ChatGPT

Software FIDO2/CTAP HID Authenticator Requirements

FIDO2 HID Specifications and Descriptor

HID Usage and Descriptor: FIDO2 authenticators over USB HID use a unique **HID Usage Page 0xF1D0** (assigned to FIDO Alliance) with a **Usage ID 0x01** for the top-level authenticator collection [1] [2]. The HID report descriptor defines two "raw" HID reports (one IN and one OUT) that map directly to the interrupt endpoints [3] [4]. A minimal descriptor (in pseudo-code) is:

- **Usage Page:** FIDO_USAGE_PAGE (0xF1D0) – identifies device as FIDO/U2F/CTAP authenticator [1]
- **Usage:** FIDO_USAGE_CTAPHID (0x01) – top-level application collection [1] [2]
- **Collection:** Application (contains two reports)
- **Usage:** FIDO_USAGE_DATA_IN (0x20) – Input report (from authenticator to host) [2]
  - Logical Min/Max 0 to 0xFF (bytes), Report Size 8 bits, Report Count = *N* (size in bytes) [5]
  - Input(Data,Var,Abs) – defines an IN report of N bytes
- **Usage:** FIDO_USAGE_DATA_OUT (0x21) – Output report (host to authenticator) [2]
  - Logical Min/Max 0 to 0xFF, Report Size 8, Report Count = *M* bytes
  - Output(Data,Var,Abs) – defines an OUT report of M bytes
- **End Collection**

For a full-speed USB device, the reference values are **N = M = 64 bytes** [6] [7]. In practice, most FIDO2 keys use 64-byte HID reports (the max for USB full-speed) so each HID report is 64 bytes long [8] [6]. (High-speed USB could allow larger reports, but 64 is typical and maximally compatible [9] [10].) The report counts in the descriptor should match the endpoint packet size used [11] [7]. This yields a maximum CTAP message size of 7609 bytes when using 64-byte packets (1 initial 64-byte frame minus 7 bytes of overhead, plus up to 128 continuation frames minus 5 bytes each) [12] [13].

Report IDs and Features: Notably, **no Report ID bytes** are used – the descriptor above defines one input report and one output report without specifying a "Report ID" tag. The CTAP HID spec implies "two raw reports" with no numbered report IDs [14]. Using a report ID (numbered reports) is neither required nor recommended; in fact, adding a report ID or extra feature reports can break compatibility with host APIs [14] [15]. (Windows and many client libraries will not recognize the device as FIDO if reports are numbered or if an unexpected feature report is present [15].) Therefore, a **Feature report** is **not required** for a FIDO2 HID device to enumerate, and most authenticators omit feature reports entirely. The HID driver will identify the device purely by the usage page/ID in the descriptor [16]. In summary: **only two un-numbered reports (IN & OUT) on usage page 0xF1D0** should be present for a standard CTAP HID device [14] [17].

Device Discovery: Host software (OS or libraries) discover FIDO devices by scanning all HID devices' descriptors for the FIDO usage page **0xF1D0** and usage **0x01** [1]. Any HID that matches is treated as a CTAP device. For example, libfido2 opens each `/dev/hidraw` and checks the descriptor's usage page, flagging it as FIDO if `usage_page == 0xf1d0` (it assumes usage 0x01) [1] [18]. The HID descriptor's presence is

also what causes operating systems to load the generic HID driver and (on Windows) label the device as "HID-compliant FIDO" in Device Manager [16]. In short, **the correct HID usage page (0xF1D0) and usage ID (0x01)** in the report descriptor are the key requirements for the device to be recognized as a FIDO/U2F/CTAP authenticator by the system [19].

CTAPHID Protocol and Framing

Packet Framing: The CTAP HID protocol (often called **CTAPHID**) encapsulates CTAP1/U2F and CTAP2 commands into fixed-size HID packets [20] [3]. Each packet is 64 bytes (for full-speed devices) and consists of: a 4-byte **Channel ID**, a 1-byte **Command** (or sequence index), 2 bytes for **payload length**, and a payload fragment [21]. The first packet of a message is an "**init frame**" (command byte with MSB=1) and continuation packets (if needed) use a sequence index (MSB=0) [22] [23]. For example, in an **initialization frame**:

- **Bytes 0–3:** Channel ID (32-bit big-endian) [21]. `0xFFFFFFFF` is a special **broadcast CID** used by the host to request a new channel [24] [25].
- **Byte 4:** Command code (bit 7 = 1 for init frames). For instance, 0x86 = CTAPHID_INIT, 0x83 = CTAPHID_MSG, 0x90 = CTAPHID_CBOR, etc. The device will see this byte with the high bit set and decode the command (the command's "base value" is `byte4 & 0x7F`) [22] [26].
- **Byte 5:** BCNTH – High byte of payload length
- **Byte 6:** BCNTL – Low byte of payload length (so length is a 16-bit big-endian integer) [21]. This length is the total message payload size in bytes.
- **Bytes 7–(n):** Payload data (part of the CTAP message, up to 57 bytes in the first frame when using 64-byte packets) [27]. Unused bytes in the 64-byte packet are padded with 0x00 [28].

