



Linux Plumbers Conference 2026

Beyond the Command Line

Modernizing Kernel Boot Options: Resolving cmdline and Bootconfig Discrepancies


Masami Hiramatsu

Google / Linux Kernel Tracing & Bootconfig Maintainer


**LINUX
PLUMBERS
CONFERENCE**

PRAGUE, CZECHIA / OCT. 5-7, 2026

The Illusion of Predictable Kernel Parameter Handling

The Paradox

Administrators often pass kernel options in bootloader menus, such as **GRUB** or **systemd-boot**, to override system defaults during emergency boots or optimizations.

The standard expectation in these scenarios is a **LAST_WIN** behavior, where the bootloader's explicit command line argument successfully overrides the underlying static configuration.

However, the reality is starkly different. For many parameters, the bootloader's override is **silently ignored** by the kernel because the command line lacks a formal grammar or unified type system to enforce consistent priority rules.

1. Underlying Bootconfig Setting

```
kernel.loglevel = 3
```



2. Explicit GRUB Command Line Argument

```
loglevel=7
```



❗ Silent Override Failure

Instead of respect for **LAST_WIN**, the bootloader override is ignored. No error is thrown due to the lack of a standardized parser grammar across various kernel subsystems.

Sources of the Kernel Command Line

How command line parameters are constructed from multiple sources during boot

1. Bootconfig (initrd)

Structured configuration appended to initrd, expanding command line limits.

```
kernel {  
  param = A  
  key = foo  
}
```

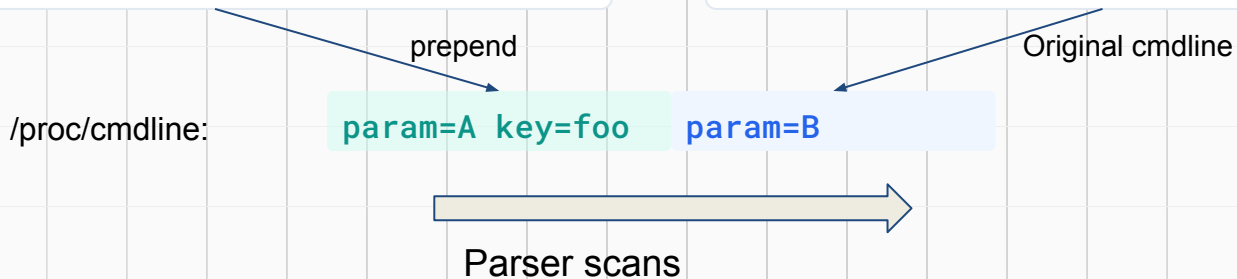
Scope: Complex key-value options, tracing, and init arguments.

2. Bootloader Arguments

Passed dynamically at boot time by GRUB, systemd-boot, or custom bootloaders.

```
param=B
```

Scope: Runtime overrides, emergency boots, and admin customization.



The Structural Complexity of /proc/cmdline

Decentralized Callback Responsibility

The root cause of silent override failures lies in how the kernel processes `/proc/cmdline`. Because there is no formal grammar or type system governing the command line, parameter parsing is entirely decentralized.

Each individual `__setup()` and `early_param()` callback is responsible for handling duplicate entries in its own ad-hoc way.

This lack of standardization means that when duplicate arguments collide, the outcome is entirely dependent on the specific implementation of that parameter's callback function.

Overwrite Behavior

Last Win

Subsequent instances overwrite previous callback values.

Input: param=A ... param=B → Result: B

Lock Out Behavior

First Win

First parsed value is locked; subsequent arguments are ignored.

Input: param=A ... param=B → Result: A

Accumulate Behavior

Cumulative

Callbacks collect or append all parameter instances.

Input: param=A ... param=B → Result: A, B

Real-World Example: Trace Instances Duplicate Bug

A prime example of parameter parsing errors occurred with dynamic trace instances (`trace_instance=`). An upstream patch titled **'tracing: Fix to avoid creating trace instances with duplicate names'** highlighted a severe flaw.

If an administrator specified multiple `trace_instance=` arguments with the identical name, all instances were created in memory, but only the very first one was actually accessible via `tracefs`.

Overwriting an already created trace instance during the boot process was impossible.

