USB Data transfer stall investigation report

USB Mass Storage Gadget Stall — Investigation Report

Date: 2026-07-24 **Board:** Custom carrier board, TI AM6234 SoC (`ti,am623-custom-board` , `ti,am625` family) **Kernel:** Linux 6.6.15, vendor tree (`6.6.15-arm64`), Debian-derived embedded Linux distribution **USB controller under test:** Synopsys DesignWare USB3 (DWC3) gadget core, `31000000.usb` , glue driver `f900000.dwc3-usb` (TI `dwc3-am62` wrapper), `GSNPSID = 0x5533330b` → DWC_usb3 core, **revision 3.30b Gadget function:** legacy BOT `f_mass_storage` ("Mass Storage Function, version: 2009/09/11"), LUN backing store an eMMC partition

Investigated over a serial console using on-target `ftrace` and `debugfs` ; no source-level kernel changes made.

---

1. Symptom (as reported)

USB mass-storage gadget works reliably through a USB hub, on Windows, Linux, and macOS hosts. When connected **directly to the host** (no hub), on **Windows and Linux** hosts (not observed on macOS), transfers intermittently stall: the host issues a SCSI READ, and the response arrives 20–30 seconds later, or the port resets and re-enumerates. Symptom is intermittent but frequent under sustained transfer.

---

2. Root cause (confirmed by direct hardware tracing)

The DWC3 gadget controller silently fails to raise the transfer-completion interrupt/event for a bulk endpoint (`ep1in`) under normal, correctly-armed transfer conditions. The CPU sits fully idle waiting for that interrupt. After ~20 seconds the host's own SCSI/USB-storage command timeout expires, the host issues a hard bus reset, and the device only recovers because of that host-forced reset — not because of anything the device driver does.

This looks like a hardware/microcode-level dropped-completion defect in the DWC3 core (or an undocumented interaction specific to this SoC integration), **not**:

- a USB link/PHY/signal-integrity problem,
- a Linux runtime-PM / clock-gating / autosuspend problem,
- a SuperSpeed bulk-streams protocol problem (this port is USB2 High-Speed only — see §4),
- a scheduler/IRQ-starvation problem (CPU is provably idle throughout each stall — see §5.3),

- an event-FIFO drain race (the hardware event-count register stays at zero throughout each stall — see §5.4),
- anything in our own gadget-management userspace daemon (confirmed unrelated — see §4.3).

A hub in the path appears to change enough low-level bus timing/traffic pattern to avoid triggering the underlying condition; it does not fix it.

---

3. Reproducibility

In a single ~40-minute capture window during one continuous large-file transfer session, **13 independent stalls** were recorded, all with the same signature:

#	Gap start (monotonic)	Gap end (monotonic)	Duration
1	2222.070069	2242.087830	20.018 s
2	2315.669616	2335.686457	20.017 s
3–13	(11 more)		20.000 – 20.013 s

All 13 stalls: same endpoint (`ep1in`), same duration band (**20.00–20.02 s**, i.e. clustered almost exactly on 20s — consistent with a host-side class-driver command timeout, not a random hang), all terminated by a hardware `Reset [U0]` event followed by full device re-enumeration.

---

4. Investigation timeline and ruled-out hypotheses

4.1 Initial state check

```
$ ls /sys/class/udc/
31000000.usb
$ cd 31000000.usb && cat state
configured
$ cat /sys/kernel/debug/usb/31000000.usb/link_state
Sleep

```

4.2 Hypothesis A (ruled out): USB2 LPM / runtime-PM autosuspend

Early in the investigation, the glue device showed aggressive runtime-PM autosuspend:

```
/sys/devices/platform/bus@f0000/f900000.dwc3-usb/power/control: auto
/sys/devices/platform/bus@f0000/f900000.dwc3-usb/power/runtime_status:
active
```

```
/sys/devices/platform/bus@f0000/f900000.dwc3-usb/power/autosuspend_delay_ms: 100

```

with power/clock supplied through the SoC's System Controller (TI SCI) power/clock domains:

```
/sys/devices/platform/bus@f0000/f900000.dwc3-usb/supplier:platform:44043000.system-controller:power-controller
/sys/devices/platform/bus@f0000/f900000.dwc3-usb/supplier:platform:44043000.system-controller:clock-controller

```

This looked plausible initially, but was **conclusively ruled out** by direct ftrace capture: zero `clock_enable` / `clock_disable` / `power_domain_target` / `device_pm_callback_*` events occurred during any of the captured 20-second gaps. No power-management transition happens during a stall.

