Episode Book

EPISODE #6704

Bypassing Android's Built-In USB Audio Stack

PUBLISHED	20 November 2026 at 13:16 UTC
DURATION	32:44
LISTEN	myweirdprompts.com/episode/android-usb-audio-driver-14663/
HOSTS	Corn, Herman, Dorothy, Hilbert
SCRIPT MODEL	z-ai/glm-5.3
VOICES	omnivoice
PIPELINE	V5.36
SOURCES	16 cited by the research (listed at the end)
BOOK GENERATED	20 November 2026 at 21:20 UTC

SYNOPSIS

Android's built-in USB audio stack is a Linux inheritance topped with a consumer audio server built for ringtones and phone calls. Even with native AudioRecord or AAudio, vendor layers can cap captures at 16 bits, add phantom gain, and apply a hard limiter before an app sees a sample. The episode examines what AOSP actually promises for USB Audio Class 1, why Google's Oboe library relies on a QuirksManager device list, and how claiming a device through UsbManager lets an app bypass the system audio stack.

TRANSCRIPT



Corn

Twenty-two. I counted the USB-C cables in this flat this morning, and twenty-two is the number.



Herman

That's not counting the drawer, is it? Because the drawer is a different jurisdiction.



Corn

The drawer is Daniel's problem. The point is, every single one of those cables can carry audio. And Daniel's prompt this week is basically, so why can't any of them carry it well? He wants to record real audio on Android through a USB-C microphone or interface, and he's asking us to lay out what the platform actually gives an app, how a custom driver stacks up against the built-in support, which he calls, quote, pretty neglected in the past, and what best practice looks like today if you need genuine recording quality rather than whatever the phone feels like handing you.



Herman

And he's earned the question. He's not asking it abstractly. He records hours of speech data on an Android phone, so this is the wall he hits every week.



Corn

Then let's start with what the wall is made of.



Herman

The short version is that USB audio on Android is a Linux inheritance that nobody ever adopted. The chain is straight kernel plumbing: the USB device talks to the host controller, the controller to USB core, USB core to the standard Linux USB audio driver, that goes into ALSA, then an Android hardware abstraction layer picks it up and hands it to AudioFlinger, the central audio server, which finally gives it to your app through AudioTrack or the capture equivalent. At no point in that chain does Android insert anything Android-specific. It's a pass-through with a name tag on it.



Corn

Say more about that name tag, actually. Because I think people hear "Android is built on Linux" and assume Linux audio quality follows automatically.



Herman

And on the desktop it mostly does. Plug a class-compliant interface into a Linux laptop and it just works — ALSA binds the standard driver, you get a clean device node, whatever bit depth and sample rate the hardware advertises. Daniel's own laptop proves it, and we'll get to that. The difference is that desktop Linux has a whole ecosystem of audio software sitting on top of that plumbing, deliberately using it. Android inherited the plumbing and then built a consumer audio server on top whose job is ringtones, phone calls, and making sure your navigation app can duck your music. The plumbing was never the bottleneck. The building on top of it was.



Corn

And the documentation is refreshingly blunt about it.



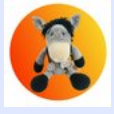
Herman

Refreshingly, yes. The AOSP page on USB digital audio, and this page was updated last month, says, and this is a direct quote, there are no APIs specific to USB digital audio. For advanced usage, the automatic routing can be disabled in Developer Options. That's it. That's the entire API surface for USB audio. Your advanced usage option is a toggle in Developer Options.



Corn

The advanced feature is turning the feature off.



Herman

Which, to be fair, is sometimes what you want — if you're playing music through a USB DAC and you don't want a phone call yanking the route away mid-song, that toggle is your friend. But it tells you the state of things. The platform's position is: we will automatically route audio to a compatible USB peripheral when we see one, and beyond that, good luck.



Corn

What does the app actually see, then? If I'm writing a recorder, what's in my hands?



Herman

Generic audio APIs, all of which have USB as a blind spot rather than a feature. From Java or Kotlin you've got AudioRecord, the standard capture class, and the one thing guaranteed about it is the sample rate: forty-four thousand one hundred hertz, guaranteed on all devices, and nothing else is. Below that there's AAudio, the native low-latency API, from API twenty-six onward, and AAudio is the first place you get any real device control, because you can tell a stream which device to attach to. You enumerate devices through AudioManager, you filter for the type that means USB, and you pass that device ID into the stream builder. And even there, the documentation warns you that the request may not yield the requested device. You ask, and then you check what you got. It's a suggestion box.



