

Making the Kernel Chat: How To Get Debug Output

Richard Weinberger

2026-10-07



sigma star gmbh

Hello



Richard Weinberger

- Co-founder of sigma star gmbh
- Linux kernel developer and maintainer
- Strong focus on Linux kernel, lowlevel components, virtualization, security, code audits

sigma star gmbh

- Software Development & Security Consulting
- Main areas: Embedded Systems, Linux Kernel & Security
- Contributions to Linux Kernel and other OSS projects

Why This Talk?

- `Starting kernel ...` and then nothing
  - Everyone who has brought up a board knows the feeling
- "Just add a `printk()`" is the standard advice
  - Doesn't help when no output arrives at all
- Beginners usually know one tool: the serial console
  - The kernel has many more, most of them cost nothing
- This talk: a toolbox for getting output out of a dying kernel
  - Ordered from cheap to heavy
  - Embedded focus: arm, arm64, a bit of x86
  - No user space debugging, that's a different talk

The Game

- Before touching any tool, get the basics right
- Build with `CONFIG_DEBUG_INFO`, keep `vmlinux` and `System.map` of *every* image you flash
  - An oops without matching symbols is worth very little
- Know your UART: base address, register width, clock
  - Datasheet, device tree, `/proc/iomem` on a working system
- Know your boot flow: BootROM → SPL → U-Boot → decompressor → kernel
  - "Nothing on the console" means something different in every stage
- Always log the console session, every single time
  - `picocom --logfile`, `script`, `tee`
  - The one crash you didn't log is the one you needed

Anatomy of a Kernel Message

- `printk()` writes into a ring buffer: `__log_buf`
  - Size: `CONFIG_LOG_BUF_SHIFT`, default 2^{17} = 128 KiB
  - Static array in the kernel image, `dmesg` reads from it
- Registered consoles get the records from there
  - Serial, framebuffer, netconsole, …
  - Only messages below the console loglevel are pushed out
  - Filtering happens at the console, *not* in the buffer
- Boot consoles (`earlycon`, `earlyprintk`) poll the UART, no driver needed
  - Replaced by the real console once it probes
- The real console is a regular driver, registered during device probe
  - Everything before that point is buffered, not lost
  - Unless the kernel dies before the console comes up

Why You See Nothing

- Crash before the console driver probed
  - Buffered messages never reach the UART
- Wrong `console=`: device name, baud rate, port instance
- Serial driver built as a module, or `CONFIG_SERIAL_*_CONSOLE` not set
- `quiet` or a low `loglevel=` in the bootargs
  - Yes, that happens more often than you'd think
- Crash in the decompressor or in early assembly
  - No `printk()` down there
- Watchdog or hardware reset strikes before the UART FIFO is flushed
- Output stuck in the buffer: interrupts off, console lock held, `printk()` recursion
- Console suspended, `no_console_suspend` missing
- Hardware: wrong voltage level, pinmux, swapped RX/TX
  - Check with U-Boot first, it uses the same UART

Strategy

- Figure out *where* output stops, then pick a tool
- Nothing at all, not even from U-Boot
  - Hardware, UART levels, pinmux. Not this talk
- U-Boot talks, kernel says nothing
  - `earlycon`, `DEBUG_LL`, `earlyprintk`
- Kernel boots a bit, then silence
  - `ignore_loglevel`, `initcall_debug`, `keep_bootcon`, `ftrace_dump_on_oops`
- Board just resets, no oops seen
  - `ramoops/pstore`, `panic=`, the U-Boot log buffer trick
- Board hangs, console dead
  - Lockup and hung task detectors first, then Magic SysRq, kgdb, JTAG
- No serial port at all
  - netconsole for output, ping SysRq for input

Serial Console

- The baseline. If it's not working, fix it first
- Available on almost every board, even if hidden on a debug header
- `console=ttyS0,115200n8`
  - Driver-specific names: `ttymxc0` (i.MX), `ttyAMA0` (PL011), `ttyPS0` (Zynq), `ttySTM0`, ...
- Device tree: `chosen { stdout-path = "serial0:115200n8"; }`
  - No `console=` needed when set
- Pros: trivial, works in early user space, full oops output
- Cons:
  - Needs the driver probed: clocks, pinmux, DT node
  - Slow: 115200 baud ≈ 11 KiB/s, a large dump takes a while
  - Gone after reset, unless you log it (see rules)

Serial Console: Pitfalls

- Multiple `console=` entries: all get output, the *last* one becomes `/dev/console`
  - `console=tty0 console=ttyS0,115200` is the usual order
- `ttyS0` does not have to be `UART0` of your SoC
  - Numbering comes from DT aliases: `aliases { serial0 = &uart1; }`
- Driver as module: no console until the module loads. Build it in
- `CONFIG_SERIAL_8250_CONSOLE`, `CONFIG_SERIAL_IMX_CONSOLE`, ... must be set
- Suspend/resume debugging: `no_console_suspend`
- Boot console gets unregistered when the real one appears
  - Crash in that window? `keep_bootcon`
- Default loglevel hides `KERN_DEBUG` and friends
  - `ignore_loglevel` prints everything
- Timestamps help: `printk.time=1`