4.3 Hypothesis B (ruled out): our own gadget-management userspace daemon

We have an in-house iterative fix effort already in place for USB gadget issues, with several successive build variants of our gadget-management daemon (baseline through several incremental fix attempts). The stall behaviour is **identical across all daemon build variants**; the daemon only manages LUN/configfs setup in userspace and cannot influence in-kernel TRB/endpoint completion handling, which is where the actual defect lives. This matches the trace evidence directly — the mass-storage kernel thread activity stops cleanly at the start of every stall and only resumes after the kernel-level `Reset` event.

(Side finding, not the root cause but worth fixing: the exported LUN backing file is the same raw partition simultaneously mounted read-write locally on the target. On an unclean host disconnect this LUN gets ejected — likely by a disconnect-safety path in the daemon — and is not always automatically re-attached in some build variants. This caused the drive to disappear mid-session during testing and is a real filesystem-corruption risk independent of the stall bug — see repeated `FAT-fs: Volume was not properly unmounted` *kernel log entries.)*

4.4 Hypothesis C (ruled out): SuperSpeed Bulk Streams

The first capture's dwc3 trace showed `streams 16` on the endpoint and link-state strings like `Reset [U0]`, which was initially (incorrectly) read as evidence of SuperSpeed operation with a lost ERDY/stream-ready handshake. This was **corrected** by checking the device tree directly:

```
$ cat .../of_node/dr_mode
peripheral
$ cat .../of_node/maximum-speed
high-speed

```

This port is **hard-limited to USB2 High-Speed in the device tree** — it has never negotiated SuperSpeed at any point in this investigation. The dwc3 driver's debug/trace code reuses the same `U0` / `U1` /... link-state naming and the same "streams" endpoint field regardless of negotiated speed, which is what caused the initial misread. A subsequent live test (force-rebinding the UDC) made no difference to the stall — consistent with the port already only ever running High-Speed.

This means the defect is not specific to SuperSpeed Bulk Streams protocol — it reproduces on plain USB2 High-Speed bulk transfers.

Note for anyone wanting to test at Full-Speed instead: `/sys/class/udc/31000000.usb/maximum_speed` is registered **read-only (`0444`)** on our kernel build, in both the bound and unbound UDC state — attempting to write to it fails with `Permission denied` even as root with a full capability set. This isn't a permissions issue to work around; the attribute simply isn't implemented as writable in this vendor kernel fork. Testing at a different negotiated speed requires changing `maximum-speed` directly in the device tree and rebooting.

4.5 Hypothesis D (ruled out): scheduler / IRQ-thread starvation

See §5.3 below — directly disproven by `sched_switch` / `sched_wakeup` tracing during a live stall.

4.6 Hypothesis E (ruled out): lost/stuck hardware event (driver fails to drain it)

See §5.4 below — directly disproven by continuous polling of the DWC3 event-count register during multiple live stalls.

---

5. Direct evidence from the DWC3 controller

5.1 First captured stall — full trace context

Captured via Linux `ftrace` (`events/dwc3/*` tracepoints), 16 MB buffer, kernel 6.6.15.

Immediately before the stall — the gadget driver correctly arms the transfer and the hardware acknowledges the endpoint command as successful:

```
file-storage-516  [002] d....  2222.070051: dwc3_ep_queue: ep1in: req 00000000c00f7d26 length 0/16384 zsI ==> -115
file-storage-516  [002] d....  2222.070060: dwc3_prepare_trb: ep1in: trb 00000000449a8bb0 (E105:D103) buf 000000008fbf8000 size 16384 ctrl 00000811 sofn 00000000 (Hlcs:sC:normal)
file-storage-516  [002] d....  2222.070069: dwc3_gadget_ep_cmd: ep1in: cmd 'Update Transfer' [30007] params 00000000 00000000 00000000 --> status: Successful

