processes
- reproducible toolchains
- long-term maintenance workflows
- standardized SBOM tooling

Compared to many RTOS ecosystems:

- fewer vendor-specific forks
- better patch visibility
- broader community auditing

Traditional firmware update

replace entire firmware image

Problems:

- large update payloads
- high validation cost
- rollback complexity

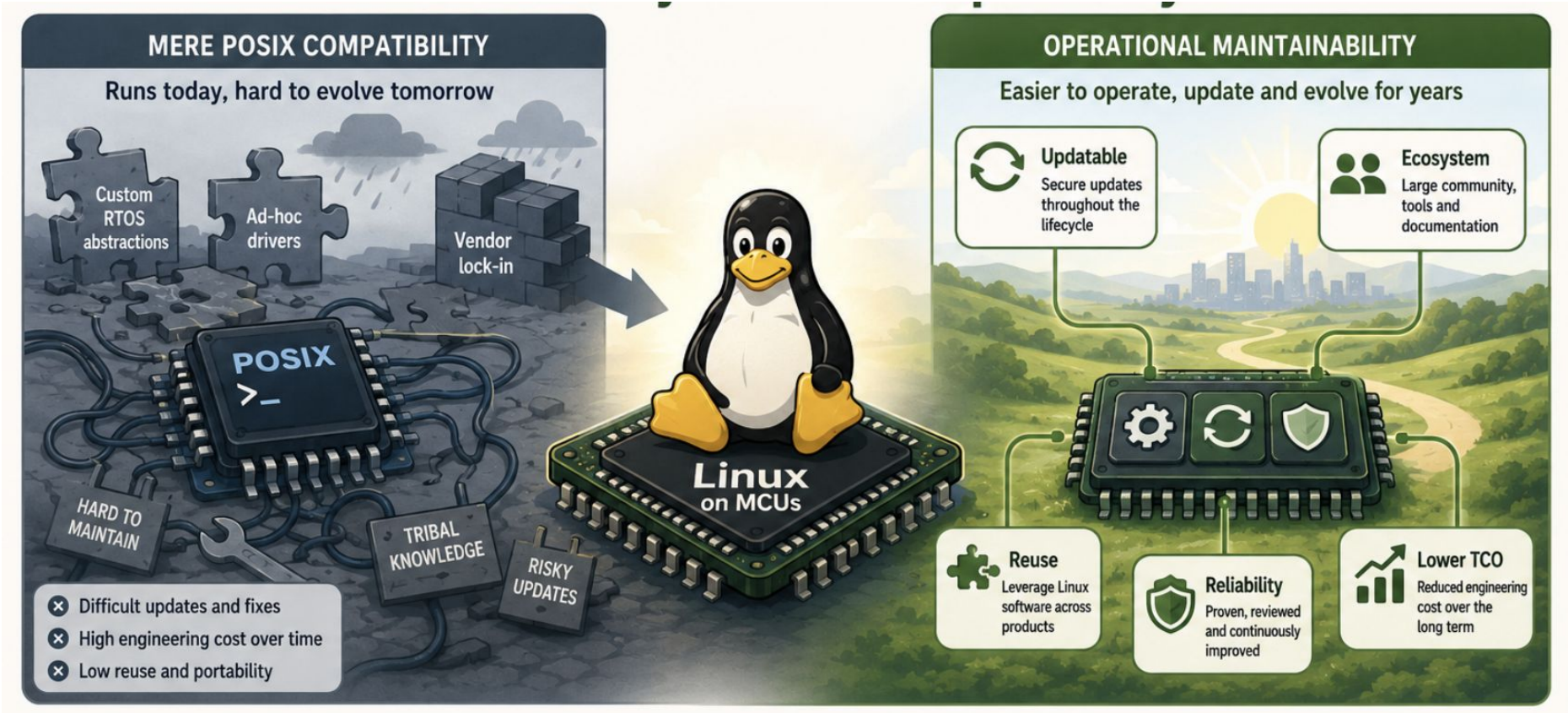
Linux/noMMU with XIP libraries

replace only affected shared object

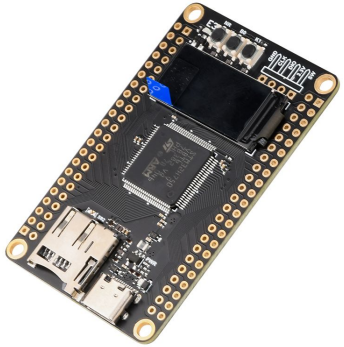
Benefits:

- smaller OTA updates
- reduced flash wear
- faster deployment
- simpler rollback strategy

The long-term value of Linux on MCUs may ultimately be operational maintainability, not merely POSIX compatibility



Part II: Components



Selected Hardwares

Two boards are used for experiments:

- STM32H750 : Minimal showcase for 7.x kernel with 1 MiB RAM
- STM32F429-Discovery : Feature-rich Linux

STM32H750: Minimal showcase
Linux v7.x with 1MiB RAM

Specifications:

- Arm Cortex-M7
- 128 KiB flash + 8 MiB external flash
- 1 MiB SRAM

Action Items:

- Run Linux v7.0
- XIP (kernel + application)
- 196 KiB free RAM after boot
- User program with libc support
- Run dynamic linked busybox