Serial Console: Example

U-Boot:

```
=> setenv bootargs 'console=ttymxc0,115200 earlycon ignore_loglevel printk.time=1'
=> saveenv
=> boot
```

Host:

```
$ picocom -b 115200 --logfile board-$(date +%F-%H%M).log /dev/ttyUSB0
```

- `earlycon` without arguments: UART from `stdout-path`
- Serial break in picocom: `C-a C-\`, needed for SysRq later

earlycon

- A polling boot console, before any driver infrastructure exists
  - No interrupts, no clock framework, no DT probing
- Works as soon as the UART is mapped, right after the MMU is on
  - Covers the gap between `Starting kernel ...` and the first real console
- `earlycon` bare: UART from DT `stdout-path` or ACPI SPCR
- Explicit: `earlycon=uart8250,mmio32,0x02800000`
  - `earlycon=pl011,0x09000000`
  - Vendor drivers: `ec_imx6q`, `lpuart32`, `cdns`, `msm_serial_dm`, `qcom_geni`, `linflex`, ...
- `CONFIG_SERIAL_EARLYCON` plus the matching `CONFIG_SERIAL_*_CONSOLE`
- Handover: `bootconsole [uart8250] disabled` when the real console takes over
  - `keep_bootcon` keeps both alive
- Pros: cheap, upstream, no code change
- Cons: UART must be initialised by the bootloader, needs a working page table

earlyprintk vs. earlycon

- x86: `earlyprintk=`, separate mechanism, needs `CONFIG_EARLY_PRINTK`
  - `earlyprintk=serial,ttyS0,115200[,keep]`
  - `earlyprintk=vga`, `earlyprintk=pciserial,00:16.3`, `earlyprintk=xdbc`, `earlyprintk=dbgp`
  - EFI framebuffer: `earlycon=efifb`, the old `earlyprintk=efi` is gone
- arm32: `earlyprintk` exists, but is a thin wrapper around `DEBUG_LL`
  - Needs `CONFIG_EARLY_PRINTK`, which needs `CONFIG_DEBUG_LL`
- arm64: no `earlyprintk` at all, `earlycon` is the only option
  - `earlycon=smh`: ARM semihosting when a JTAG debugger is attached
- Rule of thumb: `earlycon` everywhere, `earlyprintk` only on x86 and old arm32 kernels

earlycon: Example

```
earlycon=uart8250,mmio32,0x02800000 console=ttyS2,115200n8
```

What you should see:

```
[    0.000000] earlycon: uart8250 at MMIO32 0x0000000002800000 (options '')
[    0.000000] printk: bootconsole [uart8250] enabled
...
[    1.234567] printk: console [ttyS2] enabled
[    1.234567] printk: bootconsole [uart8250] disabled
```

- Base address: `reg` of the UART node in the DT, `/proc/iomem` on a running system, or the datasheet
- `mmio32` vs. `mmio`: register stride, wrong choice means garbage or silence

CONFIG_DEBUG_LL

- arm32 only: the lowest level output there is
  - Hardcoded UART, no driver, no allocation, no locks
  - Works in the decompressor, in `head.S`, anywhere
- Kconfig under “Kernel hacking”:
  - `CONFIG_DEBUG_LL=y`
  - `CONFIG_DEBUG_LL_UART_8250=y` or `CONFIG_DEBUG_LL_UART_PL01X=y`
  - `CONFIG_DEBUG_UART_PHYS`, `CONFIG_DEBUG_UART_VIRT`
  - `CONFIG_DEBUG_UART_8250_SHIFT`, `CONFIG_DEBUG_UART_8250_WORD` where needed
- `CONFIG_DEBUG_UNCOMPRESS=y`: `Uncompressing Linux...` from the decompressor
  - Not available on multiplatform kernels
- The big catch: the UART address is compiled in
  - Kconfig help: “Attempting to boot the kernel image on a different platform will not work”
  - Wrong address: kernel hangs or faults before printing anything
  - Debug kernels only, never ship it

printascii & Friends

- `arch/arm/kernel/debug.S` exports:
  - `printascii(const char *)`, `printch(char)`
  - `printhex2()`, `printhex4()`, `printhex8()` for values
- No header file, declare it yourself:
  - `extern void printascii(const char *);`
  - `setup.c` does exactly that for `early_print()`
- The “I’m here” technique: `printascii("A\n")`, `("B\n")`, … along the boot path
  - Crude, but finds the dying function in minutes
- `CONFIG_EARLY_PRINTK` + `earlyprintk` on the cmdline: all `printk()` output through `printascii()`
- arm64 equivalent: none in mainline
  - Write your own three-line `putc` to the UART TX register in `head.S`
  - Or get `earlycon` working as early as possible, it’s close enough

DEBUG_LL: Example

Kconfig fragment, i.MX6 with UART1:

```
CONFIG_DEBUG_LL=y
CONFIG_DEBUG_IMX6Q_UART=y
CONFIG_DEBUG_IMX_UART_PORT=1
CONFIG_DEBUG_UNCOMPRESS=y
CONFIG_EARLY_PRINTK=y
```