If dynamic trace instances suffer from this, how are other kernel parameters handling duplicates?

```
kernel/trace/trace.c - FIRST_WIN Strategy

/* Check if trace array already exists */
if (trace_array_find(name)) {
    pr_warn("Tracing: Instance %s already
exists\n", name);
    continue;
}
```

Safety Guard Fix

The adopted fix implements a strict **FIRST_WIN** strategy with an explicit boot warning, skipping subsequent duplicate allocations.



Auditing the Kernel Tree: The Numbers

83.4%

LAST_WIN Strategy

An exhaustive audit of all **761 legacy kernel parameter handlers** across the [linux.git](https://github.com/torvalds/linux) tree was conducted.

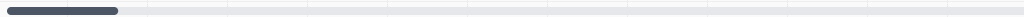
The key takeaway is that over 83% already follow LAST_WIN. Non-conforming behaviors are anomalous exceptions that violate user expectations.

STATISTICAL BREAKDOWN OF PARAMS

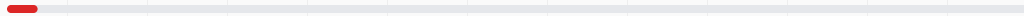
LAST_WIN Scalar overwrite (de facto standard) **635** (83.4%)



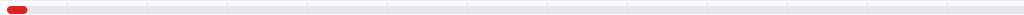
COMPLEX Subsystem state management **83** (10.9%)



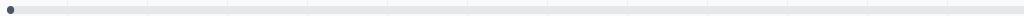
FIRST_WIN Write-once latch (breaks overrides) **23** (3.0%)



SCAN_ALL Accidental additive lists **15** (2.0%)



BITMASK_OR Monotonic flags (cannot unset) **5** (0.7%)



The 5 Parameter Collision Types

The audit identified five distinct parameter collision patterns across the kernel.

STANDARD	LAST_WIN	The variable is overwritten by subsequent declarations (e.g., <code>loglevel=</code> , <code>root=</code>). This is the standard behavior with no inherent risk.
RISK	FIRST_WIN	A guard drops later values (e.g., <code>ima_template=</code> , <code>earlyprintk</code>). This risks locking out bootloader overrides.
LIMITATION	SCAN_ALL	Appends to a list or array (e.g., <code>console=</code> , <code>initcall_blacklist=</code>). The issue here is that default values cannot be revoked.
LIMITATION	BITMASK_OR	Applies a bitwise OR operation (<code>flags = BIT</code>) (e.g., <code>iommu.strict</code> , <code>cgroup_no_v1</code>). These are monotonic flags with no off-switch.
COMPLEX	Two-Phase	Stores data on parse, but acts at initialization. This is used for memory or subsystem init and requires a rollback mechanism if parsed multiple times.



Phase 1 Proposal: Tagging kernel-parameters.txt

Currently, users and automated tools have zero visibility into parameter collision types. We propose documenting explicit collision behaviors.

PARAMETER NAME	COLLISION BEHAVIOR TAG	EXPECTED OVERRIDE REMEDY
loglevel=	[KNL] [LAST_WIN]	Later values override earlier ones. The final specified value takes precedence during boot initialization.
console=	[KNL] [SCAN_ALL]	Multiple occurrences activate all specified consoles. Use console=null to reset prior entries.
earlyprintk	[X86] [FIRST_WIN]	Only the first parameter takes effect. Subsequent occurrences are rejected and emit a warning.



Proposal Goal: Standardizing documentation structure helps both users and automated tooling reliably predict boot behavior semantics.

Phase 2 Proposal: Converting FIRST_WIN to LAST_WIN



Architectural Standardization

Standardize on **LAST_WIN** across the kernel tree to guarantee reliable command-line parameter overrides. The initial target scope focuses on converting **23 identified FIRST_WIN cases**.

Refactoring Action Plan

- **Remove accidental latch guards:** Eliminate constructs like `if (done) return;` that block overrides.
- **Two-Phase Initialization:** Delay allocation until command-line evaluation is fully completed.
- **Handle hardware constraints:** If states are physically irreversible, fail explicitly rather than silently.



Irreversible Hardware Warning

Mandatory Explicit User Alert

Handlers must never fail silently. Contradictory skipped parameters must output clean warnings to notify both developers and test harnesses.