STM32F429-Discovery: Feature-Rich Linux System

Full graphics environment with UI applications

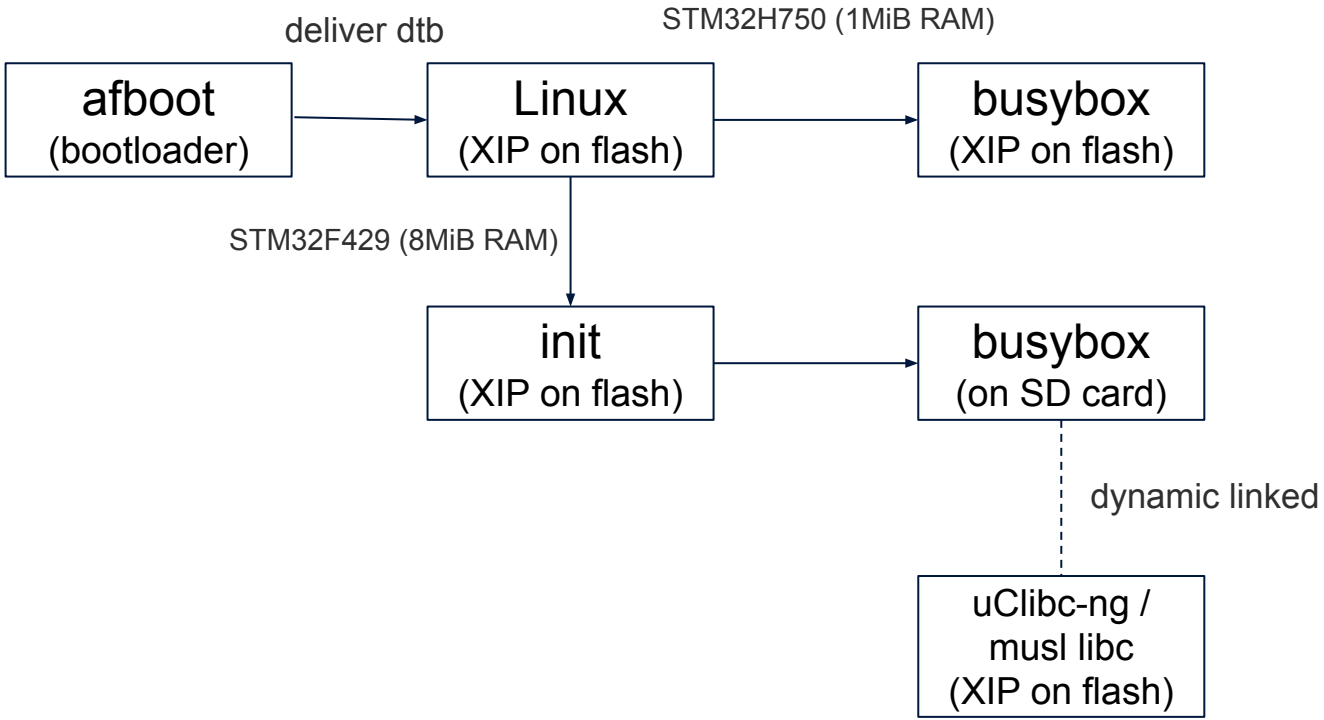
Specifications:

- Arm Cortex-M4
- 2 MiB flash
- 128 KiB SRAM + 8 MiB SDRAM
- Touchscreen
- QVGA LCD Display