From C:

```
extern void printascii(const char *);
extern void printhex8(unsigned int);

printascii("setup_arch: before paging_init\
    n");
printhex8(__pa(swapper_pg_dir)); printascii
    ("\n");
```

From assembly:

```
    adr r0, 1f
    bl  printascii
    ...
1:  .asciz  "reached __enable_mmu\n"
```

Command Line: Output Control

- `console=ttyS0,115200`, `earlycon`: where output goes
- `loglevel=8`, `debug`, `ignore_loglevel`: how much of it
- `log_buf_len=4M`: bigger ring buffer, a chatty boot doesn't wrap
- `printk.time=1`: timestamps
- `printk.devkmsg=on`: unlimited user space writes to `/dev/kmsg`
- `printk.always_kmsg_dump=1`: kmsg dump (pstore!) also on a normal reboot
- `initcall_debug`: every initcall with timing, finds the one that never returns
- `initcall_blacklist=foo_init`: skip the offender
- `dyndbg="file foo.c +p"`: enable `pr_debug()` at boot
- `keep_bootcon`, `no_console_suspend`
- `sysrq_always_enabled`

Command Line: Crash Behaviour

- `panic=10`: reboot 10 s after a panic, `panic=-1`: immediately, `panic=0`: hang forever (default)
  - Hanging forever is nice for reading the console. Not in the field
- `oops=panic`, `panic_on_warn=1`: turn every oops/WARN into a panic
  - Together with pstore or kdump: every problem leaves a trace
- `panic_print=0x20`: replay *all* kernel messages on the console when panicking
  - Bit 0 tasks, 1 memory, 3 locks, 4 ftrace buffer, 6 all-CPU backtraces
- `softlockup_panic=1`, `hung_task_panic=1`, `nmi_watchdog=panic`, 1 (x86): see Silent Hangs
  - And check your BSP defconfig for `nowatchdog`, it's there more often than you'd like
- `ftrace_dump_on_oops`: later
- `crashkernel=256M`: later
- Not a crash, just waiting: `rootwait`, `rootdelay=5`
  - SD card or USB root takes a moment, otherwise `VFS: Unable to mount root fs`

Silent Hangs: Let the Kernel Complain

- A hang is worse than a crash: no oops, no panic, nothing printed
  - The kernel can watch itself and turn a hang into a backtrace. If you let it
- Soft lockup: a CPU loops in kernel mode with interrupts *on*, nothing else gets to run
  - Default: 2 × `watchdog_thresh` = 20 s
  - `BUG: soft lockup - CPU#1 stuck for 22s! [foo:123]` + backtrace
- Hard lockup: a CPU loops with interrupts *off*, not even the timer tick gets through
  - Needs an NMI or a buddy CPU to notice, default 10 s
  - `CPU0: Watchdog detected hard LOCKUP on cpu 1`
- Hung task: a task sleeps uninterruptibly (D state) for too long
  - Default 120 s, `INFO: task foo:123 blocked for more than 120 seconds.`
- RCU stall: a CPU or task holds up a grace period, default 21 s
  - `rcu: INFO: rcu_sched detected stalls on CPUs/tasks:` + per-CPU state + stacks
- Workqueue watchdog: a worker pool makes no progress for 30 s
  - `BUG: workqueue lockup - pool cpus=0 node=0 flags=0x0 nice=0 stuck for 31s!`
- All of them: just a warning by default. The system stays hung afterwards

Silent Hangs: Configuration & Pitfalls

- Kconfig, all off by default:
  - `CONFIG_SOFTLOCKUP_DETECTOR`, `CONFIG_HARDLOCKUP_DETECTOR`, `CONFIG_DETECT_HUNG_TASK`, `CONFIG_WQ_WATCHDOG`
  - `CONFIG_BOOTPARAM_SOFTLOCKUP_PANIC`, `_HARDLOCKUP_PANIC`, `_HUNG_TASK_PANIC`: panic instead of warn
- Hard lockup detector needs a way to interrupt a CPU with IRQs off
  - x86: perf NMI (`CONFIG_HARDLOCKUP_DETECTOR_PERF`)
  - arm64: pseudo-NMI, `CONFIG_ARM64_PSEUDO_NMI` *and* `irqchip.gicv3_pseudo_nmi=1`
  - Otherwise the buddy detector (`CONFIG_HARDLOCKUP_DETECTOR_BUDDY`): each CPU watches the next
    - SMP only, 8-12 s, can't catch all CPUs locking up at once
  - Single-core arm32: no hard lockup detector at all. Hardware watchdog + pstore it is
- Knobs: `watchdog_thresh=5`, `softlockup_panic=1`, `hung_task_panic=1`, `rcupdate.rcu_cpu_stall_timeout=10`, `workqueue.watchdog_thresh=10`
  - `softlockup_all_cpu_backtrace=1`, `hardlockup_all_cpu_backtrace=1`: show the other CPUs too
  - Sysctl only: `kernel.hung_task_timeout_secs`, `kernel.panic_on_rcu_stall`
  - RCU stall and `all_cpu_backtrace` output is huge, 115200 baud remembers