Kernel Warning Implementation

```
// Warn on ignored early setups
pr_warn(
    "earlyprintk: '%s' ignored; \n"
    "already configured as '%s'\\n",
    buf,
    early_console->name
);
```

Pattern 1: Accidental FIRST_WIN (Convert to LAST_WIN)

The Problem (Latch Guard)

A defensive guard designed to prevent duplicate setups accidentally blocks legitimate overrides later in the boot sequence. This frequently occurs when a parameter merely sets a scalar, string, or enum in memory.

Real-World Case

In `ima_template.c`, the command line setup uses `if (template_setup_done) return 1;` acting as an accidental latch guard, locking out late boot overrides.

The Proposed Fix

Remove the latch guard entirely. This allows the latest parsed parameter to cleanly overwrite previous states, converting the logic into a standard, robust **LAST_WIN** behavior.

```
security/integrity/ima/ima_template.c

static int __init ima_template_setup(char *str)
{
- if (template_setup_done)
- return 1;

    ima_template = template_desc;
    template_setup_done = 1;

    return 1;
}
```

Standard Fix

Remove highlighted lines to allow cleanly overwriting `ima_template`.

Pattern 2: True FIRST_WIN (Reject with Warning)



Physical Constraint: Irreversible Hardware Setup

Parameters like `earlyprintk` perform immediate MMIO mapping and UART baud rate initialization. Tearing down or rolling back an active early console mid-parse is fundamentally unsafe.

Why Parameters Must Remain FIRST_WIN

Because early boot parameters demand early locking of physical hardware, the kernel must maintain strict `FIRST_WIN` semantic rules. In `arch/x86/kernel/early_printk.c`, the function checks for an existing registration and returns zero.

The Solution: Rejection Overlap Warning

To prevent risky parameter hijacking and avoid silent parsing failures, subsequent override attempts must be explicitly rejected with a system console warning.

`arch/x86/kernel/early_printk.c`

```
/* Reject override but warn admin */
if (early_console) {
    pr_warn("earlyprintk: '%s' ignored;\n"
            "already configured\n"
            , buf);
    return 0;
}
```

Pattern 3: Two-Phase Initialization (Deferred LAST_WIN)



Deferred LAST_WIN standard

Decouples parsing from execution into two distinct phases. Resolves dangerous naive LAST_WIN memory leaks (e.g. repeated `memblock_alloc()`) by storing intermediate parsed choices in static `__initdata`.

Phase 1: The Parse Phase

The system only validates and stores options in static `__initdata`. This cleanly overwrites previous values without allocating memory on every duplicate occurrence.

Phase 2: The Init Phase

Resources are allocated exactly once at `early_initcall()` using the final, resolved value.

Deferred Implementation

```
// Static temporary storage
static char *chosen_device __initdata;

// Parse Phase callback
static int __init device_setup(char *str)
{
    chosen_device = str;
    return 1;
}

// Init Phase allocation
early_initcall(device_setup);
```

Pattern 4: SCAN_ALL Lists (Support Reset Tokens)

Accumulation Behavior

The fourth pattern involves SCAN_ALL lists, where additive collections accumulate values rather than replacing them.

For example, `initcall_blacklist=` appends every provided symbol to a linked list using `list_add(&entry->next, &blacklisted_initcalls);`.

The Problem: If a Bootconfig blacklists a function, the bootloader has no mechanism to un-blacklist it because the parameter only appends.



Proposed Solution

Sentinel Reset Token

To fix this, we must preserve the accumulation behavior but introduce a reset sentinel token.

By supporting a token like `initcall_blacklist=none` or `clear`, the system can empty the list before adding new entries. (Or, `initcall_blacklist=!*`)

Kernel Precedent: *Passing `console=null` successfully resets the console selection to `ttynull`.*

Pattern 5: Monotonic Flags (Make BITMASK_OR Invertible)

The Core Problem: One-way bitwise OR operations lack an off-switch. Once a Bootconfig sets `flags |= FEATURE_BIT`, passing `param=off` on the command line does absolutely nothing.

Real-world Bug: In `drivers/iommu/iommu.c`, `iommu.strict` executes `iommu_cmd_line |= IOMMU_CMD_LINE_STRICT`, creating a one-way latch.