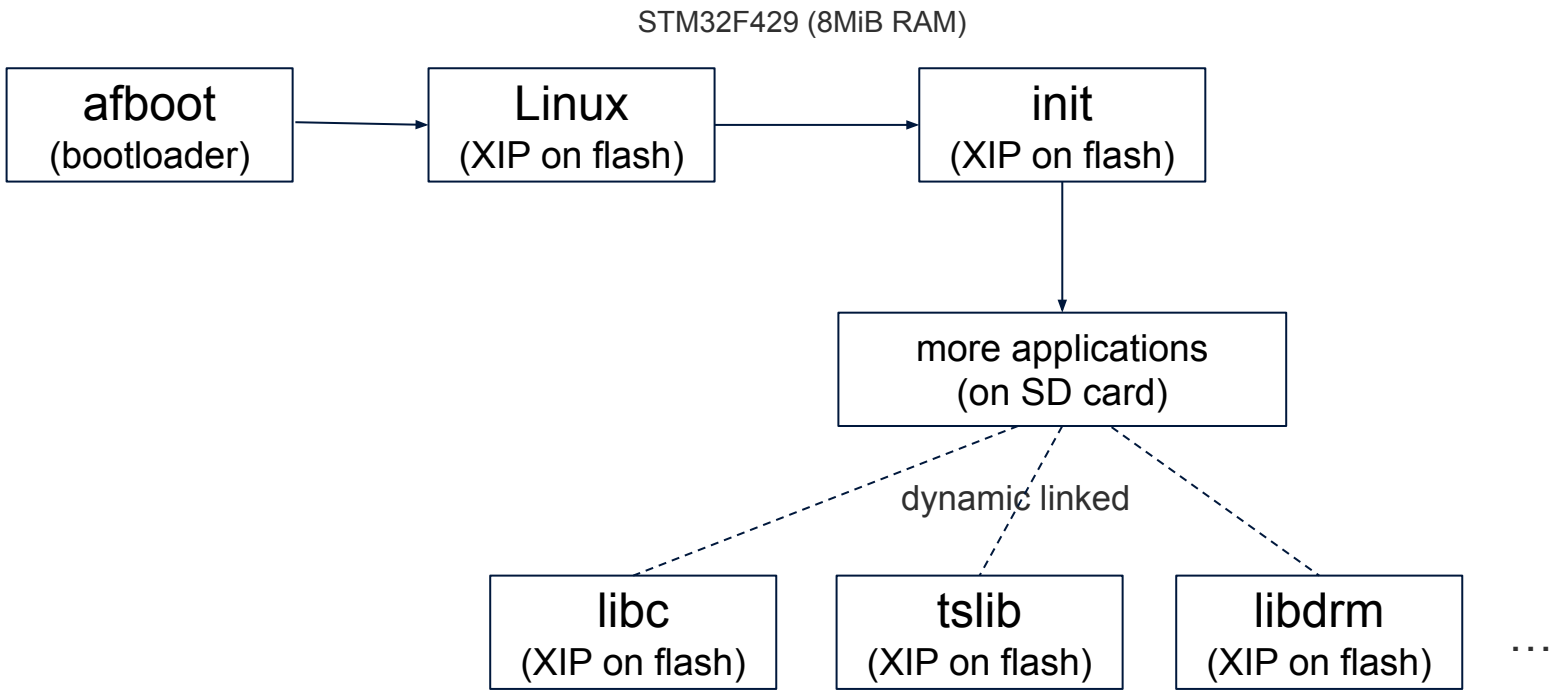
Action Items

- XIP for both kernel and applications
- 6.2 MiB free RAM after boot
- Support SD card, touch input events, and hardware-assisted 2D display
- Integrate LVGL and related libraries
- Demonstrate a cellphone prototype as a showcase

Boot sequence



Boot sequence (w/ more applications)



Composition of a minimal Linux

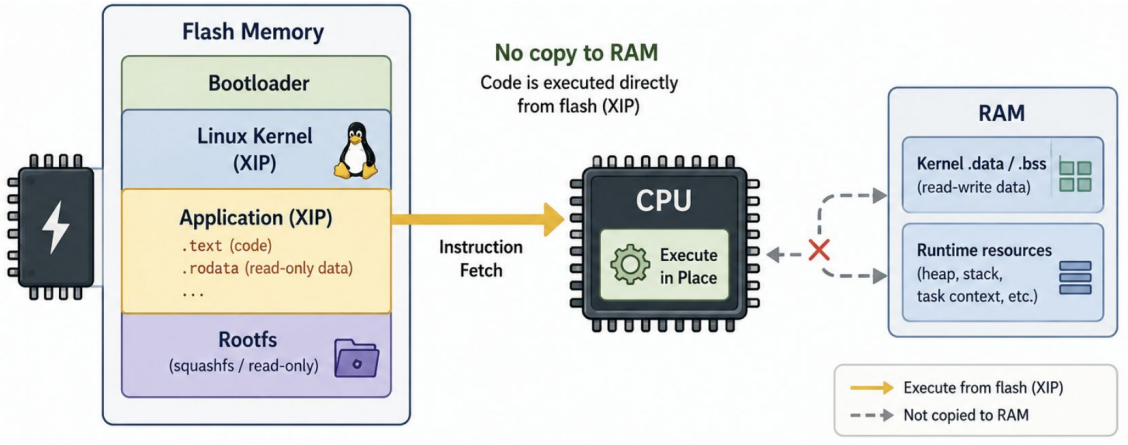
- Bootloader + Device Tree Blob (DTB): 20 KiB
- XIP kernel stored on flash (only `.data` is compressed): 1 MiB
- Root filesystem (`rootfs`)
  - static linking, or
  - dynamic linking with shared libraries: 300 KiB
- POSIX threads disabled in libc

All components reside on flash memory.

Built with GCC 15.2 using -Os optimization

rootfs

- ROMFS is used as the root filesystem
  - No RAM copy during boot
  - Preserves flash-native execution
- Avoids RAM duplication compared to initramfs
- Enables direct application XIP from flash memory



Executable Formats on Linux-noMMU

- FLAT Binary
  - Execute-In-Place (XIP) capable
  - Statically linked userspace
- FDPIC
  - Execute-In-Place (XIP) capable
  - Supports dynamic linking
  - Enables shared libraries on noMMU systems

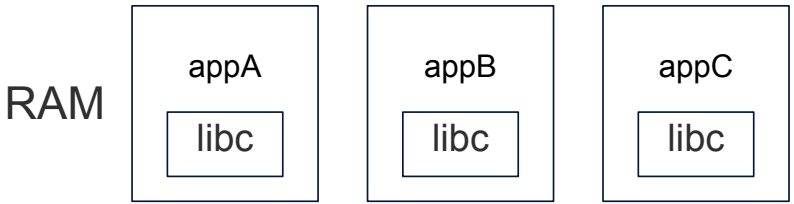
In more feature-rich deployments, FDPIC enables a Linux-style shared library architecture on no-MMU systems, where dynamically linked libraries can be executed directly from flash and shared across multiple processes. This greatly improves memory efficiency and software reusability.

