VTX USB 1: USB devices in a session

Nov 15, 2026 · VTX 1 · extension · draft d01 · MIT licence · vtx.foundation/spec/v1/usb-1/

§1 Status and scope

USB 1 is an extension of VTX 1, draft d01, in conformance profile 3. It lets the user plug a USB device of their own device into a session, as RDP's RemoteFX USB redirection and USB/IP do: the device disappears from the user's machine, appears on the session host as if it were plugged in there, and the host's own drivers and programs use it. It is the catch-all for devices nothing else covers: serial converters on lab balances and machine controllers, signature pads, licence dongles, 3D mice, measuring instruments, programmers.

It works at the level of USB requests (URBs). The client sends the device's descriptors when the user shares it; the server attaches the device to a virtual host controller on the host; every request a driver on the host makes becomes a SUBMIT on the channel, and the client performs it on the real device and answers with COMPLETE. Neither side needs to understand the device: what works on a local port works in the session, given enough bandwidth and not too much delay.

That generality is also the risk, so USB 1 is careful about which devices it carries:

- Devices another extension serves better are never passed through: keyboards and mice, storage, printers, smart card readers, cameras and audio (Section 6). Those extensions keep the user's protections, which raw USB would bypass.
- A session cannot reflash a device's firmware unless the user allowed it for that device (Section 7).
- The user picks every device, the trusted zone shows each one, and a device returns to the user's machine reset (Section 9).

Out of scope. USB 1 carries no hubs and does not pass through a whole port: a device is shared, not a socket. Devices of the classes in Section 6 cannot be shared with USB 1 at all in this draft, even when the organization would allow it. Isochronous devices work, but they need a fast, steady connection; audio and video devices go through Media Devices 1 instead.

§2 Capability and channel

The client offers `redirect.usb` in HELLO, under the `redirect.` prefix of the redirections; the server grants it with `{}`. After READY the client opens one dynamic channel of type `usb` (an odd ID). An open channel shares nothing yet: devices enter the session only when the user shares them, one by one. The server never opens the channel, and a connection has at most one.

```
cap-usb = {
  ? "max": 1..16,             ; client: most devices plugged at once (default 4)
}
```

Limit	Value	Why
Devices plugged at once	`max`, 1 to 16, default 4	Each one is a device the user is not using locally
URBs waiting per device	64	Enough for streaming bulk and interrupt reads; the host's drivers queue more themselves
Bytes in one URB	1,048,576	Larger transfers are split by the host's drivers, as on a local controller
Packets in one isochronous URB	1,024	As USB/IP

Closing the channel (v1's CLOSE) unplugs every device at once; each returns to the user's machine as Section 7 describes. No permission is added: the user shares devices through the trusted zone, and the server asks for one with WANT (Section 3).

§3 Sharing a device

The user shares a device from the trusted zone's device menu, or when a program in the session asks for one. Either way the client then plugs it in with PLUG, which carries everything the host needs to enumerate the device without asking it first.

```
usb-want = {                                          ; 0x10 WANT, S->C: ask the user
to share a device
  "want": uint,
  "filters": [1*8 usb-filter],
  "detail": tstr .size (1..200),  ; why, shown in the chooser
}
usb-filter = {
  ? "vendor": 0..65535, ? "product": 0..65535,
  ? "class": 0..255, ? "subclass": 0..255, ? "protocol": 0..255,
}
usb-plug = {                                          ; 0x11 PLUG, C->S: a device the
user shared
  "device": usb-device-id,    ; chosen by the client; never reused in a connection
  ? "want": uint,             ; the WANT this answers
  "name": tstr .size (1..128),
  "vendor": 0..65535,
  "product": 0..65535,
  "speed": "low" / "full" / "high" / "super" / "super-plus",
  "descriptor": bstr .size 18,  ; the device descriptor
  "configurations": [1*8 bstr],  ; each configuration descriptor and what follows it
  "config": 0..255,           ; the active configuration; 0 when unconfigured
  ? "firmware": true,         ; the user lets the session update this device's firmware
(Section 7)
}
usb-declined = { "want": uint, "reason": "user" / "none" / "blocked" }   ; 0x13
DECLINED, C->S
usb-unplug = { "device": usb-device-id, "reason": "user" / "removed" / "reset" }   ;
0x12 UNPLUG, C->S
```