Silent Hangs: Example

Debug bootargs, everything short and everything panics:

```
watchdog_thresh=5 softlockup_panic=1 softlockup_all_cpu_backtrace=1
hung_task_panic=1 sysctl.kernel.hung_task_timeout_secs=30
rcupdate.rcu_cpu_stall_timeout=10 sysctl.kernel.panic_on_rcu_stall=1
workqueue.watchdog_thresh=10 panic_print=0x20 panic=-1
```

Same at runtime:

```
# sysctl kernel.watchdog_thresh=5 kernel.softlockup_panic=1
# sysctl kernel.hung_task_timeout_secs=30 kernel.hung_task_panic=1
```

What you get instead of silence:

```
[  142.511233] watchdog: BUG: soft lockup - CPU#2 stuck for 11s! [foo_worker:412]
[  142.511240] CPU: 2 PID: 412 Comm: foo_worker Tainted: G  O  6.12.0 #1
[  142.511247] pc : foo_wait_for_fifo+0x2c/0x48 [foo]
[  142.511260] Call trace:
[  142.511262]  foo_wait_for_fifo+0x2c/0x48 [foo]
[  142.511268]  foo_irq_thread+0x9c/0x140 [foo]
[  142.511300] Kernel panic - not syncing: softlockup: hung tasks
```

netconsole

- Sends every console message as a UDP packet
- When useful:
  - No UART pins reachable, or the serial port belongs to the product (modem, RS485)
  - Dozens of boards in a test rack: one syslog server instead of 40 USB adapters
- Static: `netconsole=...` on the cmdline, driver built in
- Dynamic: `CONFIG_NETCONSOLE_DYNAMIC`, configure at runtime via configfs
- Extended format (`+`): level, sequence number, timestamp per line, same as `/dev/kmsg`
- `r`: prepend the kernel release
- Receiver: anything that listens on UDP
- Pros: no cable, full Ethernet speed, many boards into one log
- Cons: next slide

netconsole: Pitfalls

- Starts after the NIC driver is up: zero help for early boot crashes
- The NIC driver must support netpoll (`ndo_poll_controller` or NAPI)
  - Most do, some vendor drivers don't. Then: silence
- No DHCP: source IP, interface and target MAC go on the cmdline
  - Receiver in another subnet: use the *gateway's* MAC
  - Target MAC left out: broadcast. Works, annoys everyone else on the segment
- UDP: packets get lost, especially during a panic flood
- VLANs, bonding, WiFi: usually not
- Interface names change between kernels (`eth0` vs. `end0`): use the MAC as `<dev>`
- Last words of a dying kernel go out with interrupts off, from a polled path
  - Works surprisingly well, but a hard hang still eats them

netconsole: Example

Static:

```
netconsole=+6665@192.168.1.20/eth0,6666@192.168.1.10/aa:bb:cc:dd:ee:ff
```

Receiver:

```
$ nc -u -l 6666 | tee netconsole.log
$ socat udp-recv:6666 -
```

Dynamic:

```
# cd /sys/kernel/config/netconsole && mkdir target1 && cd target1
# echo eth0 > dev_name
# echo 192.168.1.10 > remote_ip
# echo aa:bb:cc:dd:ee:ff > remote_mac
# echo 1 > extended
# echo 1 > enabled
```

- Cmdline targets show up as `cmdline0` once you `mkdir` them

pstore

- Persistent store: write crash output somewhere that survives a reboot
- Frontends, *what* gets stored:
  - dmesg: the last KiBs of the log on oops/panic (`CONFIG_PSTORE`)
  - console: everything the console sees (`CONFIG_PSTORE_CONSOLE`)
  - ftrace: function trace (`CONFIG_PSTORE_FTRACE`)
  - pmsg: user space writes to `/dev/pmsg0` (`CONFIG_PSTORE_PMSG`)
- Backends, *where* it goes:
  - RAM: `ramoops` (`CONFIG_PSTORE_RAM`), *the* embedded choice
  - Block device or MTD: `pstore/blk`, `mtdpstore` (`CONFIG_PSTORE_BLK`, `CONFIG_MTD_PSTORE`)
  - EFI variables (`CONFIG_EFI_VARS_PSTORE`), ACPI ERST, PowerPC NVRAM
- Reading after reboot:
  - `mount -t pstore pstore /sys/fs/pstore` (systemd does it for you)
  - Files: `dmesg-ramoops-0`, `console-ramoops-0`, `ftrace-ramoops-0`, `pmsg-ramoops-0`
  - `rm` a file to clear the record, otherwise it's back on the next boot
- `pstore.backend=ramoops` when more than one backend is available