```

Then: total silence for 20.018 seconds (no `dwc3_event` of any kind). The gap ends with a hardware-initiated bus reset and full re-enumeration:

```
irq/478-dwc3-517  [000] D....  2242.087830: dwc3_event: event (00000101): Reset [U0]
irq/478-dwc3-517  [000] D....  2242.087855: dwc3_gadget_ep_disable: ep1in: mps 512/1024 streams 16 burst 0 ring 105/103 flags E:swBp:<
irq/478-dwc3-517  [000] D....  2242.087888: dwc3_gadget_ep_cmd: ep1in: cmd 'End Transfer' [30c08] params 00000000 00000000 00000000 --> status: Successful
irq/478-dwc3-517  [000] D....  2242.087913: dwc3_gadget_giveback: ep1in: req 000000007922c83b length 0/16384 zsI ==> -108
irq/478-dwc3-517  [000] D....  2242.087937: dwc3_gadget_giveback: ep1in: req 00000000c00f7d26 length 0/16384 zsI ==> -108
irq/478-dwc3-517  [000] D....  2242.087943: dwc3_gadget_ep_disable: ep1out: mps 512/1376 streams 16 burst 0 ring 15/15 flags E:swBp:>
irq/478-dwc3-517  [000] D....  2242.087967: dwc3_gadget_ep_cmd: ep1out: cmd 'End Transfer' [20c08] params 00000000 00000000 00000000 -> status: Successful
file-storage-516  [002] .....  2242.088078: dwc3_free_request: ep1in: req 000000007922c83b length 0/16384 zsI ==> -108
file-storage-516  [002] .....  2242.088090: dwc3_free_request: ep1in: req 00000000c00f7d26 length 0/16384 zsI ==> -108
irq/478-dwc3-517  [000] D....  2242.139086: dwc3_event: event (00000201): Connection Done [U0]
irq/478-dwc3-517  [000] D....  2242.172255: dwc3_event: event (0000c040): ep0out: Transfer Complete (sIL) [Setup Phase]
irq/478-dwc3-517  [000] D....  2242.172265: dwc3_ctrl_req: Get Device Descriptor(Index = 0, Length = 64)

```

The `-108` giveback status is `-ESHUTDOWN` — both pending 16384-byte requests are force-aborted by the reset, not completed. The device is then re-enumerated from scratch starting with `Get Device Descriptor`, i.e. the host treated this as a dead/unresponsive device and power-cycled the port logically.

5.2 DWC3 `DSTS` register decode during the gap

`DSTS` (Device Status Register, global offset `0xc70c`) was polled continuously (~every 200 ms) throughout the gap by our own gadget-daemon watchdog thread.
Decoding `USBLNKST` (bits 21:18) and `SOFFN` (bits 17:3, frame counter) from the raw values:

```
t=2222.177355 raw=0x00022c38 LNKST=0 SOFFN=1415
t=2223.984565 raw=0x0003f008 LNKST=0 SOFFN=15873
t=2230.215418 raw=0x000205b0 LNKST=0 SOFFN=182     (counter wrapped, still incrementing)
t=2236.438001 raw=0x00021950 LNKST=0 SOFFN=810
t=2241.862911 raw=0x00036580 LNKST=0 SOFFN=11440
t=2242.064580 raw=0x000397e8 LNKST=0 SOFFN=13053

```

`LNKST=0` = link state **ON / U0 (fully active)** for the entire gap. `SOFFN` increments continuously and wraps — the physical link between host and device never dropped, never suspended, and kept receiving normal periodic bus traffic (SOF/keepalive) the whole time. **The**

bus was electrically and physically fine; only the endpoint's transfer-complete notification failed to appear.

5.3 Second capture — scheduler proof the CPU was idle, not starved

To rule out the possibility that the interrupt fired but the driver/IRQ thread was simply delayed by CPU contention, `sched_switch` / `sched_wakeup` tracepoints (filtered to the dwc3 IRQ thread and the mass-storage kernel thread) were added alongside the dwc3 event trace for a second, longer capture session (13 stalls total — see §3). Full context for one representative 20.017 s gap:

```
file-storage-518  [001] d....  2315.660232: dwc3_ep_queue: ep1in: req 000000005bdf6b26 length 0/13 zsI ==> -115
   ... (endpoint command queued normally) ...
file-storage-518  [001] d....  2315.669616: sched_switch: prev_comm=file-storage prev_pid=518 prev_prio=120 prev_state=S ==> next_comm=swapper/1 next_pid=0 next_prio=120

  [ *** 20.017 seconds of complete silence on every traced CPU — no dwc3 events, no sched events for the traced threads *** ]

<idle>-0          [000] dNh..  2335.686457: sched_wakeup: comm=irq/478-dwc3 pid=519 prio=49 target_cpu=000

```

`<idle>-0` is the kernel's idle task. It is the entity that performs the wakeup at t+20.017s — meaning **CPU0 itself was sitting completely idle**, doing nothing, fully available to run the IRQ thread instantly, for the entire 20-second span. This directly disproves any scheduler-starvation or CPU-contention explanation: there was nothing to schedule around. The interrupt genuinely did not occur until the host-forced reset arrived.