Asking. WANT works like WebUSB's device chooser: the client shows the user, in its trusted zone, the devices that match any of the filters and pass the class rules, with the `detail` the server gave. A filter matches when every field it names agrees: `vendor` and `product` with the device, `class`, `subclass` and `protocol` with the device or with one of its interfaces. The user picks one device, and the PLUG carries the WANT's number; or the client answers DECLINED: `user` when the user closed the chooser, `none` when no device matches, `blocked` when only devices the class rules keep out match. Every WANT gets exactly one of the two. The server learns nothing about the devices the user did not pick.

The descriptors. PLUG carries the device descriptor and every configuration descriptor with the interface, endpoint and class descriptors that follow it, as USB 2.0 Chapter 9 defines them. They must be well formed: a device descriptor of 18 bytes whose vendor and product match the PLUG's, as many configurations as it announces, each one exactly `wTotalLength` long with unique values, and an active `config` that is one of them or 0. The host can still read any descriptor from the device itself, including its string descriptors.

Unplugging. UNPLUG takes a device out of the session: `user` when the user stopped sharing it, `removed` when it was pulled out, `reset` when it came back as a different device (Section 7). URBs still waiting on it end with it, without a COMPLETE. Device ids are never reused in a connection, so a late answer can never be mistaken for one from a later device.

§4 At a glance

> **Diagram: a shared device · from the host's driver to the device on the laptop.** Drawn in the PDF version, `17-usb-1.pdf`.

Marcel picks the balance's serial cable in the trusted zone's chooser. The client detaches it from the laptop's own driver and plugs it into the session; the host's virtual host controller shows it on a port, and the host's ordinary FTDI driver drives it. Every request that driver makes travels down as SUBMIT and comes back as COMPLETE, and the client checks each device against the rules of Sections 6 and 7 before it ever reaches the host.

§5 Transfers

Each URB a driver on the host submits becomes one SUBMIT; the client performs it on the device and answers with exactly one COMPLETE. UNLINK cancels a URB that is still waiting.

```
usb-submit = {                                          ; 0x14 SUBMIT, S->C: one USB
request
  "device": usb-device-id,
  "urb": usb-urb,
  "type": "control" / "bulk" / "interrupt" / "isochronous",
  "endpoint": 0..15,
  "dir": "in" / "out",
  ? "setup": bstr .size 8,    ; control only: the SETUP packet
  "length": 0..1048576,       ; in: most bytes to read; out: the bytes in data
  ? "data": bstr .size (0..1048576),  ; out only
  ? "packets": [1*1024 (0..49152)],   ; isochronous only: each packet's length
  ? "exact": true,            ; in: a shorter answer is an error ("short")
  ? "zero": true,             ; out: end with a zero-length packet
}
usb-unlink = { "device": usb-device-id, "urb": usb-urb }   ; 0x15 UNLINK, S->C: cancel
a SUBMIT
usb-complete = {                                        ; 0x16 COMPLETE, C->S
  "device": usb-device-id,
  "urb": usb-urb,
  "status": "ok" / "stall" / "short" / "overflow" / "canceled" / "error" / "denied",
  ? "data": bstr,             ; in: what the device sent
  ? "actual": uint,           ; out: bytes the device took
  ? "packets": [1*1024 usb-packet],   ; isochronous: each packet's result
}
```

What a SUBMIT must say. URB numbers grow on each device, and at most 64 wait on one device at a time. Endpoint 0 carries control transfers; any other endpoint must exist in the active configuration with the transfer's type and direction. A control transfer carries its 8-byte SETUP packet, and the packet must agree with the transfer: its direction bit with `dir`, its wLength with `length`. Out transfers carry exactly `length` bytes of data, or none for a transfer of no bytes; isochronous ones list their packets, which add up to `length`. The server tracks SET_CONFIGURATION from its completion, so the endpoints it may use follow the configuration the device has.