Static Linking vs. Dynamic Linking + XIP

In feature-rich systems, applications may become too large to fit entirely in flash memory.
Assume appA, appB, and appC all require libc:

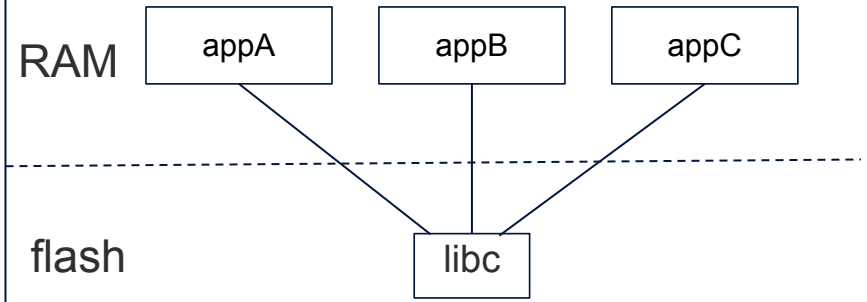
Static Linking

- Each application contains its own copy of libc
- Executable code must be loaded into RAM
- Results in duplicated flash and RAM usage



Dynamic Linking + XIP

- Applications share a common libc.so
- Shared libraries remain executable directly from flash via XIP
- Only writable sections need RAM residency
- Significantly reduces overall memory usage



Evaluated libc Implementations

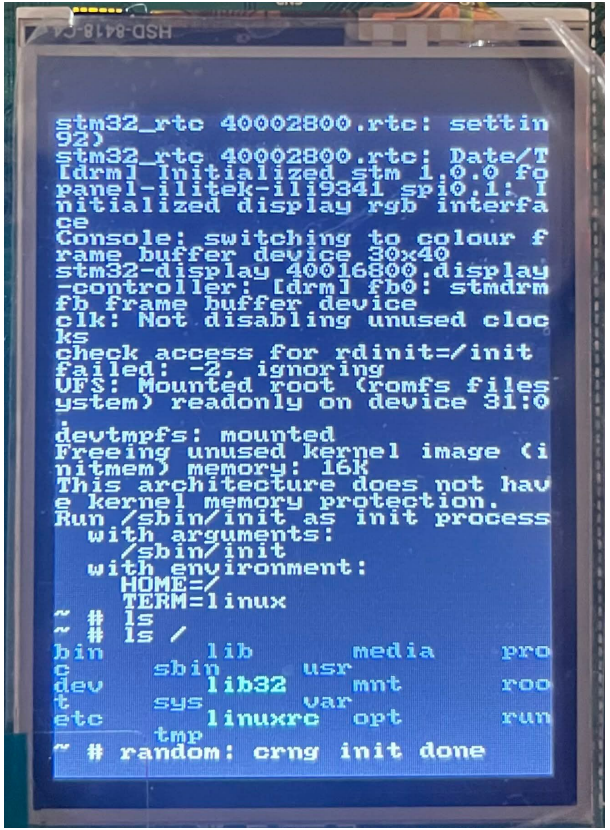
- uClibc-ng
  - Mature support for embedded Linux
  - Supports FDPIC-based dynamic linking
  - Well suited for XIP-oriented deployments
- musl
  - smaller and cleaner runtime design, widely used in containers
  - Lower memory footprint
  - FDPIC dynamic linker support currently requires out-of-tree patches

Requirements for cellphone-prototype

- SD/MMC support for expanded userspace storage
- Utilization of 8 MiB RAM storage, including ~1 MiB reserved for userspace applications
- XIP-capable dynamically linked shared libraries
- LCD display and touchscreen support
- LVGL-based graphics stack
- POSIX thread support (pthread)
- Linux userspace middleware integration:
  - libdrm
  - libevdev
  - tslib

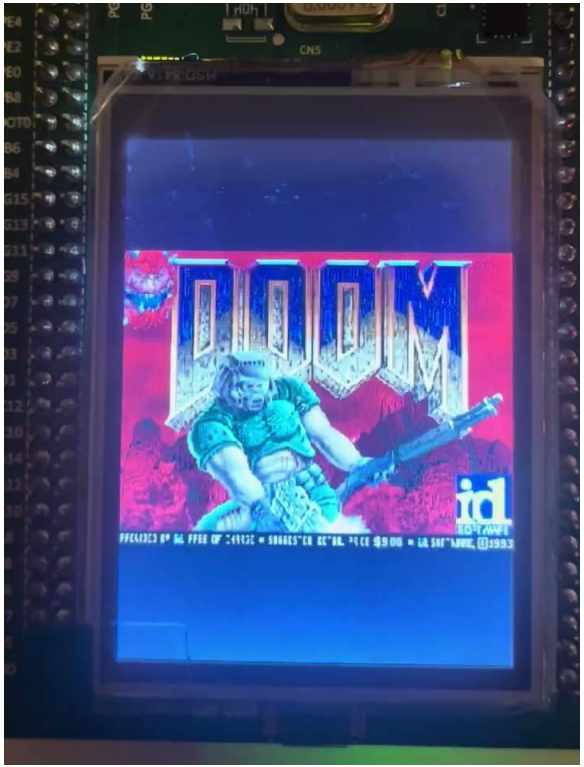
Linux Graphics Stack on MCU

- QVGA display (240×320)
- DRM-based rendering architecture
- libdrm userspace integration
- fbdev and fbcon evaluated for compatibility and comparison