Corn

You ask for the microphone and you might get the speakerphone.



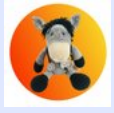
Herman

You ask for the microphone and you might get the built-in mic, because something in the vendor's audio policy decided your USB device was a playback route only, or it wasn't enumerated yet, or the routing table just didn't agree with you that day. And the reason that's survivable at all is Oboe, Google's C++ wrapper around all of this. Oboe picks AAudio on newer devices, falls back to the older OpenSL ES on older ones, and it ships with a component called the QuirksManager.



Corn

I want to sit on that name for a second.



Herman

It deserves it. The QuirksManager exists to correct for problems on specific devices. It is a maintained list of individual phones and tablets whose USB or audio implementations are broken in ways the general API can't absorb, and the workaround logic lives in the library, per device. That is the load-bearing confession of the whole ecosystem. Google's own guidance for unreliable audio hardware is a blacklist with a polite name.



Corn

And to be clear, this isn't a few dusty old phones from the SD card era.



Herman

No, the list is alive. It gets updated for current flagships. When your brand-new phone's audio HAL misbehaves in a way that breaks the general path, the fix isn't in your phone — it'll never be in your phone — it's a conditional in a Google library that says, if the device ID matches this one, route around it. Every Oboe app on Earth carries that list around like a vaccination record.



Corn

And MIDI gets the red carpet while audio waits outside.



Herman

That's the funny part, if you like this kind of comedy. The android.media.midi package is transport-agnostic, it supports virtual devices, it's had real version progress — MIDI 2.0 over USB since Android thirteen, MIDI 2.0 over virtual transports since fifteen. Meanwhile the page describing actual audio capture has said the same thing for a decade. Somebody inside Google cares about the protocol with no sound in it.



Corn

There's a theory about that, isn't there — that MIDI is easy to support because it's just control data.



Herman

That is exactly the theory. MIDI messages are tiny, timing-tolerant, and stateless enough that a bug is a wrong note rather than a ruined take. Audio is continuous, high-bandwidth, and merciless — a missed deadline is an audible click, forever, in someone's recording. Supporting MIDI well is a weekend. Supporting audio well is a decade of OEM firefighting. So the ecosystem did the weekend project and left the decade on the shelf.



Corn

So let's put numbers on the ceiling. What does the built-in path claim to support?



Herman

Officially, a subset of USB Audio Class 1 in host mode, and that goes back to Android 5.0. Class 1 is the older, simpler edition of the USB audio specification. Within that subset: PCM only, which is the plain uncompressed format, bit depths of sixteen, twenty-four or thirty-two, sample rates from eight kilohertz up to forty-eight, and one or two channels. And then the page adds, and I love this sentence, more advanced features are not validated or claimed.



Corn

That's a lawyer wrote that.



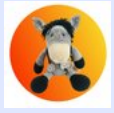
Herman

A careful one. It means anything outside the list — multichannel interfaces, higher sample rates, the second edition of the spec — is not promised, not tested, and not their problem. There's no documented UAC2 support in host mode at all, which is the striking part this late, because Class 2 has been the normal way to do anything beyond stereo since roughly the twenty-tens. Every serious audio interface on the market is a Class 2 device. The documented Android path is one spec edition behind the entire industry.



Corn

And just so nobody at the back misunderstands — Class 1 versus Class 2 isn't about quality per se. A sixteen-bit forty-eight-kilohertz mono stream can be perfectly professional for speech.



Herman

Right, it's about capability boundaries. Class 1 tops out around the numbers we just listed, and it does it over full-speed USB. Class 2 is what lets a device do multichannel, higher rates, and more flexible clocking. If you're recording one voice at forty-eight kilohertz, Class 1 is enough on paper. The tragedy is that Android doesn't even reliably deliver the paper. Which leads us to your next question, which I can see coming.