If the payload is larger than fits in one packet, the host sends follow-up **continuation frames**: - **Bytes 0–3:** Same Channel ID
- **Byte 4:** Sequence index (0x00 for the first continuation, 0x01 for the next, etc., with bit7=0) [29].
- **Bytes 5–(m):** Next chunk of payload (up to 59 bytes per continuation frame) [30].

All packets are **64 bytes** on the wire (the device must always send/receive the full report size) even if the actual payload is shorter [28]. Multi-byte values in the CTAPHID protocol (like length and channel) are conveyed in big-endian/network order (the spec shows BCNTH/BCNTL and Channel ID in that order) [21]. The device should parse and construct these frames accordingly. The entire message (possibly spanning an init and multiple continuation packets) carries one higher-level CTAP command.

CTAPHID Commands: A CTAPHID authenticator **must** support a core set of HID-layer commands [31]. Mandatory commands (with their 7-bit code in hex) include:

- **CTAPHID_PING** (0x01) – Echoes data for testing latency [32]
- **CTAPHID_MSG** (0x03) – Handles a U2F (CTAP1) message frame [33]. This encapsulates a U2F v1.1 request/response (for backward compatibility with U2F APIs) [34].
- **CTAPHID_LOCK** (0x04) – (Optional) Lock channel for exclusive use (not often used in practice) [35].
- **CTAPHID_INIT** (0x06) – Initializes a channel or allocates a new one (if sent to broadcast CID) [36] [24]. The host sends an 8-byte nonce in the INIT request; the authenticator must respond with the **same nonce** and its info.
- **CTAPHID_WINK** (0x08) – (Optional) Causes device identification blink/beep (if supported) [35].

- **CTAPHID_CBOR** (0x10) – Transport for CTAP2 (FIDO2) messages encoded in CBOR [37]. All FIDO2 operations (makeCredential, getAssertion, etc.) are carried in this command's payload.
- **CTAPHID_CANCEL** (0x11) – Cancels the current transaction on a given channel (e.g. user clicked cancel in the browser) [38]. The host can send this to signal the authenticator to abort processing.
- **CTAPHID_KEEPALIVE** (0x3B) – Sent **by the authenticator** to the host to indicate it is still processing a long operation [39]. The payload is a single byte status: typically `0x01` = **PROCESSING**, or `0x02` = **UP_NEEDED** (user presence needed) [40] [41]. The device should send KEEPALIVE messages at least every 100ms (spec recommends up to ~500ms) while waiting for user interaction or lengthy cryptographic operations [39] [42]. This prevents host timeouts.
- **CTAPHID_ERROR** (0x3F) – Error response from device if a command is invalid or cannot be processed [43]. The first byte of the payload is an error code (e.g., 0x01 = INVALID_CMD, 0x02 = INVALID_PAR, etc.) [44].

All CTAPHID command codes are actually sent with the high bit set in the first frame (as mentioned), but documentation often refers to them by the lower 7-bit value (shown above) [26] [45]. The device's firmware needs to implement handlers for these commands. **CTAPHID_INIT** in particular is critical: on receiving it (often as the very first message from a host on the broadcast CID), the device must reset any ongoing transaction and reply with an **INIT response**. The **INIT response** is 17 bytes of data: it echoes the 8-byte nonce, plus: a newly allocated 4-byte Channel ID (if a new channel was requested) or the existing CID, a 1-byte **Protocol Version** (should be 2 for CTAPHID v2 [46]), 3 bytes of device version (major, minor, build – can be device-defined), and a 1-byte **Capabilities** bit mask [47] [48]. For example, a CTAPHID_INIT response data layout is: nonce[8] || CID[4] || 0x02 (interface ver) || verMajor || verMinor || verBuild || capFlags [47] [48]. The **Channel ID** should be a non-zero random 32-bit number (unique per session) that the authenticator tracks; the host will use that ID in subsequent packets so multiple client applications can be multiplexed over one HID interface [49] [24].