What a COMPLETE must say. An in transfer returns `data`, never more than `length`; an out transfer returns `actual`, the bytes the device took. With `exact`, a shorter read is `short`, not `ok`. Bulk and interrupt reads may wait as long as the device has nothing to say: a serial converter's read waits for the next character, a pen's for the next movement.

Status	Meaning	Linux URB status on the host
`ok`	Done	0
`stall`	The endpoint stalled, or the device refused a control request	-EPIPE
`short`	An exact read got fewer bytes	-EREMOTEIO
`overflow`	The device sent more than the buffer holds	-EOVERFLOW
`canceled`	Ended by UNLINK	-ECONNRESET
`error`	A transfer error on the bus	-EPROTO
`denied`	The client's rules refused it (Section 7)	-EPERM

Canceling. UNLINK names a URB the server submitted. If it is still waiting, the client cancels it on the device and completes it with `canceled`; if its COMPLETE is already on the way, that COMPLETE stands and nothing more is sent. A URB is completed as `canceled` only after an UNLINK for it.

Delay. Every control request makes a round trip, so enumeration and drivers that poll take longer than on a local port; drivers already allow for slow devices, and most devices in Section 1 send little data. Isochronous transfers need the packets on time, so devices that use them only work well on a fast, steady connection.

§6 Devices that are not shared

Some devices must never be handed to a session as raw USB, because another part of VTX serves them with protections raw USB would bypass, or because the host would gain control it should not have. The client does not offer them in its chooser or its device menu, and never plugs them:

Device	USB class	Why not	Use instead
Keyboards and mice	HID (03) boot keyboard or mouse, or any HID that reports keys or pointer movement	Every keystroke, including those typed into the trusted zone, would go to the host, and the user could no longer use them locally	The input channel (v1)
Security keys for sign-in	HID with the FIDO usage page	A session holding the key could sign in to any website as the user	A later extension for WebAuthn
Storage	Mass storage (08)	Raw blocks let the host read deleted files and write anything, even corrupt the file system	Drives 1, folder by folder, read-only if wanted
Printers	Printer (07)	Needs a driver on the host	Printers 1
Smart card readers	Smart card (0B)	PINs would pass through the host, and payment cards would be reachable	Smart Cards 1
Microphones, speakers, cameras	Audio (01), video (0E)	Raw isochronous audio and video over a network breaks up, and echo cancellation needs the device	Media Devices 1
Hubs	Hub (09)	Devices are shared, not ports	Share each device behind the hub
Bluetooth and other radios	Wireless controller (E0)	The host would control the user's radio and could pair with devices near the user	None

Every interface counts. The rule applies to the device class and to every interface of every configuration. A composite device with one excluded interface is not shared at all: a YubiKey, which is a keyboard, a FIDO key and a smart card reader at once, stays on the laptop and reaches the session through Smart Cards 1. Keyboards and FIDO keys that only show themselves in the HID report descriptor are found by the client reading it before it offers the device.

What remains. Everything else may be shared: vendor-specific devices (class FF) such as serial converters and dongles, communication devices such as modems and CDC serial ports, other HID devices such as signature pads, 3D mice, uninterruptible power supplies and measuring instruments, and test and measurement devices. When a WANT matches only excluded devices, the client answers `blocked`, so the program can tell the user which extension to use instead.

This draft has no way for an organization to allow an excluded device; if a real need comes up, a later draft can add a policy list.

§7 Firmware, resets and mode switches

A device the session can reflash can come back to the user's machine as something else: a cable that types commands, a dongle that lies. USB 1 closes the standard way in and is careful about devices that change identity.

Firmware updates. USB's Device Firmware Upgrade class (DFU, interface class FE, subclass 01) is how most devices take new firmware. The client answers every class request to a DFU interface (DETACH, DNLOAD and the rest) with `denied`, unless the PLUG carried `firmware`. The user sets that per device in the client's settings, for the rare update that has to run in the session; the client never sets it on its own. Vendor-specific update commands cannot be recognized in general; a device that takes its firmware that way can be updated by a session that has it, as on any machine its owner plugs it into.