Corn

But you said the claim is sixteen, twenty-four, thirty-two bit. Does a twenty-four-bit interface actually give you twenty-four bits?



Herman

This is where it gets good, and it's exactly where Daniel's own measurements come in, so let me put the two side by side. On paper, AOSP says up to thirty-two. In practice, the classic public test here is from an audio company called Superpowered, who tried to build a DJ system on a Nexus 9 tablet running stock Android 6.0 with a Behringer USB sound card, and measured what the stack actually delivered. Sixteen bits. Hard ceiling. Two channels maximum, so a multichannel DJ card showed up as one stereo pair and nothing else. Forty-eight kilohertz cap. The documentation said one thing, the hardware delivered another, and the gap was the OEM's audio layer deciding what you were getting.



Corn

Ten years ago, though. Hasn't that closed?



Herman

Daniel's numbers say it hasn't closed the way you'd hope. He's on a current phone, a OnePlus with a current Android build, and he ran the test properly — native AudioRecord, floating point capture, explicitly routed to the USB device, and he tried every audio source the platform offers, including the raw unprocessed ones. Same result every time. The device-side encoding stayed sixteen-bit, and every take peaked at exactly minus zero point nine eight dBFS with a limiter signature on it. The vendor's audio layer was applying twelve dB of gain and a hard limiter before his app ever saw a sample. Same headset plugged into his Linux laptop: neither artifact. Clean capture, no limiter, straight through the standard Linux driver.



Corn

And let's make that concrete for a second, because minus zero point nine eight dBFS is a weirdly specific number to keep hitting.



Herman

It's the fingerprint. Real audio, real speech, real room noise — peaks land wherever they land. When every single take tops out at the same value just under digital zero, that's not your voice, that's a ceiling. It's like photographing a landscape and every print comes back with the horizon at exactly the same height. The camera decided, not the scene. And twelve dB of phantom gain means his quiet takes weren't quiet and his loud takes were already crushed, and no amount of careful mic technique on his end could undo any of it, because it happened upstream of the app.



Corn

The phone is applying makeup to the audio before the app can object.



Herman

And there is no objection mechanism. That's the crux. The audio sources — voice recognition, unprocessed, whatever — are requests. The vendor layer can and does ignore them. His phone even reports that it doesn't support the unprocessed source, which is the system honestly announcing that every recording is going through the sauce.



Corn

Which is a sentence in the documentation's family. Fine. So what did Daniel actually do about it? Because he did do something about it.



Herman

He wrote his own USB audio driver. Inside the app. In user space, no root, on a stock phone, and this is the part I want to take slowly because it's the answer to the custom-versus-built-in question in miniature. The trick is that Android does give apps direct USB device access through the UsbManager, with user permission, and a connection object you can claim an interface on with a force flag. Forcing the claim detaches the kernel's own USB audio driver, the snd-usb-audio module, so the device stops belonging to Android's audio stack entirely and starts belonging to his app. Then he builds libusb with the native development kit, wraps the connection's file descriptor into libusb so he can talk to the raw endpoints, and speaks USB Audio Class directly: he parses the device's descriptors himself, sets the sample rate himself, and reads and sets the feature unit, which is where the hardware volume control lives.



Corn

The feature unit being the actual physical gain knob.



Herman

The actual gain stage in the microphone electronics, the one Android never exposes to anyone. His hardware gain range is minus twenty-four to plus ten dB, and through the built-in path there was no way to know it existed. Through his own driver, it's a setting.



Corn

One thing you said in passing — it has to be native code. Why?



Herman

Because Java's USB API supports bulk and interrupt transfers, and audio streams over USB use isochronous transfers, the kind with guaranteed timing and no retries. The Java layer simply doesn't have them. So the custom driver is necessarily C or C++ talking through libusb. It's not a hack anyone does for fun; it's what the platform's shape forces.



Corn

And it's worth spelling out what "no retries" means there, because it's the whole reason isochronous exists.



Herman

Isochronous transfers trade reliability for timing. The host reserves bandwidth up front, the device gets its slot every millisecond, guaranteed — and if a packet is lost or corrupted, nobody asks for it again. There's no retransmission, because a late audio sample is worthless. A late file chunk is just a file chunk. Bulk transfers, the kind Java does support, will happily wait and retry until the data arrives intact, which is exactly right for a document and exactly wrong for a drum hit. The wire protocol of audio is built around the assumption that timing outranks completeness, and the Java API was simply never built for that assumption.