pstore: Pros & Cons

- Pros:
  - Survives reboot, watchdog reset, even a panic with a dead console
  - No cable, no host, no network
  - Answers “device rebooted in the field, why?”: `systemd-pstore` archives it
  - `max_reason` controls what triggers a dump: Oops, Panic, or everything
- Cons:
  - Only records what reached the printk buffer. Crash before `printk()` works? Nothing
  - Backend must be initialised before the crash, early boot crashes are a gap
  - Record size is limited, long oopses get cut (`record_size`)
  - Compressed records (`.enc.z`) need `CONFIG_PSTORE_COMPRESS` on the reading side too
  - Needs someone to read and clear `/sys/fs/pstore`, or it fills up
- Test it before you need it: `echo c > /proc/sysrq-trigger`, reboot, look

ramoops

- The pstore RAM backend: a chunk of DRAM the bootloader and kernel leave alone
- DRAM keeps its content across a warm reset. Usually. That's the whole trick
- Configuration, pick one:
  - Device tree `reserved-memory` node, `compatible = "ramoops"`
  - Module parameters:
    `ramoops.mem_address=0x8f000000 ramoops.mem_size=0x100000`
  - Recent kernels: `reserve_mem=2M:4096:oops ramoops.mem_name=oops`
    - No DT change, no fixed address to pick
- Region layout: `record_size` × N dmesg slots, then `console_size`, `ftrace_size`, `pmsg_size`
- `ecc=1` or `ecc-size = <16>`: Reed-Solomon ECC, detects bit rot after reset
- `max_reason`: 2 = Oops and Panic (default), larger values also on reboot/shutdown
- `mem_type`: 0 write-combined, 1 unbuffered, 2 cached
  - Unbuffered is safest for the last words, slower

ramoops: Pitfalls

- The bootloader *must not* touch the region
  - U-Boot relocates itself to the top of RAM, loads kernel/DT/initrd somewhere: check `bdinfo` and your `loadaddrs`
  - SPL/BootROM scratch areas, DDR training may scribble
- Cold boot loses everything, a power cycle is not a reset
- Some SoCs reinitialise or scramble DRAM on reset. Then ramoops is dead on arrival
  - ECC errors on every boot are the hint
- `no-map` in the DT, otherwise the kernel treats it as normal RAM
- KASLR and memory layout don't matter: fixed physical address is the point
- Region too small: `console_size` below one boot log means you only see the end
- Dump arrives truncated: `record_size` smaller than the oops, raise it
- Check on a healthy boot: `dmesg | grep ramoops` shows the mapped region

ramoops: Example

Device tree:

```
reserved-memory {
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;

    ramoops@8f000000 {
        compatible = "ramoops";
        reg = <0 0x8f000000 0 0x100000>;
        record-size = <0x20000>;
        console-size = <0x40000>;
        ecc-size = <16>;
        no-map;
    };
};
```

Same via cmdline:

```
ramoops.mem_address=0x8f000000 ramoops.↲
    mem_size=0x100000
ramoops.record_size=0x20000 ramoops.↲
    console_size=0x40000
ramoops.ecc=1
```

Test and read:

```
# echo c > /proc/sysrq-trigger
... reboot ...
# ls /sys/fs/pstore/
console-ramoops-0  dmesg-ramoops-0
# head -2 /sys/fs/pstore/dmesg-ramoops-0
Panic#1 Part1
<6>[   42.123456] sysrq: Trigger a crash
```

ftrace as a Flight Recorder

- ftrace keeps a per-CPU ring buffer in RAM, separate from printk
- `trace_printk()`: like `printk()`, but
  - Goes into the trace buffer only, never to the console
  - Cheap enough for hot paths, interrupt and even NMI context
  - Prints a loud NOTICE banner at boot: not for production
- Tracers at boot: `ftrace=function_graph`, `trace_event=sched:*,irq:*`
- `ftrace_dump_on_oops`: on oops/panic, dump the trace buffer to the console
  - The last thing the kernel did before it died, with function granularity
  - `ftrace_dump_on_oops=orig_cpu`: only the CPU that crashed
  - Instances: `ftrace_dump_on_oops=foo=orig_cpu`
  - Runtime: `/proc/sys/kernel/ftrace_dump_on_oops`, manual dump with SysRq z
- `tp_printk`: tracepoints to printk as well. Can live-lock on a busy event
- Bonus on recent kernels: a trace instance that survives reboot
  - `reserve_mem=12M:4096:trace trace_instance=boot_map@trace`

ftrace: Pitfalls

- Dumping over serial is *slow*: 115200 baud, buffer of MiBs per CPU
  - Documentation says “several minutes to hours”. It is not joking
  - `trace_buf_size=64K`, `orig_cpu`, and trace only what you need
- Function tracer overhead changes timing, races may disappear
- A `trace_printk()` left in a driver gets you a friendly reminder on every boot
- Runtime configuration through tracefs: `/sys/kernel/tracing`
  - Needs `CONFIG_TRACING`, `CONFIG_FUNCTION_TRACER` and friends
- `tracing_on` is 0 after a dump, buffer stays frozen until re-enabled
- Not persistent by default: a reset loses it
  - Unless you use pstore’s ftrace frontend or the `reserve_mem` instance

ftrace: Example

```
ftrace=function_graph ftrace_dump_on_oops=orig_cpu trace_buf_size=256K
```

```
trace_printk("got irq %d, status 0x%08x\n", irq, status);
```