Requests that change the device. Some control requests change the device's state rather than ask it something. The client performs those through its own USB stack instead of sending them raw, as USB/IP does, so the laptop's stack keeps track: SET_CONFIGURATION, SET_INTERFACE, CLEAR_FEATURE for a halted endpoint, and a port reset, which the server sends as SET_FEATURE(PORT_RESET) addressed to the port (bmRequestType 23).

Mode switches. After a reset, or on its own, a device can come back with other descriptors: a modem leaving its storage mode, a device starting its bootloader, an update that changed what it reports. The client then sends UNPLUG with `reset` and looks at the new device as it would at any other. It plugs it in again under a new id if it passes the class rules, so a mode switch the host's software asked for keeps working; a device that now comes up in DFU mode is plugged in again only if the user allowed firmware updates for it. A device that comes back as a keyboard, storage or anything else Section 6 excludes stays on the laptop, and the trusted zone tells the user what happened.

Back to the laptop. When a device leaves the session, by UNPLUG, by closing the channel or because the connection dropped, the client resets it before the laptop's own driver takes it back. Nothing the session left behind (a configuration, an unlocked dongle, a half-sent command) carries over.

§8 The session host and the application interface

On a Linux host the server attaches every plugged device to `vhci-hcd`, the virtual host controller that comes with the kernel's USB/IP support. The kernel enumerates the device as if it were on a local port and binds its ordinary driver (`ftdi_sio`, `cdc_acm`, `usbhid` and so on); the server carries the URBs between `vhci-hcd` and the channel. Programs in the session then open `/dev/ttyUSB0`, a `hidraw` node or the device through libusb as they would on any machine. On a Windows host the server presents the devices on a virtual USB bus, as RemoteFX USB redirection does.

One session's devices. A virtual host controller belongs to the host's kernel, not to one session. On a host that runs several sessions, the server must give a plugged device's nodes (`/dev/bus/usb/…`, `/dev/ttyUSB*`, `/dev/hidraw*` and the like) to the session's user alone, through udev rules, and keep other sessions away from them. Some drivers create resources the whole host sees, such as the network interface of a USB modem. Organizations that need strict separation should allow USB 1 only on hosts that serve one user.

Application events. The server tells the session manager about plugged devices with the `usb` event:

`action`	When	Fields
`added`	A device was plugged into the session	`id`, `name`, `vendor` and `product` in lowercase hex, `busid` (where the virtual host controller put it)
`removed`	It left the session	`id`, `reason` as in UNPLUG

```
{"kind":"usb","action":"added","id":1,"name":"FT232R USB
UART","vendor":"0403","product":"6001","busid":"3-1"}
{"kind":"usb","action":"removed","id":1,"reason":"removed"}
```

USB 1 adds no scene op and no application line: programs reach the devices through the host's own USB stack.

§9 Security and privacy

A shared USB device belongs to the session for as long as it is shared: the session can do anything with it that the device allows. USB 1 limits which devices that can be, and makes sure the user always knows:

- **The user decides.** Only devices the user picks are plugged in. A WANT tells the server nothing about the devices the user did not pick. The trusted zone lists every shared device and unplugs it with one click; while shared, the device is gone from the laptop, so the user notices.
- **No input, storage or keys.** Keyboards, mice, storage, smart card readers, FIDO keys, cameras, microphones and radios are never passed through (Section 6), so the protections of the input channel, Drives 1, Smart Cards 1 and Media Devices 1 cannot be bypassed.
- **No new firmware unasked.** A session cannot use DFU to reflash a device unless the user allowed it for that device, and a device that comes back as something excluded stays on the laptop (Section 7).
- **A clean return.** A device is reset before the laptop's driver takes it back.
- **Locked device.** Shared devices stay shared while the user's device is locked, because dongles and instruments break when their transfers stall. The user can set the client to unplug every device when the device locks.
- **One session's devices.** On hosts with several sessions, a device's nodes belong to the session's user alone (Section 8).