Using DOOM to Demonstrate fbdev Functionality

Using DOOM to Demonstrate DRM Support



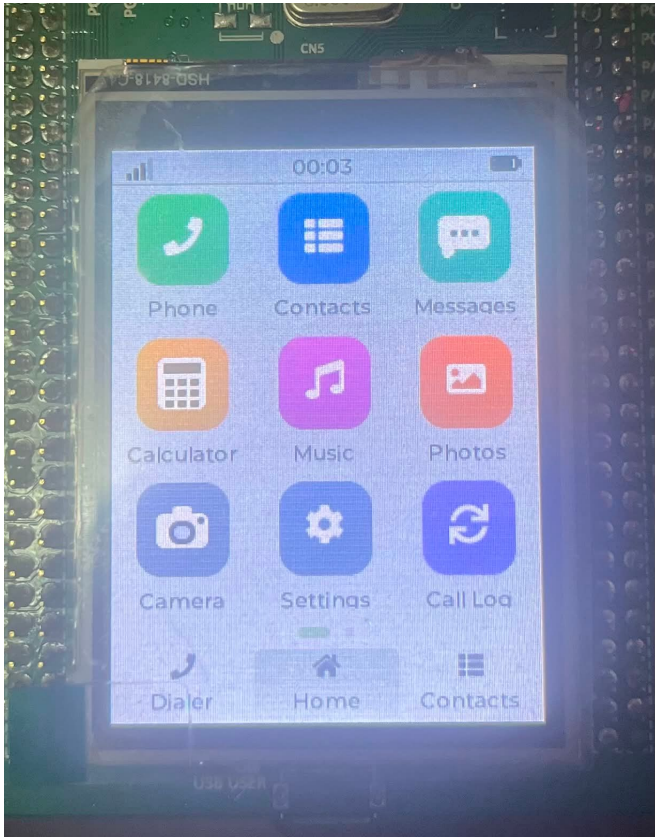
https://www.youtube.com/shorts/6LviKy-gtEM

Touchscreen support

- Based on the Linux input subsystem
- libevdev used for input event processing
- tslib integrated for touchscreen abstraction and calibration
- Supports calibrated touch input for GUI applications

LVGL + DMA2D Accelerations

- LVGL successfully ported through standard cross-compilation
- DMA2D used as graphics accelerator for rendering offload. Hardware-assisted ops include:
  - framebuffer fill
  - image blitting
  - pixel format conversion
- On bare-metal STM32 platforms, LVGL typically accesses DMA2D registers directly through MMIO.
  - Linux-noMMU preserves this programming model, enabling the same low-level DMA2D control path to be reused with little or no modification.
  - This demonstrates that many bare-metal graphics optimizations remain applicable on Linux-noMMU systems.



Part III: Methodologies

Linux on MCUs requires different optimization goals

- Traditional Linux optimizations
  - maximize throughput
  - optimize CPU utilization
  - improve scalability
- MCU Linux optimizations
  - minimize writable memory
  - preserve flash residency
  - reduce runtime duplication
  - simplify update and maintenance paths

⇒ RAM becomes the primary constrained resource

Shrink Linux into MCU-scale systems

- Core Strategy
  - Start from tinyconfig
  - Keep executable code on flash (XIP)
  - Reduce writable memory residency
  - Move from monolithic firmware to reusable components
  - Measure every subsystem quantitatively
- Flash is abundant. RAM is not.
  - Traditional embedded Linux:
    RAM stores both code and writable state
  - Linux-noMMU + XIP:
    Flash stores executable text
    RAM stores only mutable state
  - ⇒ Optimize for RAM residency, not only image size

Layer	Goal
Kernel	Reduce `.data`, `.bss`, slab usage
Userspace	Shared libraries + FDPIC + XIP
Storage/Layout	ROMFS + flash-native execution

Measurement-Driven Tuning

- Analyze System.map
- Treat flash as execution memory, not passive storage
  - Traditional deployment: Flash → copied to RAM → executed
  - XIP-oriented deployment: Flash → directly executed / RAM → mutable state only
- Inspect slab allocation sources
- Compare .text vs .data + .bss
- Measure free RAM after boot

Metric	Purpose
.text	flash occupancy
.data + .bss	mandatory RAM residency
slab usage	runtime kernel memory
free RAM after boot	system viability
/proc/self/maps	verify XIP mappings

XIP affects the entire software stack

- Not only kernel execution:
  - kernel XIP
  - application XIP
  - shared-library XIP
  - ROMFS flash residency
- Design implications:
  - fewer RAM copies
  - lower fragmentation pressure
  - reduced continuous allocation requirements

Verify XIP

Kernel-side:

- inspect memory mappings
- verify executable pages remain on flash

Userspace-side:

- inspect /proc/self/maps
- confirm shared libraries mapped as r-xp
- verify writable sections isolated into RAM

Part IV: Tuning
full source available

https://github.com/rota1001/stm32h7-linux
https://github.com/rota1001/stm32f429-linux

First, Let's talk about a minimal Linux on STM32H750

SPARSEMEM (STM32H750)

- contains 1 MiB SRAM, but the memory is fragmented into multiple non-contiguous regions:
  - 0x20000000 - 0x20020000 (128 KiB)
  - 0x24000000 - 0x24080000 (512 KiB)
  - 0x30000000 - 0x30048000 (288 KiB)
  - 0x38000000 - 0x38010000 (64 KiB)
- Linux therefore cannot treat the SRAM as a single contiguous memory range.
- Approach:
  - enable SPARSEMEM_EXTREME
  - configure SPARSEMEM_MANUAL
  - reduce SECTION_SIZE_BITS

Result:

- Linux can allocate memory across fragmented SRAM regions
- sparsemem metadata overhead remains small

tinyconfig (STM32H750)

- `stm32_defconfig` is too large (.data + .bss ≈ 400 KiB)
- instead start from tinyconfig and re-enable only required components.

```
ARCH_STM32 y
DRAM_BASE 0x20000000
DRAM_SIZE 0x20000
SERIAL_STM32 y
SERIAL_STM32_CONSOLE y
SERIAL_NONSTANDARD y
SPARSEMEM_MANUAL y
XIP_KERNEL y
XIP_PHYS_ADDR 0x90000000
PRINTK y DEBUG_LL y
EARLY_PRINTK y
ARCH_FORCE_MAX_ORDER 8
LOG_BUF_SHIFT 12
```

`LOG_BUF_SHIFT` must be configured after enabling PRINTK, since the default printk log buffer is relatively large (64 KiB).