5.4 Live GEVNTCOUNT (event-FIFO) polling — no stuck/undrained hardware event

A background poller was run independently on-target for the duration of the second capture, sampling the DWC3 global event-count register (`GEVNTCOUNT(0)` , offset `0xc40c`) via `debugfs` `regdump` roughly every 50–100 ms:

```
$ grep -c 'GEVNTCOUNT(0) = 0x00000000' gevnt_monitor.log
3445
$ grep -v 'GEVNTCOUNT(0) = 0x00000000' gevnt_monitor.log
2251.27 ... GEVNTCOUNT(0) = 0x00000004 ...
2336.75 ... GEVNTCOUNT(0) = 0x00000004 ...

```

Out of 3,447 samples spanning the entire second capture (including all 13 stalls), `GEVNTCOUNT(0)` was non-zero on only **2 isolated samples**, both immediately drained on the very next poll — normal, momentary event-processing latency, not a stuck event. **During every one of the 13 confirmed stalls, the event count register read exactly zero the entire time.** The hardware genuinely never places a completion event in its own event FIFO for the

stalled transfer — this is not a case of the driver failing to notice/drain an event that the hardware did raise.

5.5 Core identification

```
$ grep GSNPSID /sys/kernel/debug/usb/31000000.usb/regdump
GSNPSID = 0x5533330b

```

`0x5533` = ASCII "U3" (Synopsys DesignWare USB3 core ID prefix), revision field `0x330b` → **core revision 3.30b**.

```
$ cat /proc/device-tree/compatible
ti,am623-custom-board ti,am625

```

---

6. Mitigation attempts (device-side, no kernel rebuild available)

No compiler or kernel build tree is present on this image, so a real fix at the `dwc3` driver level could not be attempted in this session. Two userspace-only recovery strategies were built and tested live, to see whether the ~20s stall could at least be shortened:

1. **Detection.** After some iteration (an `ep1in` / `ep1out` `trb_ring`-polling approach proved unreliable for reasons not fully root-caused, and a continuous `trace_pipe` consumer worked but added enough CPU load to itself measurably slow down transfers), a **low-frequency (1 Hz) tail of** `/sys/kernel/debug/tracing/trace` proved reliable and cheap (~1% CPU): once per second, check whether the most recent `ep1in: cmd 'Update Transfer' ... Successful` line is newer than the most recent `ep1in ... Transfer In Progress` line; if so, and it's been armed for more than ~2s, declare a stall. This correctly detected multiple live stalls with no false positives.
2. **Recovery — and why it's the wrong lever.** On detection, the script forced `echo disconnect > /sys/class/udc/31000000.usb/soft_connect` followed by `echo connect > ...` ~300ms later. This *does* work at the USB link level — a host-side capture confirmed the reset — and cuts the stall from ~20s down to ~2–3s. **However, it makes the user-visible outcome worse, not better.** The host reported `0x800701B1` (`ERROR_DEVICE_NOT_CONNECTED`) and the copy operation aborted outright, and separately the exported filesystem was left in a bad state after repeated triggers.

The reason: `soft_connect` toggles the D+ pull-up, which is electrically indistinguishable from a real unplug/replug — Windows' `usbstor.sys` tears down and recreates the whole device stack rather than retrying the in-flight command. This is a fundamentally different, more disruptive event than the **host-initiated protocol-level bus reset** that naturally ends each stall

today (visible in the traces as `dwc3_event: event (00000101): Reset [U0]`) — that kind of reset is part of the standard USB Mass Storage Bulk-Only Transport reset-recovery procedure, which `usbstor.sys` handles gracefully by retrying the failed command, not by failing the whole copy.

Implication for a real fix: any device-side recovery needs to happen at the endpoint/command level (e.g. `DEPCMD End Transfer` + re-arm on just the stuck endpoint) rather than via the pull-up, so the host never sees a topology change — but issuing endpoint commands directly isn't exposed through any existing sysfs/debugfs interface on this build, only through the in-kernel `dwc3` driver itself. This effectively confirms the fix has to land in the kernel driver (or as a firmware/silicon erratum workaround), not as a userspace workaround.

Caution for reproducing this: repeatedly forcing `soft_connect` toggles, especially during writes, risks filesystem corruption on the exported LUN — compounded here by the LUN backing file being the same partition that's simultaneously mounted read-write locally on the gadget (see §4.3 side note). Run `fsck` / `chkdsk` on that filesystem before trusting its contents after any test session that used this mitigation.