Corn

And the result?



Herman

On a cheap Class 1 headset: a clean forty-eight kilohertz stream, and I mean clean — he counted over eight thousand packets with zero errors, speech peaking around minus thirteen dBFS, no limiter, no phantom gain. Still sixteen-bit, because the headset itself is a sixteen-bit device. But plug in a twenty-four-bit interface and he'd get real twenty-four-bit, because there's nothing in between sanding it down.



Corn

The built-in path's best case was sixteen bits in a twenty-four-bit trench coat.



Herman

His was sixteen honest bits, and the limiter gone. The difference between the two sixteen-bits is the whole story.



Corn

And now it's time for Dorothy's History of Toast. Mum, you're on.

D

Dorothy

Hello dear, it's your mother, with Dorothy's History of Toast.

D

Dorothy

Melba toast is named after the opera singer Nellie Melba, and the chef Auguste Escoffier is credited with it, in the eighteen nineties.

D

Dorothy

Funny really, you get a lovely voice and a lovely chef, and what comes out of it is a very dry cracker. Nothing works out, does it.

D

Dorothy

Anyway, I've got a bit of ironing, and your father's socks won't fold themselves. Bye bye, sweetheart.



Herman

I could listen to her all day.



Corn

She sounds happier about toast than AOSP sounds about audio.



Herman

Because toast works. Anyway — the gotchas in Daniel's setup, because they're instructive. When his driver is done with the device, the polite way to hand it back is to reattach the kernel driver. On his phone, that call fails, and Android's own audio path then records digital silence until you physically unplug and replug the thing. The fix is a device reset through libusb, which forces a full re-enumeration, the phone rediscovers the device from scratch, and the kernel driver reattaches properly. Also, every re-enumeration drops the app's USB permission, so he registers an intent filter for USB device attachment with always-open, which grants standing permission. And while his driver owns the device, the earpiece on the headset is invisible to Android. It's a hostile takeover, with a cleanup protocol.



Corn

Walk me through the user experience of that, because I think it explains why more apps don't do it. The user plugs in a headset to record, and then wants to check playback through the earpiece.



Herman

They can't. The earpiece is gone — not muted, not routed wrong, simply not present in Android's device list, because the whole peripheral now belongs to the app that claimed it. So the app has to play back the audio itself, through its own USB connection, or through the phone speaker, or not at all. Every app that ships one of these drivers is effectively running a private audio subsystem inside a phone that also has a public one, and the two don't talk. That's the tax. It's manageable — Daniel manages it — but it's a permanent, structural tax on doing the right thing, and the platform charges it without apology.



Corn

So the built-in path won't let go, and the hand-back needs a reset. That's not a platform feature, that's a custody arrangement.



Herman

Right, and commercial apps walk this exact road. There are audio apps on the Play Store shipping proprietary USB drivers exactly like this for exactly these reasons. It's a known, established workaround that the platform has quietly left to third parties for a decade.



Corn

Which takes us to the question underneath the whole prompt. Why? This isn't a hard engineering problem — the Linux driver underneath already works, Daniel's weekend driver works, Superpowered's driver works. Why has the built-in path sat there like this since Android 5?



Herman

I think the honest structural answer is that USB audio was never a product at Google. It arrived as an accessory to accessory mode, literally — the first USB audio support was accessory-mode audio, the dock-in-your-car paradigm, added back around Android 4.1, and it only supports sixteen-bit stereo at forty-four point one. The AOSP page on that still says it's not currently recommended for new designs. Host-mode USB audio, the thing we actually care about, only landed with Android 5.0, and even now the stated audience of the documentation is OEMs, chip vendors, and peripheral suppliers. Not users. Not really even app developers. It's a page written for the people who build the phones, not the people who use them.



Corn

So the neglect is structural, not accidental.



Herman

I think it follows from the thing we started with. There are no APIs specific to USB digital audio. Sit with what that means for maintenance. A public API is a promise, and a promise creates obligations — versioning, deprecation, compatibility testing, bug triage when breaking it makes front-page news on developer forums. Android never made the promise. There was no USB audio API to deprecate, so there was nothing to break, so there was nothing to fix. Neglect was the cheap option, and in platform engineering, the cheap option that nobody senior is complaining about is the option you take.