`stm32_defconfig` vs. `tinyconfig` (STM32H750)

	.text	.data + .bss
stm32_defconfig	2753 KiB	397 KiB
tinyconfig + config for STM32	1081 KiB (-1627 KiB)	152 KiB (-245 KiB)

- .data + .bss reduced to 152 KiB (-62%)
  - This is significantly smaller, but still exceeds the size of a single 128 KiB SRAM bank.
- Challenge: STM32H750 SRAM is physically fragmented:
  - 128 KiB / 512 KiB / 288 KiB / 64 KiB
- Can Linux-noMMU fit its writable kernel state within these fragmented regions?

Tuning more (STM32H750)

Remove the reserved memory region (32KiB) for page table and kexec so that we can use the entire 128 KiB

```
# arch/arm/Makefile
-textofs-y   := 0x00008000
+textofs-y  := 0x0
```

Disable configs that other STM32s need:

```
MACH_STM32F429 n
MACH_STM32F469 n
MACH_STM32F746 n
MACH_STM32F769 n
COMMON_CLK_STM32MP n
```

Tuning more (STM32H750)

Disable some unused linux config:

```
BASE_SMALL y
GPIO_CDEV n
LDISC_AUTOLOAD n
LEGACY_PTYS n
LEGACY_TIOCSTI n
UNIX98_PTYS n
VT n
```

I take a look at the System.map, sorted by size, and remove reduntant UART ports:

```
-#define STM32_MAX_PORTS 9
+#define STM32_MAX_PORTS 1
```

Numbers of tuning (STM32H750)

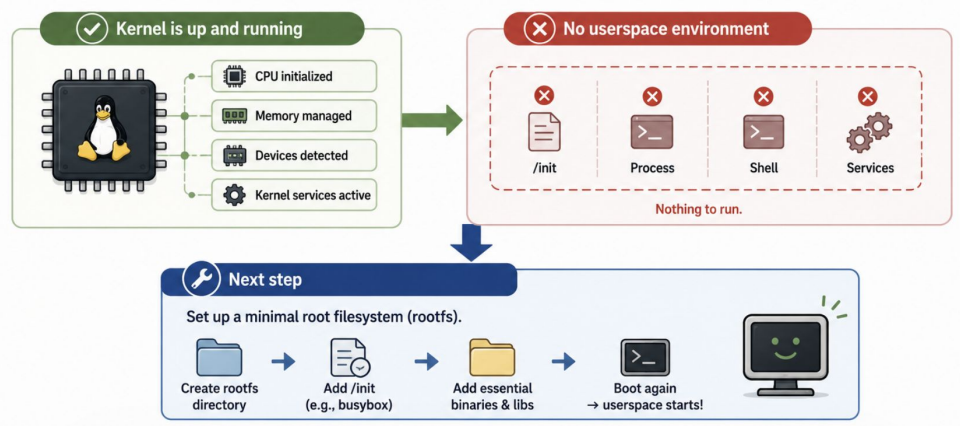
	.text reduction	.data + .bss reduction
STM32 related config	144 KiB	28 KiB
unused Linux config	67 KiB	16 KiB
UART ports	few bytes	3 KiB

Total RAM reduction: -47KiB (-31%)

Finally, we have .data + .bss = 105KiB

The kernel finally boots, but immediately panics

- **`Kernel panic - not syncing: No working init found.`**
  - This means the kernel is now operational, but no userspace environment exists yet.
- Next step:
  - set up a minimal root filesystem (rootfs).



romfs (STM32H750)

The target size of rootfs is 300 KiB, and romfs can enable the Application XIP

Enable configs:

MTD y
MTD_PHRAM y
ROMFS_FS y

Specify a range in flash to place the romfs

"phram.phram=romfs,0x90600000,0x200000 root=mtd:romfs rootfstype=romfs init=/init"

Successfully mount romfs:

VFS: Mounted root (romfs filesystem) readonly on device 31:0.

Minimal validation for applications (FLAT binary format)

Now, let's implement a free standing (no std) user program to ensure the flat binary format work

Run /init as init process

Hello from userspace! (with write)

However, we want libc.

Static Linked uClibc-ng with flat binary format

We use uClibc-ng, and also, implement a helloworld with C Library function.

This is what happen:

nommu: Allocation of length 102400 from process 1 (init) failed

...

binfmt_flat: Unable to allocate RAM for process text/data, errno -12

The .text requires 100KiB of continuous free RAM, which we don't have.

```
text        ...  filename
98460   ...  busybox_unstripped.gdb
```

So we need XIP, and flat binary can be XIP

Run /init as init process

Hello from userspace! (with printf)

Dynamic Linked uClibc-ng with FDPIC format

Also, we can use FDPIC executables so that we can enable dynamic linked

No special hacking needed, it is XIP by default

Tuning uClibc-ng

For flat binary format, tuning the config of uClibc-ng has almost no effect to the program size. This is because unused parts will not be compiled into the program.

However, for dynamic linked library, we will not know if a function useful until the program is loaded.