The Solution: Handlers should parse values using `kstrtobool()`. This allows developers to explicitly clear the bit based on the parameter value.

```
// Fix: Invertible Bitwise Handling

bool val;
if (kstrtobool(str, &val) == 0) {
    if (val)
        iommu_cmd_line |=
            IOMMU_CMD_LINE_STRICT;
    else
        iommu_cmd_line &=
            ~IOMMU_CMD_LINE_STRICT;
}
```



Section 02

Part 2: Towards Unified Bootconfig

Shifting focus to modern, structured configurations for predictable kernel setting management.

01 / Predictable Configuration

With `LAST_WIN` standardized across the kernel, command-line overrides applied on top of Bootconfig will finally become fully predictable.

This unlocks the ideal vision: transparently managing all system and kernel settings via structured Bootconfig files.



The Early Parameter Limitation

However, transitioning focus to modern boot configurations reveals a significant stumbling block. Bootconfig currently cannot handle early parameters, such as **early_param**. This limitation creates a barrier to seamless integration.

02 / Architectural Roadmap

The following sections will explore:

1. Why this limitation exists in the early boot phases
2. The benefits of moving fully to Bootconfig & solutions to overcome the wall.

Why Move from Cmdline to Bootconfig?



⚠️ Legacy Flat /proc/cmdline

- **Unformatted:** Single flat string.
- **Tight Limit:** Typically ~2–4KB (`COMMAND_LINE_SIZE`).
- **Fragile Syntax:** No comments allowed; relies on complex whitespace quoting.
- **Unordered:** Unsorted parameters make differences hard to diff.

✅ Modern Structured /proc/bootconfig

- **Hierarchical:** Clean tree structure with subkeys and namespaces.
- **Scalable Limit:** Dynamic sizing up to 32KB (`XBC_DATA_MAX`).
- **Rich Features:** Supports shell-style comments (#) and native arrays (`key = val1, val2`).
- **Maintainable:** Key-sorted structure yields clean, deterministic diffs.



The Boot Process Dilemma

While Bootconfig resolves readability, layout, and length limitations, it introduces a critical chicken-and-egg initialization sequence challenge during the kernel boot process.

The Early Param Paradox

The Primary Limitation

Bootconfig data is appended directly to the end of the `initrd` image. However, the physical memory location of the `initrd` is only resolved within the architecture setup function.

Because `setup_boot_config()` execution occurs long after early architecture parameters are parsed, the structured config data is available **too late** in the bootstrap sequence to influence critical architectural steps like `earlycon`, `mem=`, or `nokaslr`.

Boot Sequence Timeline Conflict



1. `parse_early_param()`

Early architecture parameters are executed here. Bootconfig data is **inaccessible** at this point.



2. `setup_arch()`

Resolves the `initrd` memory location. The system now knows where the bootconfig metadata resides.



3. `setup_boot_config()`

Identifies `initrd` and parses Bootconfig. This occurs **too late** in the sequence to influence architectural setup.

Workarounds & Their Friction



WORKAROUND 1: BUILD-TIME CONFIG_CMDLINE_FROM_BOOTCONFIG

Compiles the Bootconfig `kernel.*` tree directly into the kernel image at build-time (available in v6.x+).

This prepends the configuration data early in `setup_arch()` before `parse_early_param()` runs.



HIGH OPERATIONAL FRICTION

Modifying configuration parameters requires a **full kernel recompile**, completely blocking rapid runtime iteration and debugging.

WORKAROUND 2: INITRD-LESS Dummy Initrd Fallback

Bootconfig on Initrd-less systems is structurally limited. Modifying architecture boot protocols for every CPU architecture to support this is exceptionally daunting.



PROPOSED LOW-FRICTION SOLUTION

The bootloader passes a minimal container holding only the Bootconfig footer. The kernel extracts the Bootconfig, detects no filesystem, and proceeds to direct roots mount, requiring **zero architecture-specific changes**.

The Future: Bootloader Cooperation



Solving the Early Parameter Paradox

We need bootloaders to treat Bootconfig as a first-class citizen. This eliminates initialization dependencies and allows early parameters to function natively.



UEFI Standard

UEFI Configuration Table

The bootloader loads `bootconfig.bconf` dynamically as an EFI configuration table mapped to `LINUX_BOOTCONFIG_GUID`.

The kernel reads this table securely before `setup_arch()`, unlocking true early parameters natively.