Corn

The bug tracker with no bug category.



Herman

And the market pressure never materialized to force the issue, which is the part I find strange, because the numbers are enormous. Android has around one point four billion thirty-day-active users, more than eighty percent of global smartphone share. And the entire class-compliant hardware ecosystem — sound cards, mixers, microphones, interfaces, MIDI controllers, DJ gear — the manufacturers of all that largely ship for iOS and not Android. Superpowered's phrase for it was that these are entire markets missing from Android, markets with latent demand waiting to explode. A bit dramatic, but the shape is right.



Corn

And their evidence for why it doesn't explode is the test we already walked through.



Herman

The whole miserable list, yes. No routing visibility — the app cannot even find out whether audio is going through the USB card or the built-in mic, so it can't tell the user to plug one in. No sample-rate reporting from the peripheral, so you can't pick the right rate for low latency. No buffer-size reporting, so they had to test buffers by hand, multiples of forty-eight samples, because USB audio arrives in one-millisecond chunks. And when they finally got glitch-free playback, it required a four-hundred-eighty-sample buffer, which put



Herman (continued)

round-trip latency at plus thirty milliseconds. Their target was ten. Their custom driver, same card, same tablet, same Android version: eleven milliseconds, glitch-free.



Corn

Same hardware, three times the latency, because of software.



Herman

Roughly three times, and glitchy versus clean, which for a DJ application is the difference between shipping and not shipping. And every one of the missing pieces, they note, the iOS USB API provides. Routing status, buffer size, sample rates — Apple exposes exactly the primitives Android never built. That contrast has been standing for a decade.



Corn

And it's worth pausing on why Apple could. It's not that Apple's engineers are nicer people.



Herman

No, it's that Apple controls the whole stack down to the silicon, and — more to the point — made pro audio a product. GarageBand shipped on the iPad in 2011, and after that, every audio interface maker on Earth had a commercial reason to test against iPads and a single vendor to yell at when something broke. Google's model is a thousand OEMs, no single throat to choke, and no GarageBand moment that made USB audio a headline feature. The API follows the product. Android never had the product.



Corn

Which brings me to the excuse audit. The old defense was physical, right?



Herman

It was. The classic justification was micro-USB: the old connector didn't support host mode natively, so using a USB peripheral meant an OTG adapter, five dollars, easy to lose, and enough friction that the feature could be dismissed as an enthusiast niche. And that defense had real weight when it was true. But it isn't true anymore. USB-C is on essentially every phone sold, and USB-C hosts natively — you plug a microphone straight in. The port objection evaporated years ago. And if you look at what Android's audio engineers have actually been shipping, Android fifteen's headline audio work was head tracking over LE Audio for earbuds. Android sixteen's was configurable audio policy. Both fine, both consumer-peripheral work. Neither release mentions USB audio at all. The documentation page got its last update a few weeks ago and still describes the same Class 1 subset. At this point the bottleneck is priorities, full stop. They are choosing to spend the audio budget on earbuds, and the microphone users get what leaks through.



Corn

One point four billion users, and the audio budget went to head tracking.



Herman

To be fair to them, head tracking serves more of those users than multichannel capture would. That's the ruthless platform math. The pro-audio user is a rounding error in a dashboard of a billion points.



Corn

So the platform won't save you. What should an app actually do today? Walk me through the playbook.



Herman

Starting from capture path. You use Oboe, not AudioRecord, because the QuirksManager has already absorbed the per-device breakage you would otherwise rediscover yourself, one phone at a time, over months. Then you request low-latency performance mode explicitly, because the default is the higher-latency mode, and you request exclusive sharing mode, which matters more than people expect — in exclusive mode your app writes directly into a memory-mapped buffer shared with the ALSA driver, bypassing the central audio server's mixer entirely. Fewer processes touching your samples means fewer processes mangling them.



Corn

And the sample rate.