- Tuning uClibc-ng according to the use senario can reduce the library size

Tuning uClibc (For FDPIC)

The following config can be set:

LDSO_LDD_SUPPORT n
LDSO_PRELOAD_ENV_SUPPORT n
LDSO_SEARCH_INTERP_PATH n
LDSO_LD_LIBRARY_PATH n
UCLIBC_HAS_SYSLOG n
MALLOC_SIMPLE y
UCLIBC_DYNAMIC_ATEXIT n
UCLIBC_HAS_SHADOW n

UCLIBC_HAS___PROGNAME n
UCLIBC_HAS_GNU_ERROR n
UCLIBC_BSD_SPECIFIC n
UCLIBC_HAS_BSD_ERR n
UCLIBC_HAS_EPOLL n
UCLIBC_HAS_XATTR n
UCLIBC_HAS_PROFILING n
UCLIBC_HAS_CRYPT_IMPL n

The network utilities are almost disabled except for the UCLIBC_HAS_SOCKET, which is needed by the busybox (included in the libbb.h)

Tuning uClibc (For FDPIC)

Here are the numbers:

	.text	.data + .bss
libuClibc before tuning	368 KiB	23 KiB
libuClibc after tuning	290 KiB (-78 KiB)	22 KiB (-1 KiB)

static linked hello vs dynamic linked hello

Here are the numbers of how many space we need on flash to run a hello program:

	Image size (.text + .data)	.data + .bss
static linked (flat binary)	48 KiB	28 KiB
dynamic linked (FDPIC)	2 KiB (hello) + 294 KiB (libc) = 296 KiB (+248 KiB)	0.3 KiB (hello) + 22 KiB (libc) = 22.3 KiB (5.7 KiB)

Considering the hello program size on the flash memory, the dynamic linked program is much bigger than the static linked one.

However, as the program size goes up, the advantage of dynamic linked will show (will be discussed later)

Why we need dynamic linked?

When the program becomes bigger or more programs are going to be executed:

- Programs can be put in SD card while dynamic linked libraries XIP on flash
- Programs can share a common library on flash

Run FDPIC, dynamic linked busybox (STM32H750)

Finally, we run dynamic linked busybox on STM32H750

```
BusyBox v1.37.0 (2026-04-17 04:21:07 CST) hush - the humble shell
Enter 'help' for a list of built-in commands.

~ # uname -a
Linux (none) 7.0.0 #19 Fri Apr 17 04:51:22 CST 2026 armv7ml GNU/Linux
~ #
```

Verify the Application XIP (STM32H750)

Finally, we run dynamic linked busybox on STM32H750

```
/ # cat /proc/self/maps
30010000-30012000 rw-p 00010000 1f:00 492288   /bin/busybox
30012000-30014000 rw-p 00005000 1f:00 463520   /lib/ld-uClibc.so.0
30014000-30016000 rw-p 00000000 00:00 0        [stack]
…
906000e0-906460e0 r-xp 00000000 1f:00 192      /lib/libc.so.0
906712c0-906772c0 r-xp 00000000 1f:00 463520   /lib/ld-uClibc.so.0
90678320-90688320 r-xp 00000000 1f:00 492288   /bin/busybox
```

Free RAM after boot (STM32H750)

We can use /proc/meminfo to see the free RAM

I print it in a minimal init program, it shows that we have 116 KiB free RAM

Memory Footprint

We can enable the DEBUG_FS, SYSFS, SLUB_DEBUG to analyse the slab usage

Although it doesn’t have extra RAM to enable these configs, we implemented a QEMU SoC model and added external RAM to do the analysis

With only SLUB_DEBUG enabled (SYSFS and DEBUF_FS off), we can find that the total slab usage is 692 KiB (by summarizing all the <num_slabs> * <pagesperslab> * 4 KiB in /proc/slabinfo)

After enabling the DEBUG_FS, SYSFS, SLUB_DEBUG, we can analysis the source of the kmalloc-192, kmalloc-4k… by reading /sys/kernel/debug/slab/…/alloc_traces

Remove sysfs and debugfs related stuff in the memory footprint (After removing these, the sum of RAM usage is 703 KiB, which is near to original 692 KiB)

Memory Footprint

Category	RAM usage
driver:stm32_pinctrl_gpio	199 KiB
driver:stm32_clk	159 KiB
sched	154 KiB
vfs	63 KiB
irq	36 KiB
driver:stm32_serial	22 KiB
workqueue	18 KiB
slab	17 KiB
procfs	16 KiB

Cut the unused device tree node

The 199 KiB stm32_pinctrl_gpio uses is unusually big, so I decompile the dtb and find that there are some unused nodes (GPIOs)

I just cut them off, and the free RAM after boot changed from 116 KB to 196KB (+80KB)

Now, let's talk about a feature-rich Linux on STM32F429-Discovery

After enabling some configs (STM32F429)

cellphone-prototype needs MMC, LCD display and touchscreen

Our steps are:

1. Start from tinyconfig
2. Enable configs that STM32 needs
3. Enable configs for peripherals
4. Turn off the exactly same configs that we turn off on the first board

After doing these, the kernel size comes to 1611 KiB
kernel + bootloader + dtb = 1633 KiB

2 MiB flash is actually small!

However, this almost hit the limit of tuning the config

- We have to introduce patches to push it further

After enabling some configs (STM32F429)

What's our target now:

- 6 MiB free RAM after boot
- Reduce the RAM usage while executing bigger user program

When the user program becomes bigger, it can not be placed on flash, which means it has to be loaded into RAM