What the session learns. Device ids are made up for each connection, but the descriptors are the device's own, so a session can read its vendor, product and serial number, as any machine it is plugged into can.

What remains. A session can send a shared device any vendor command it understands, including ones that change its settings, and a device with a weak firmware can be attacked that way. Share devices only with sessions you would plug them into directly.

§10 Definitions and checks

The definitions live in the bundle under `ext/usb-1/`:

File	What it holds
`vtx-usb-1.cddl`	The `redirect.usb` capability and the usb channel's frames
`vtx-app-events-usb-1.schema.json`	Server lines to an application, with the `usb` event; generated
`examples/valid/trace-lab-quic.jsonl`	The trace below, 55 frames
`examples/valid/app-usb.jsonl`	Four `usb` events
`examples/invalid/plug-rules.jsonl`	Six PLUGs that pass the CDDL but break a rule
`examples/invalid/traces.jsonl`	The trace broken 22 ways, one at a time
`examples/invalid/app-schema.jsonl`	Two events the schema rejects

`tools/gen_ext_usb.py` writes the schema, built on the Smart Cards 1 generator, and `tools/make_usb.py` writes the examples. The descriptors are those of an FT232R serial converter (0403:6001) and of a made-up signature pad under the pid.codes test id (1209:0001); pen reports are made-up bytes from a fixed seed, so every run writes the same files.

The trace. A lab program in Marcel's hosted desktop asks for the balance on his desk, and he shares its USB serial cable. The host reads the descriptors and the product name, sets the configuration and sets the line up (9600 baud, 8N1) with the converter's vendor requests; the program sends `SI` and reads back `S S 12.345 g`. A delivery program asks for the signature pad; the host reads its report descriptor (the pad stalls SET_IDLE, as many devices do), two pen reports arrive, the program's attempt to put the pad into firmware update mode is `denied`, and it cancels its last read. An import tool asks for a USB stick, and the client answers `blocked`. Marcel stops sharing the pad, the balance's cable is pulled, and he closes the channel.

The validator's Section 25 runs 57 checks, and every one passes:

Part	Checks
The schema, the CDDL alone and with every earlier extension, no rule defined twice, earlier schemas rejecting the `usb` event	5
CDDL samples, 7 valid and 10 malformed, and the 6 broken PLUGs	23
The trace, and each broken trace failing for its labelled reason	23
Application events, 4 valid and 2 rejected	6

Label	What a broken trace does
`caps`, `direction`, `channel`	A channel the server did not grant; the server opening it; the client sending WANT; a second channel
`plug`, `class`	A vendor that differs from the descriptor; a device id used twice; a boot keyboard; a USB stick
`want`	A PLUG matching none of the WANT's filters; a WANT answered twice
`device`, `urb`, `limit`	A SUBMIT for a device never plugged; a SUBMIT after UNPLUG; a urb number used twice; 65 URBs waiting; a second device over `max`

Label	What a broken trace does
`endpoint`, `setup`, `cddl`	A bulk write to an endpoint the configuration lacks; a control read whose SETUP says host to device; a SUBMIT for 2,000,000 bytes
`result`, `unlink`, `firmware`	A read answered with more than asked; a URB completed twice; a URB canceled without UNLINK; a DFU detach completed without `firmware`

Section 17, the extensions together and the registry, gained three checks for the new channel type. The whole bundle runs 2,543 checks, all passing.

§11 Wire summary

Everything USB 1 adds to the wire is on its own channel type, `usb`, which only the client opens (an odd ID), at most once per connection:

Type	Frame	Direction	Purpose
0x10	WANT	S→C	Ask the user to share a device matching filters
0x11	PLUG	C→S	A device the user shared, with its descriptors
0x12	UNPLUG	C→S	A device leaves the session
0x13	DECLINED	C→S	No device for a WANT
0x14	SUBMIT	S→C	One USB request
0x15	UNLINK	S→C	Cancel a waiting request
0x16	COMPLETE	C→S	The answer to a request

Besides the channel, it adds the `redirect.usb` capability. It adds no permission, no frame on v1's well-known channels, no close code and no datagram.