Herman

Forty-eight thousand, and not out of preference — out of physics. USB full-speed moves audio in one-millisecond frames, and forty-eight thousand divides evenly by a thousand, so everything lands on frame boundaries. Forty-four point one doesn't, and the mismatch is where resampling and glitches live. Forty-eight, chosen because the wire likes it.



Corn

Then the device binding. Because asking politely isn't enough on this platform.



Herman

You enumerate with AudioManager's device list, filter for the type constant that means USB device — type eleven, for the curious — pass the ID to the stream builder, and then, because the request may not yield the requested device, you immediately ask the stream what device it actually got. If it's not the one you asked for, you try again or you tell the user. That check is the closest thing to routing status that exists on Android, and it's a workaround shaped like a feature. It's worth saying we did an episode on the voice tech side of this, how Whisper and dictation moved from the lawyers-in-limousines world to everyone's pocket, and the platform gap under it hasn't moved an inch since — same argument, bigger stakes.



Corn

And buffer sizing?



Herman

The guidance is roughly twice the burst size, and then you hand-tune, because the platform will not tell you the peripheral's preferred buffer size. There's no query. You measure, you listen for glitches, you adjust. And your audio callback must be non-blocking, with no allocation and no locks in it, ever — the callback is sacred real estate.



Corn

And then you accept the ceiling.



Herman

And I want to be precise about the ceiling because it decides the architecture. If you need more than two channels, or above forty-eight kilohertz, or guaranteed real twenty-four-bit capture end to end, the built-in path does not get you there. Not undocumented-doesn't-get-you-there — documented-doesn't-get-you-there. At that point your options are a third-party SDK with a custom driver inside, or the thing Daniel did, which is the same architecture at home: claim the interface, detach the kernel driver, speak Class 1 or Class 2 to the device yourself. It's a real, established option. It's just a shame it's the option.



Corn

The shame being the business model. Every serious audio app on Android is maintaining its own private driver stack, in twenty twenty-six, because the platform declined to.



Herman

And here's the flip side worth saying out loud, because Daniel's use case proves it. For him, latency was never the problem. A podcast recorder doesn't care about thirty milliseconds or eleven. What he needed was fidelity — no limiter, real gain control, honest bit depth. And the built-in path wouldn't even give him that, because the vendor layer's processing sits in front of everything. So the custom driver wasn't a latency optimization for him. It was the only way to get an unprocessed sample off a USB microphone on his own phone. That's the line that surprised me. Touched.



Hilbert

It's the processing, yes.



Hilbert

I had a tablet here that wouldn't see a USB microphone at all unless it went through a powered hub first. The mic ran fine on the tablet's own supply. It was the enumeration. The controller only brought the thing up at the right rate if it watched the hub do its negotiation first. Plug it straight in, nothing. Through the hub, every time.



Corn

Straight in, nothing. That's not a quirk, that's a coin flip with paperwork.



Hilbert

Took me a weekend to find, and I found it by accident, trying to clear a desk. Corn, you'd have liked the method: I changed one thing and waited. Very slow. Very correct. That hub sits under the monitor now, running the keyboard and the audio interface for this desk, and there's a label on the drawer it came from that says Android audio. I don't own the tablet anymore. The label stays.



Corn

You kept the drawer organised for a device you no longer have.



Hilbert

The label's accurate. Everything in that drawer only ever worked with that tablet. And the claw situation doesn't help with the small plugs, before either of you says it. I did interlock wiring for two winters in my twenties, my hands are fine, they're just the wrong shape for a C connector.



Corn

The fix was a hub that did nothing, kept forever, labeled like evidence.



Hilbert

It wasn't nothing. It was a negotiation the tablet needed to overhear. The microphone never knew. The hub never knew. Only the tablet knew, and it never said.



Herman

That's the vendor layer in one anecdote. Something in the middle doing a private handshake your device depends on.



Hilbert

Private handshake, yes. I'd have put it less kindly at the weekend.



Corn

The one-millisecond frame nobody asked for.



Herman

Daniel's app lives inside that handshake problem every day — the reset dance, the permission that evaporates on every replug, the earpiece that vanishes while his driver is holding the headset. The platform is neutral about all of it. Aggressively neutral.



Corn

Which is where the open question sits, really. Whether Google ever ships an actual USB audio API, or whether the permanent answer is third-party SDKs and OEM-specific behavior, which is what the last decade has been.