Capabilities Flags: In the CTAPHID_INIT response, the last byte is a bitfield of device capabilities [48] [50]:
- Bit 0x01 – **WINK** capability (set if device supports the CTAPHID_WINK command) [51].
- Bit 0x04 – **CBOR** capability (set if device supports CTAPHID_CBOR for CTAP2, which a FIDO2 authenticator should) [51].
- Bit 0x08 – **NMSG** (No MSG) – set this **if the device does *NOT* implement CTAPHID_MSG** (i.e. it does not support the legacy U2F "MSG" command) [51].

Typically, a pure CTAP2 device would set `capFlags = 0x04` (CBOR supported, no wink, and it *does* implement MSG unless it explicitly opts out). If you choose not to support U2F at all, include the NMSG flag (0x08) to inform hosts that CTAPHID_MSG is not available [52]. Setting 0x01 (WINK) is optional; many keys set it to 0 and do nothing for wink, but some might implement a software-visible blink.

Transaction Flow and Timeouts: The CTAPHID layer allows concurrent "transactions" on different channels – each host application (browser, SSH agent, etc.) should allocate a channel via INIT and then send commands on its channel [53] [24]. The authenticator must manage state per channel (e.g., track if a request is in progress). It should handle interleaving safely by queueing or returning BUSY errors (there is an error code for CHANNEL_BUSY 0x06) if a second command arrives on the same channel before the first is completed [54]. However, well-behaved clients will generally not violate this. If a command is taking long, the device sends KEEPALIVE as noted. The host might send CTAPHID_CANCEL on the same CID if the user aborts. The spec defines a general **message timeout** of 3 seconds for transactions (and a longer timeout at application level for user presence) [55] [56]. The device can use this as a guideline: e.g., if no continuation

packet arrives within a certain time, abort the transaction (and possibly return ERR_MSG_TIMEOUT 0x05) [44].

Finally, once a CTAP command (e.g., authenticatorGetInfo, MakeCredential, etc.) is executed, the device sends back the response encapsulated in the CTAPHID protocol. For CTAPHID_CBOR, the response will carry a CTAP status (0x00 for success or an error code) and CBOR-encoded data. For CTAPHID_MSG (U2F), the response carries the U2F status byte (e.g. 0x90 for NO_ERROR) and response data [57] [58]. At the CTAPHID level, the host doesn't interpret the payload; it just shuttles it to the higher layers (WebAuthn client or U2F API) which decode the CBOR or U2F data. Ensuring correct byte ordering and frame assembly at this layer is crucial so that higher-level protocol messages are reconstructed accurately.

Operating System Integration

Although CTAP over HID is standardized, each operating system has its own interface for HID devices. Below we summarize requirements for Linux (using `/dev/uhid` to emulate a device) and notes on Windows/macOS behavior:

Linux UHID Requirements (User-space HID)

On Linux, a user-mode program can create a virtual HID device using the **UHID** interface. This requires opening `/dev/uhid` and sending a **UHID_CREATE2** event with the device's attributes and report descriptor [59] [60]. Key fields in the `struct uhid_create2_req` include:

- `name`: a name for the device (e.g. "Virtual FIDO2 Authenticator")
- `bus`: the HID bus type. Use `BUS_USB` (value 0x03) to emulate a USB HID device, as FIDO keys are USB class devices.
- `vendor`, `product`, `version`: you can supply arbitrary VID/PID and device version. There are no fixed values required for FIDO – using test or self-assigned IDs is fine (e.g. `0xFFFF/0xFFFF` or any pair not conflicting). These will show up in `/sys` and can help identify the device, but they do not affect FIDO functionality.
- `rd_size` and `rd_data[]`: the HID report descriptor bytes. This must describe the FIDO usage page, etc., as discussed above. Ensure `rd_size` matches the descriptor length.

After writing UHID_CREATE2, the kernel will instantiate a new hidraw device (e.g. `/dev/hidrawX`). The UHID interface will then send a **UHID_START** event back to your program indicating the device is ready [61]. At this point, the virtual device is "live" in the system.

When an application (e.g. browser or `libfido2`) opens the hidraw node, your UHID file descriptor will receive a **UHID_OPEN** event (and later UHID_CLOSE when closed) [62]. The UHID driver handles reference counting, so you'll get one OPEN for the first user and a CLOSE when the last user closes [62]. You may ignore OPEN/CLOSE if not needed, but it can be useful to pause processing when no one is using the device (to simulate power saving).

Receiving Output Reports: When the host sends data to the authenticator (e.g. a CTAPHID command), the kernel HID driver will deliver it via **UHID_OUTPUT** events. Your program should read from `/dev/uhid` to get these events. Each UHID_OUTPUT event contains an `output` struct with: the report type (should be

UHID_OUTPUT_REPORT for our OUT reports) and the `data` buffer plus length [63] [64]. For a FIDO device, this will be the 64-byte output report. Since we have no report IDs, the data will start at byte 0 of the CTAPHID frame. Your code should parse this (per the CTAP framing) and handle the command. For example, on receiving an OUTPUT event containing 64 bytes: interpret the first byte of that buffer as the command+type and proceed to read the whole message (assembling continuation packets if `BCNT` indicates the message spans multiple reports).