Runtime:

```
# echo 1 > /proc/sys/kernel/ftrace_dump_on_oops
# echo function_graph > /sys/kernel/tracing/current_tracer
# echo 'foo_*' > /sys/kernel/tracing/set_ftrace_filter
# echo z > /proc/sysrq-trigger     # dump right now
```

Output on oops:

```
Dumping ftrace buffer:
---------------------------------
   <...>-1   0d... 42.123456: foo_irq_handler <-__handle_irq_event_percpu
```

The Log Buffer Is Still in RAM

- Kernel died, nothing on the console, no pstore configured. Game over?
- No: the printk ring buffer is still sitting in DRAM
  - `__log_buf`, `1 << CONFIG_LOG_BUF_SHIFT` bytes, default 128 KiB
  - Static array in the kernel image, address in `System.map`
- Warm-reset the board, stop in U-Boot, read it from memory
  - Zero kernel configuration, works with any kernel
  - Rescued more than one bring-up
- Layout since 5.10: data ring of `printk_rb_static`
  - Each record: `[id (one long)][text]`, padded to long alignment, no newline
  - Timestamps and levels live in a separate array, `_printk_rb_static_infos`
  - Raw dump is readable text with a few binary bytes in between: `strings` is enough

U-Boot Dump: Pitfalls

- Virtual to physical: same offset as the kernel image itself
  - `phys(__log_buf)= phys(_text)+ (__log_buf - _text)`, both symbols from `System.map`
  - `phys(_text)` is where U-Boot put the kernel: `booti`/`bootm` print it
- KASLR moves the whole kernel: boot with `nokaslr`, or search RAM for a known string
- `log_buf_len=` on the cmdline: buffer is allocated dynamically via memblock, address unknown
  - Leave it off for this trick
- U-Boot itself: relocation and `loadaddr` may overwrite the area. `bdinfo` shows where it lives
- Caches: the last lines may still sit in dirty cache lines when the reset hits
- DDR reinit on reset: some SoCs/bootloaders clear or retrain. Then ramoops won't work either
- Hard reset via PMIC or power cycle: everything gone

U-Boot Dump: Example

```
$ grep -w -e '_text' -e '__log_buf' System.map
ffff800080000000 T _text
ffff800081e40000 b __log_buf
```

Image loaded at `0x40200000`, so `__log_buf` is at `0x40200000 + 0x1e40000 = 0x42040000`:

```
=> md.b 0x42040000 0x40
42040000: 00 00 00 00 00 00 00 00 42 6f 6f 74 69 6e 67 20    ........Booting
42040010: 4c 69 6e 75 78 20 6f 6e 20 70 68 79 73 69 63 61    Linux on physica
```

Better: dump the whole buffer to a file

```
=> tftpput 0x42040000 0x20000 192.168.1.10:logbuf.bin
=> fatwrite mmc 0:1 0x42040000 logbuf.bin 0x20000
$ strings -n 8 logbuf.bin | less
```

- Structured alternative with JTAG or a crash dump: `gdb vmlinux` + `lx-dmesg`

Magic SysRq

- Kernel is alive but user space is dead, or the console shows nothing new
- `CONFIG_MAGIC_SYSRQ=y`, enable with `sysrq_always_enabled` or `kernel.sysrq=1`
- Over serial: send BREAK, then the key within 5 seconds
  - picocom: `C-a C-\`, then the key. Two BREAKs in a row: a normal break
- From a shell: `echo t > /proc/sysrq-trigger`
- Useful keys:
  - `9`: console loglevel to 9, everything gets printed from now on
  - `R`: replay the whole kernel log on the consoles. Missed the early part? Here it is again
  - `t`: all tasks with backtraces, `w`: blocked tasks only
  - `l`: backtrace of every CPU, finds the one spinning
  - `p`: registers, `d`: held locks, `m`: memory info
  - `z`: dump the ftrace buffer
  - `c`: crash now. Test your pstore and kdump setup with this one
  - `g`: enter kgdb

SysRq Over the Network: Ping SysRq

- Board hangs, no serial access: remote rack, product without UART. netconsole is output only
- Out-of-tree patch “net: sysrq via icmp” by Carsten Emde (OSADL)
  - Part of the PREEMPT_RT patch queue for years, e.g. `5.4.93-rt51`
  - *Not* in mainline, apply it yourself. It’s small
- Adds `/proc/sys/net/ipv4/icmp_echo_sysrq`: a 32-bit cookie, 0 = off
- An ICMP echo request with exactly 57 bytes of payload, the cookie at payload offsets 20 and 50, and the SysRq key byte right after each cookie calls `handle_sysrq()`

```
board# echo 0x01020304 > /proc/sys/net/ipv4/icmp_echo_sysrq
host$  ping -c1 -s57 -p0102030468 board     # 0x68 = 'h': help
host$  ping -c1 -s57 -p010203046c board     # 0x6c = 'l': all-CPU backtraces
host$  ping -c1 -s57 -p0102030463 board     # 0x63 = 'c': crash, test pstore
```