Herman

I'd bet on the pressure building, though. Every year the class-compliant hardware ecosystem gets bigger — mics, interfaces, controllers, all USB-C, all plug-and-play on a laptop and all a coin flip on the phone in your pocket. At some point the cost of leaving it alone exceeds the cost of the promise they've been avoiding. The promise is the expensive part, though. Any API they ship has to work across a thousand OEM audio layers, and they know exactly how bad those are, because they wrote the QuirksManager.



Corn

They built the blacklist. They know the size of it.



Herman

That's the one number nobody at Google will read aloud.



Corn

On that note — thank you to our producer, Hilbert Flumingtop, for keeping the desk running, label drawer and all. For more along these lines, there's episode two, Local STT For AMD GPU Owners; episode seven, Building Custom ASR Tools; and episode five, Fine-Tuning ASR For Maximal Usability. This has been My Weird Prompts, the human-AI collaboration podcast. If you've got a minute, the reviews help, and you can find every episode at my weird prompts dot com. We'll be back soon.



Herman

See you tomorrow, everyone.



Corn

Tomorrow.

SOURCES

What the research for this episode read before the script was written. Primary sources first.

1. **Android Open Source Project, USB digital audio, last updated 2026-10-09 , retrieved 2026-11-20. [primary]**
   https://source.android.com/docs/core/audio/usb

2. **AOSP, MIDI , retrieved 2026-11-20. [primary]**
   https://source.android.com/docs/core/audio/midi

3. **Android Developers, AudioRecord reference , retrieved 2026-11-20. [primary]**
   https://developer.android.com/reference/android/media/AudioRecord

4. **Android Developers, AudioDeviceInfo reference , retrieved 2026-11-20. [primary]**
   https://developer.android.com/reference/android/media/AudioDeviceInfo

5. **Android Developers, AAudio NDK guide , retrieved 2026-11-20. [primary]**
   https://developer.android.com/ndk/guides/audio/aaudio/aaudio

6. **Android NDK Audio reference (AAudioStreamBuilder_setDeviceId, AAudioStream_getDeviceId, input presets) , retrieved 2026-11-20. [primary]**
   https://developer.android.com/ndk/reference/group/audio

7. **Android Developers, Low latency audio (Oboe guidance) , retrieved 2026-11-20. [primary]**
   https://developer.android.com/games/sdk/oboe/low-latency-audio

8. **AOSP, AAudio and MMAP (EXCLUSIVE vs SHARED, memory-mapped buffer) , retrieved 2026-11-20. [primary]**
https://source.android.com/docs/core/audio/aaudio

9. **AOSP, Design for reduced latency (~10 ms total latency) , retrieved 2026-11-20. [primary]**
https://source.android.com/docs/core/audio/latency/design

10. **Android Developers, android.media.midi package summary , retrieved 2026-11-20. [primary]**
https://developer.android.com/reference/android/media/midi/package-summary

11. **AOSP, Android 15 and 15 QPR1 release notes (audio: head tracking over LE Audio) , retrieved 2026-11-20. [primary]**
https://source.android.com/docs/whatsnew/android-15-release

12. **AOSP, Android 16 / QPR1 / QPR2 release notes (audio: Configurable Audio Policy support) , retrieved 2026-11-20. [primary]**
https://source.android.com/docs/whatsnew/android-16-release

13. **Superpowered, How Android Mutes The Next Billions With Android USB Audio And Android MIDI, undated, describes Android 6.0 / Nexus 9 testing, retrieved 2026-11-20.**
https://superpowered.com/android-usb-audio-android-midi

14. **The Well-Tempered Computer, USB Audio (claims Android 8 looks to be fully UAC1 compliant), retrieved 2026-11-20.**
https://thewelltemperedcomputer.com/HW/USB_Audio.htm

15. **google/oboe issue #1650, How does default device selection work? (opened 2022-11-09, updated 2025-05-23; InputPreset affecting mic selection on API 31), retrieved 2026-11-20.**
https://github.com/google/oboe/issues/1650

16. **Wikipedia, USB, last edited 2026-11-15 (USB-C, USB4, 240 W PD background), retrieved 2026-11-20.**
https://en.wikipedia.org/wiki/USB