- We can make the dynamic linked libraries XIP on flash to reduce the RAM usage

Space we need on flash (STM32F429)

- Dynamic linked libraries (uClibc, libdrm, libevdev, tslib) XIP on flash
- User programs are placed on SD card

How many space we need on flash?

uClibc + libdrm + libevdev + tslib + minimal init = 505KiB

Still need 90 KiB (1633 KiB + 505 KiB - 2 MiB) more space

Link Time Optimization (STM32F429)

With some patches, GCC LTO can be enabled

Image size: -48 KiB

Introduce patches (STM32F429)

Remove uneeded elements under /proc:

- PROC_STRIP (-9KiB)
- SCHED_DEBUG_OUTPUT (-8KiB)

Remove EABI unwind table (used only by panic backtracking):

- UNWIND_NONE (-77KiB)

Simplify Linux sheduler implementation:

- SCHED_DEADLINE_CLASS (-8KiB)
- SCHED_FAIR_TINY (-16KiB)
- SCHED_TOPOLOGY_MINIMAL (-4KiB)
- SCHED_NO_RICH_API (-4KiB)

Totally: -126 KiB

Finally, we got a 1437 KiB kernel image.

FDPIC with musl

We introduce patches to make musl FDPIC work

Finally, we got a smaller libc

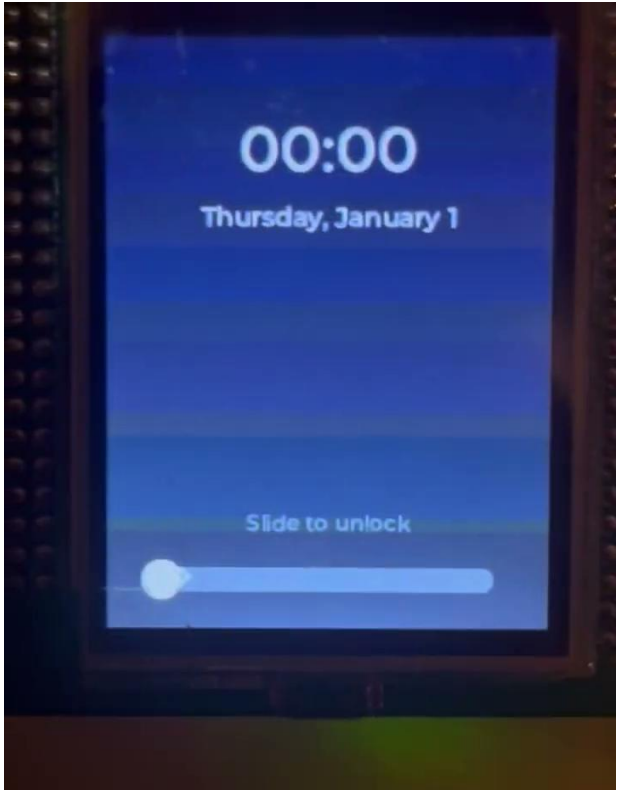
	Excutable Size	.bss + .data
uClibc-ng (ld + libc)	299 KiB	40 KiB
musl (ld + libc)	270 KiB (-29 KiB)	18 KiB (-22 KiB)

Run the cellphone-prototype (STM32F429)

Kernel + bootloader + dtb + libraries + minimal init = 1964 KiB can be placed on flash.

After booting, it has 6180 KiB of free RAM.

Free RAM is enough, it successfully runs.



https://www.youtube.com/shorts/oCtdOp8n9cE

RAM usage of cellphone-prototype (STM32F429)

	RAM usage
cellphone-prototype	676 KiB (.text) + 1040 KiB (.data + .bss)
buffer for DRM	160 KiB * 2
libc.so	16 KiB
tslib	8 KiB
libdrm	8 KiB
total	2068 KiB

The Extra RAM that the dynamic linked libraries uses is only 32 KiB

RAM usage comparison with non-XIP (STM32F429)

	With XIP	Without XIP
cellphone-prototype	1716 KiB	1716 KiB
buffer for DRM	320 KiB	320 KiB
libc.so	16 kiB	272 KiB
tslib	8 KiB	28 KiB
libdrm	8 KiB	64 KiB
total	2068 KiB	2400 KiB

We save 332 KiB of RAM through XIP

Part V: Conclusions and Future Work

Conclusions

- Linux-noMMU on MCUs is now practical
- Linux v7.x can run on 1 MiB RAM microcontrollers
  - XIP enables flash-resident kernel and userspace execution
  - FDPIC enables Linux-style shared library architectures on noMMU systems
- Modern MCU Linux systems can support:
  - graphics
  - touchscreen
  - dynamically linked applications
  - reusable middleware stacks

Future Work

- Kernel
  - further slab reduction
  - shrink CPU scheduler, delivering O(1)
  - better noMMU memory allocators
  - smaller scheduler configurations
  - upstreaming optimization patches
  - lightweight profiler/tracer
- Userspace
  - improved musl FDPIC support
  - richer graphics stack
  - package management for MCU Linux
  - connectivities
- Toolchain
  - improved LTO support
  - FDPIC optimization passes
  - profile-guided size optimization
  - libc customizations
- Potential applications:
  - industrial HMI systems
  - connected appliances
  - secure OTA-capable devices
  - educational Linux platforms

Questions?

Contact: Ching-Chun (Jim) Huang
<jserv@ccns.ncku.edu.tw>
National Cheng Kung University, Taiwan