- `-p` repeats the 5-byte pattern, so cookie and key land at offsets 20 and 50 by themselves
- Runs from the NIC’s receive path: as long as interrupts and the network stack still work, SysRq `l`, `t`, `w` reach netconsole or pstore
- Caveats: whoever knows the cookie can crash the box, pick a random one, debug builds

Decoding the Oops

- You got an oops. Now make it useful
- `CONFIG_DEBUG_INFO`, `CONFIG_DEBUG_BUGVERBOSE`, `CONFIG_KALLSYMS` are mandatory
  - Frame pointers or unwind tables make the backtrace trustworthy
- `scripts/decode_stacktrace.sh`: adds `file:line` to every frame

```
$ ./scripts/decode_stacktrace.sh vmlinux < oops.txt
```

- `scripts/faddr2line`: single symbol+offset to source line

```
$ ./scripts/faddr2line vmlinux foo_probe+0x1a/0x80
```

- `gdb vmlinux`, then `l *foo_probe+0x1a` or `disassemble foo_probe`
- The `Code:` line is the instruction stream at the crash, `objdump -d` finds it
- Modules: pass the module directory to `decode_stacktrace.sh`
- Keep `vmlinux` per build, without it nothing above works

kgdb / kdb

- A real debugger inside the kernel, over the serial port
- `CONFIG_KGDB=y`, `CONFIG_KGDB_SERIAL_CONSOLE=y`, `CONFIG_KGDB_KDB=y`, `CONFIG_FRAME_POINTER=y`
- `console=ttyS0,115200 kgdboc=ttyS0,115200 nokaslr`
  - `kgdbwait`: stop at the earliest point and wait for the debugger
  - Runtime: `echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc`
- Enter: SysRq g, a breakpoint, or an oops
- kdb: no host tools needed, you type at the console
  - `dmesg`, `bt`, `ps`, `summary`, `go`
- gdb: `gdb vmlinux`, `set serial baud 115200`, `target remote /dev/ttyUSB0`
  - Full source level debugging, breakpoints, memory inspection
- One UART for console and debugger: `agent-proxy` from the kgdb project splits it into two TCP ports
- Cons: kernel must still take interrupts and breakpoints
  - No help against hard hangs, an oops before `kgdboc` is up stays silent

kexec / kdump

- Reserve memory for a second, small crash kernel: `crashkernel=256M`
  - arm32 places it automatically, arm64 needs a 2 MiB aligned region
- `CONFIG_KEXEC` or `CONFIG_KEXEC_FILE`, `CONFIG_CRASH_DUMP`, `CONFIG_PROC_VMCORE` in the crash kernel
- Load it once the system is up:

```
# kexec -p /boot/Image --initrd=/boot/initrd-crash \
    --append="root=/dev/mmcblk0p2 1 nr_cpus=1 reset_devices"
```

- On panic, the crash kernel boots and exposes the old memory as `/proc/vmcore`
  - `cp /proc/vmcore /mnt/dump`, or `makedumpfile -l -d 31 /proc/vmcore dump`
- Analysis on the host:
  - `makedumpfile --dump-dmesg dump dmesg.txt`: just the log
  - `crash vmlinux dump`, then `log`, `bt`, `ps`
  - `drgn`: scripted analysis in Python
- Cons: RAM budget, crash kernel needs storage or network, lots of moving parts
  - Test it with SysRq c before you rely on it

JTAG / OpenOCD

- When the kernel is truly stuck: halt the CPU and look at memory
  - Works with interrupts off, in the decompressor, in `head.S`, anywhere
- `openocd -f interface/jlink.cfg -f target/imx6.cfg`, then `telnet localhost 4444`

```
> halt
> reg pc
> dump_image logbuf.bin 0x42040000 0x20000
```

- Or gdb: `gdb vmlinux`, `target extended-remote :3333`
  - `CONFIG_GDB_SCRIPTS` + `scripts/gdb/vmlinux-gdb.py`: `lx-dmesg`, `lx-ps`, `lx-lsmod`
  - `lx-dmesg` parses the ring buffer properly, no `strings` needed
- Semihosting: `earlycon=smh` prints via the debugger, no UART at all
- Cons:
  - Hardware and SoC support needed, debug access often fused off on production units
  - MMU on: gdb sees virtual addresses, OpenOCD sees physical ones
  - Multi-core, power domains, secure world: every SoC has its own surprises

Last Resorts

- "The device just reboots, no oops anywhere"
  - Usually the watchdog. Disable it while debugging, or let the kernel feed it early (`CONFIG_WATCHDOG_HANDLE_BOOT_ENABLED`)
  - `panic=-1` plus ramoops: a panic leaves a record instead of a hang
  - U-Boot shows the reset reason on many SoCs: `Reset cause: WDOG`
- Nothing works, not even `DEBUG_LL`
  - Blinkenlights: toggle a GPIO or LED in the suspicious code, watch with a scope or your eyes
  - `writel()` straight into the GPIO register, no driver, no `printk()`
- Bisect: `git bisect` between the last working and the first broken kernel
  - Boring, reliable, often faster than clever
- Compare with a known-good boot log of the same board, line by line
  - You did log it, right?