Embedded & ARM

Device Tree Injection

Utilizes standard devicetree paths at `/chosen/bootconfig` to pass structured configuration.

U-Boot or platform firmware injects the bootconfig blob directly into the DTB during the early boot phase.



Legacy Fallback

Early Param Splitting

The bootloader parses Bootconfig, extracts critical `kernel.early.*` arguments into the legacy cmdline buffer.

Remaining structured data is passed as a standard bootconfig payload to preserve hierarchy.

Retaining Bootconfig Post-Boot

Currently, Bootconfig memory is discarded via `xbc_exit()` immediately after initialization. For modern scale systems with hundreds of configuration options, keeping this data alive unlocks critical core benefits:

1. Structured In-Kernel Registry Keeps the key-value tree alive in read-only memory, establishing an official, robust dynamic querying API.

2. Drop Fragile String Parsing Completely replaces vulnerable `/proc/cmdline` regex scanning inside `modprobe` and scripts.

3. Transparent Sysfs Node Exposes the live, runtime-resolved configuration tree under `/sys/kernel/bootconfig` for painless user-space querying.

```
bootconfig_registry.h

/* In-Kernel Registry API */
const char *xbc_get_module_param(
    const char *module,
    const char *key
);

# User-Space Runtime Query Node
$ cat /sys/kernel/bootconfig
kernel.sched_latency_ns = 24000000
module.ext4.lazy_itable_init = 1
module.pcie_aspm.policy = "powersave"
```

Tailoring to the Deployment Target

Bootconfig adoption can be tailored to specific production environments.

STREAM 01

Fast-Boot Embedded

For Fast-Boot and Minimal Embedded systems, the bootloader renders Bootconfig into the flat `cmdline` buffer at load time.

This results in **zero runtime parser overhead** in the kernel, making it optimal for sub-second boot requirements.

STREAM 02

Consumer Mobile

For Consumer and Mobile platforms (e.g., Android), vendors can ship system and vendor configuration in distinct **Bootconfig trees**.

This keeps the standard `Cmdline` strictly reserved for critical runtime bootloader or fastboot overrides.

STREAM 03

Enterprise & Cloud

For Enterprise and Cloud Servers, hundreds of complex kernel tuning knobs can be centralized in human-readable files.

This completely bypasses legacy **COMMAND_LINE_SIZE** truncation issues prevalent in large multi-socket machines.

Summary & Questions for Plumbers

Synthesizing the proposals, we have three main areas for discussion with the LPC audience.

1. Standardization

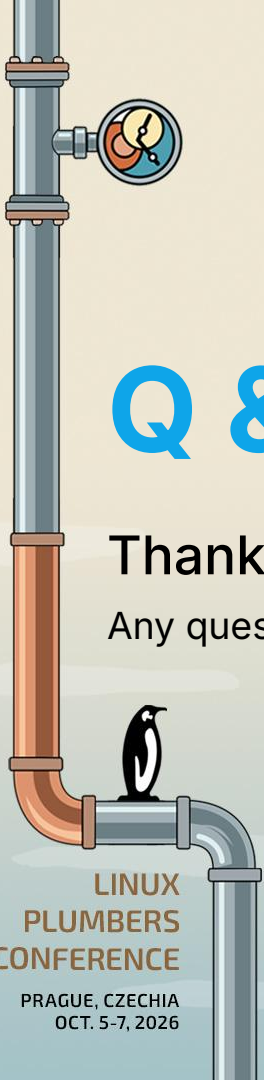
- Can we reach a consensus that **LAST_WIN** is the standard, and that **FIRST_WIN** requires an explicit warning?
- Should we proceed with adding collision tags (**[LAST_WIN]**, **[SCAN_ALL]**) to `kernel-parameters.txt`?

2. Architecture & Bootloaders

- Are bootloader maintainers (**GRUB**, **systemd-boot**, **U-Boot**) open to passing Bootconfig via a UEFI Table or Device Tree?
- Should we introduce a kernel fallback to accept a Bootconfig blob on initrd-less systems?

3. Runtime Integration

- Should we keep Bootconfig memory alive for driver and module parameter lookup post-boot?



Q & A

Thank you for listening!

Any questions?

LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA
OCT. 5-7, 2026