---

7. Untested candidate fixes — DWC3 quirk properties already compiled into this kernel

Since no toolchain is available on this board (§6), a real fix has to be validated by the team that owns the build pipeline. To narrow that search, the actual `dwc3.ko` binary on this image was inspected directly (`grep -a` over the module file — a definitive check, not a documentation lookup) to see which of the standard upstream DWC3 device-tree quirk properties are actually compiled into this driver build, independent of what's currently set in the device tree:

```
KO=/lib/modules/6.6.15-arm64/kernel/drivers/usb/dwc3/dwc3.ko
grep -ac -- 'snps,<property-name>' "$KO"

```

Result: 18 of 19 checked upstream quirk properties are compiled in (only `snps,dis-off-mode-quirk` was absent, which is a useful negative control — it confirms the check isn't just matching everything). Cross-checked against the current device tree (§4.4 / §6): **none of these properties are currently set anywhere** — the `usb@31000000` node's `of_node` only has `dr_mode`, `maximum-speed`, `interrupts`, `interrupt-names`, `reg`, `status`; the parent glue node (`compatible = "ti,am62-usb"`) only has clock/power-domain/PHY-refclk properties. The driver is running on bare defaults.

Two of the confirmed-present properties map unusually well onto the observed defect and are the recommended first experiments, in priority order:

1. `snps,parkmode-disable-hs-quirk` (and its sibling `snps,parkmode-disable-ss-quirk`, also present). DWC3 has a power-saving "park mode" for bulk/interrupt endpoints: when an endpoint has no more queued work its internal datapath resources get parked, and have to be un-parked on the next transfer. This is a documented erratum-prone area specifically under **rapid re-arming of the same endpoint** — which is precisely our pattern: a second `Update Transfer` DEPCMD lands ~18µs after the first completes (§5.1). This quirk forces the core to never park High-Speed endpoint resources, at a small power cost, in exchange for (potentially) avoiding whatever race causes the dropped completion event.
2. `snps,dis-start-transfer-quirk`. Targets a known erratum area in how the core processes the transfer-start command sequence — directly on-target given the stuck event in every capture is a `Start` / `Update Transfer` DEPCMD that the hardware acknowledges as `Successful` but never follows up on.

Both are plain boolean device-tree properties on the `usb@31000000` node (`snps,dwc3` compatible) — enabling them requires only a DTB change and reboot, no compiler needed.

Caveats: this was verified against general upstream DWC3 driver knowledge, not this kernel's actual source — the exact register bits and semantics these properties trigger in `drivers/usb/dwc3/core.c` for *this* build should be confirmed before relying on them. Confirming a property is compiled in is not the same as confirming it's the correct fix for this defect; treat both as the most promising available experiments, not a diagnosis.

---

8. Summary for the kernel/silicon vendor

- **SoC:** TI AM6234 (AM62x family)
- **USB IP:** Synopsys DesignWare USB3 (DWC3), core revision **3.30b**, integrated via TI's `dwc3-am62` glue driver
- **Mode:** USB2 High-Speed peripheral only (`dr_mode=peripheral`, `maximum-speed=high-speed` in DT) — **not** SuperSpeed
- **Kernel:** 6.6.15 (vendor tree, `6.6.15-arm64`)
- **Defect:** A correctly armed, hardware-acknowledged bulk-IN transfer (`DEPCMD Update Transfer` returns `Successful`) never generates the corresponding `Transfer In Progress` completion event/interrupt. The link stays in `U0`/ON, SOF numbering continues incrementing normally, and the controller's own event-count register (`GEVNTCOUNT`) stays at zero — i.e. no event is ever placed in the event FIFO for that transfer. The CPU/IRQ thread is provably idle and waiting, not starved. Recovery only happens because the host times out (~20s, consistent with a default SCSI/USB-storage class-driver command timeout) and

forces a hard port reset, which the device correctly handles, disabling/re-enabling endpoints and re-enumerating from scratch.

- **Trigger conditions (empirical):** occurs under direct host connection without an intervening USB hub; not observed (or much rarer) when a hub is in the path; occurs on both Windows and Linux hosts; occurs on `ep1in` specifically, both for full 16 KB bulk-data-phase transfers and for the short 13-byte BOT command-status-wrapper transfer — i.e. it is not tied to a specific transfer size, so it looks like a timing/race condition in the core's completion-notification path rather than a fixed data-size erratum.
- **Reproducibility:** 13 occurrences captured in one ~40-minute session, all clustered at 20.00–20.02 seconds duration.
- **Ruled out:** USB2 LPM/L1, runtime-PM/clock-gating, SuperSpeed Bulk Streams/ERDY handshake, scheduler starvation, event-FIFO drain race, and our own gadget-management userspace daemon (all build/fix variants behave identically).