Cheat Sheet

Symptom	First try	Then
U-Boot talks, kernel silent	`earlycon`	`DEBUG_LL`, JTAG
Boots a bit, then nothing	`ignore_loglevel`, `initcall_debug`	`keep_bootcon`, `ftrace_dump_on_oops`
Oops shown, but useless	`decode_stacktrace.sh`	`faddr2line`, kgdb
Resets on its own	ramoops	`panic=-1` + ramoops, U-Boot dump
Hangs, console dead	lockup detectors + `*_panic=1`	SysRq `l`, `t`, kgdb, JTAG
Hangs, no serial, has network	ping SysRq (RT patch) + netconsole	ramoops
No UART on the product	netconsole	pstore/blk, pmsg
Need the whole picture	kdump + crash	drgn

Summary

- A dying kernel leaves traces. You just need to know where to look
- Keep the serial console, always logged
- `earlycon` goes into every bootargs, it costs nothing
- `ramoops` goes into every device tree, it costs 1 MiB
- `ignore_loglevel printk.time=1` while debugging, `panic=` set deliberately
- Lockup and hung task detectors on, and let them panic
- Keep `vmlinux` and `System.map` of what you flashed
- U-Boot can read what the kernel wrote
- The heavy tools exist. Most days you won't need them

Further Reading: Kernel Documentation (1)

- Kernel command line, every option in this talk: https://docs.kernel.org/admin-guide/kernel-parameters.html
- Serial console, `console=` semantics: https://docs.kernel.org/admin-guide/serial-console.html
- printk basics, loglevels, `/dev/kmsg`: https://docs.kernel.org/core-api/printk-basics.html
- Dynamic debug (`dyndbg=`): https://docs.kernel.org/admin-guide/dynamic-debug-howto.html
- `DEBUG_LL`, no prose docs, read the Kconfig help: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm/Kconfig.debug
- arm / arm64 booting requirements, what the bootloader must set up: https://docs.kernel.org/arch/arm/booting.html, https://docs.kernel.org/arch/arm64/booting.html

Further Reading: Kernel Documentation (2)

- netconsole: https://docs.kernel.org/networking/netconsole.html
- ramoops: https://docs.kernel.org/admin-guide/ramoops.html
- ramoops DT binding: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/devicetree/bindings/reserved-memory/ramoops.yaml
- pstore/blk and mtdpstore: https://docs.kernel.org/admin-guide/pstore-blk.html
- `systemd-pstore`: https://www.freedesktop.org/software/systemd/man/latest/systemd-pstore.service.html
- ftrace: https://docs.kernel.org/trace/ftrace.html
- Using ftrace to debug, `ftrace_dump_on_oops`, persistent buffers: https://docs.kernel.org/trace/debugging.html
- Boot-time tracing: https://docs.kernel.org/trace/boottime-trace.html

Further Reading: Kernel Documentation (3)

- Magic SysRq: https://docs.kernel.org/admin-guide/sysrq.html
- Lockup detectors: https://docs.kernel.org/admin-guide/lockup-watchdogs.html
- RCU stall warnings: https://docs.kernel.org/RCU/stallwarn.html
- Sysctls: hung tasks, `panic_on_*`, watchdogs: https://docs.kernel.org/admin-guide/sysctl/kernel.html
- Bug hunting, decoding an oops: https://docs.kernel.org/admin-guide/bug-hunting.html

Further Reading: Kernel Documentation (4)

- Debugging guides index: https://docs.kernel.org/process/debugging/index.html
- kgdb/kdb: https://docs.kernel.org/process/debugging/kgdb.html
- `lx-dmesg` and the other gdb scripts: https://docs.kernel.org/process/debugging/gdb-kernel-debugging.html
- kdump: https://docs.kernel.org/admin-guide/kdump/kdump.html

Further Reading: Tools (1)

- U-Boot `md`, `tftpput`, `fatwrite`: https://docs.u-boot.org/en/latest/usage/index.html
- picocom: https://github.com/npat-efault/picocom
- agent-proxy, kgdb console splitter: https://git.kernel.org/pub/scm/utils/kernel/kgdb/agent-proxy.git
- kexec-tools: https://git.kernel.org/pub/scm/utils/kernel/kexec/kexec-tools.git
- makedumpfile: https://github.com/makedumpfile/makedumpfile

Further Reading: Tools (2)

- crash: https://crash-utility.github.io/
- drgn: https://drgn.readthedocs.io/
- OpenOCD user guide: https://openocd.org/doc/html/index.html
- Ping SysRq patch (OSADL, PREEMPT_RT): https://www.osadl.org/monitoring/add-on-patches/5.4.93-rt51...5.4.93-rt51/net-ipv4-icmp-ping-sysrq.patch.html
- elinux.org kernel debugging tips, old but still useful: https://elinux.org/Kernel_Debugging_Tips

FIN



Thank you!

Questions, Comments?

Richard Weinberger
richard@sigma-star.at