Open questions for TI / Synopsys

1. Is there a published erratum for DWC_usb3 core rev 3.30b (or the TI AM62x `dwc3-am62` integration) matching "armed TRB / successful `Update Transfer` DEPCMD with no subsequent completion event on a bulk endpoint"?
2. Is there a known interaction between **not** having a hub in the topology (i.e. direct root-port attachment) and this core's LTSSM/notification logic that would explain the hub masking it? Working hypothesis (§7, unverified): the hub's added repeater latency loosens the timing of `f_mass_storage`'s buffer-pipelined back-to-back `Update Transfer` commands (~18µs apart in our captures) enough to avoid a race in the core's command-processing pipeline.
3. Are there recommended `GUCTL` / `GUCTL1` / `GUCTL2` quirk bits for this core revision on AM62x that address dropped bulk-endpoint completion events? Specifically: does `snps,parkmode-disable-hs-quirk` or `snps,dis-start-transfer-quirk` (both already compiled into this kernel's `dwc3.ko` but unset in the device tree — see §7) address this failure mode?

---

9. Appendix — commands used to capture this evidence

```
# Arm ftrace (dwc3 tracepoints)
echo 0 > /sys/kernel/debug/tracing/tracing_on
echo 16384 > /sys/kernel/debug/tracing/buffer_size_kb
echo nop > /sys/kernel/debug/tracing/current_tracer
echo 1 > /sys/kernel/debug/tracing/events/dwc3/enable
echo 1 >
/sys/kernel/debug/tracing/events/power/device_pm_callback_start/enable
```

```
echo 1 > /sys/kernel/debug/tracing/events/power/device_pm_callback_end/enable
echo 1 > /sys/kernel/debug/tracing/events/power/clock_enable/enable
echo 1 > /sys/kernel/debug/tracing/events/power/clock_disable/enable
echo 1 > /sys/kernel/debug/tracing/events/power/power_domain_target/enable
echo > /sys/kernel/debug/tracing/trace
echo 1 > /sys/kernel/debug/tracing/tracing_on

# (second capture) add filtered scheduler tracing
echo 'prev_comm ~ "*dwc3*" || next_comm ~ "*dwc3*" || prev_comm ~ "file-storage*" || next_comm ~ "file-storage*"' \
  > /sys/kernel/debug/tracing/events/sched/sched_switch/filter
echo 'comm ~ "*dwc3*" || comm ~ "file-storage*"' \
  > /sys/kernel/debug/tracing/events/sched/sched_wakeup/filter
echo 1 > /sys/kernel/debug/tracing/events/sched/sched_switch/enable
echo 1 > /sys/kernel/debug/tracing/events/sched/sched_wakeup/enable

# ... reproduce the stall on the host side (direct connection, no hub) ...

# Freeze and inspect
echo 0 > /sys/kernel/debug/tracing/tracing_on
grep -vE 'dwc3_readl|dwc3_writel' /sys/kernel/debug/tracing/trace > trace_filtered.txt

# Find timestamp gaps > 1s (gaps.awk, run against trace_filtered.txt):
#   ts = field 4 of each ftrace line (strip trailing ':'); flag any gap > 1.0s
#   between consecutive timestamps, printing the surrounding lines.

# Live event-FIFO (GEVNTCOUNT) polling, background, on-target:
while true; do
  grep -E 'GEVNTCOUNT\(0\)|DEVTEN =|DSTS =|GDBGLTSSM' /sys/kernel/debug/usb/31000000.usb/regdump
  usleep 50000
done   # logged to file, filtered for change-on-write to keep size manageable

# Core/board identification
grep GSNPSID /sys/kernel/debug/usb/31000000.usb/regdump
cat /proc/device-tree/compatible
cat .../of_node/dr_mode
cat .../of_node/maximum-speed
```

10. Gemini Recommendations

[ti am62x gets slow usb - Google Search]

Hints 1, 2 and probably 4 sound promising. 3 could be tested to see if this is the root cause.

Direction of thought is that the PC overruns the device with requests and device gets stalled because it cannot follow up. There are some hints on PC and device side how to avoid this. According to the theory, the HUB is slowing down the requests, so that the stall will not happen.