Sending Input Reports: When your virtual authenticator needs to send data back (e.g. a CTAPHID response or a spontaneous KEEPALIVE), you must write a **UHID_INPUT2** event. This contains an `input2` struct where you provide the data buffer and length [65]. For instance, to send a 64-byte response packet, you set the size to 64 and copy the bytes into the data field, then write that single event to /dev/uhid. The kernel will then deliver it to whichever process has the hidraw device open (and also make it available via the interrupt IN endpoint as if hardware had sent it) [66] [67].

Control Channel (Feature Reports): HID has a control channel for feature reports, which in UHID are represented by **UHID_GET_REPORT** and **UHID_SET_REPORT** events [66]. In our case, since we did not define any feature reports in the descriptor, these events will rarely occur. However, some operating systems might still query certain reports via the control channel. Notably, Windows may perform a GET_REPORT on the "feature" report 0x00 during device initialization (some HID drivers do this to verify the device is responding). If you receive a UHID_GET_REPORT (with a particular `rtype` and report number) [68] [69], you are expected to reply with UHID_GET_REPORT_REPLY, even if just to say "no data" or an error. Similarly, UHID_SET_REPORT would be used if a host tries to send a feature report. Typically, you can respond with an error code if these are not supported. The UHID protocol allows you to indicate an error by setting `reply->err` to a HID error code (e.g., `EIO` or `ECANCELED`) in the reply struct [66]. Many FIDO authenticators simply might not implement any custom feature reports, so hosts generally won't use this path for CTAP (instead, all CTAP data goes via interrupt OUT reports). Ensure your UHID loop handles GET_REPORT/SET_REPORT by quickly replying (even with err=NOTSUPP) to avoid the host hanging on those requests [70].

UHID Teardown: If the virtual device is unplugged or your program exits, you should send a **UHID_DESTROY** event to clean up [71]. Closing the UHID file descriptor will also destroy the device automatically if not already destroyed. After a UHID_DESTROY, the hidraw node is removed from the system. You can then create a new device again with UHID_CREATE2 if needed (without reopening /dev/uhid) [71].

Device Permissions (udev): On Linux, **udev rules** are important so that regular users can access the hidraw device (otherwise it might be root-only by default). Distributions often include a rule matching FIDO usage page 0xf1d0 to set appropriate permissions. For example, a udev rule may import an ID for FIDO devices and tag them for the desktop login session. A common approach is:

```
KERNEL=="hidraw*", SUBSYSTEM=="hidraw", ENV{ID_SECURITY_TOKEN}!="?*",
IMPORT{program}="u2f-detect $devnode"
```

paired with granting access if `ID_SECURITY_TOKEN` is set [72]. Modern systems (with systemd) have built-in rules that detect usage 0xF1D0 via an external helper or kernel attributes and then apply the `uaccess` tag to allow the logged-in user to read/write the hidraw node [72] [73]. If you find your virtual device isn't

accessible, ensure that the `70-u2f.rules` (or similar) is installed. In short, **no special driver** is needed on Linux – the generic hidraw driver + a udev rule suffice to make the device available to FIDO client software.

Windows and macOS Considerations

Windows 10+ has native support for FIDO2 HID devices. When you plug in a security key with usage page 0xF1D0, Windows will automatically recognize it and use a built-in driver (shown as **"HID-compliant FIDO Device"** in Device Manager) [74]. This driver is part of Windows Hello and allows the key to be used for WebAuthn (e.g. login via Edge/Chrome using the OS's WebAuthn API). **No vendor driver is required** as long as the HID descriptor is correct – the standard HID class driver handles communication. In practice, a software-only FIDO HID device on Windows is not trivial to implement because Windows does not provide a userland UHID equivalent. One would have to write a kernel-mode driver or use a virtual USB bus to simulate the device. If attempting this, the driver (e.g. a virtual HID minidriver) would need to be code-signed to load on 64-bit Windows. However, assuming the device is present, Windows expects the same basics: Usage Page 0xF1D0, Usage 0x01, an interrupt input and output report (64 bytes each). Windows will likely send a HID **Get_Report** control request to fetch the report descriptor during enumeration and then treat the device as a standard HID. It may also issue a feature report query for report ID 0 on startup (some HID class drivers do) – so the device should handle or safely ignore that. Once the device is enumerated, applications can either use the Windows WebAuthn API (which leverages the system's handling of the authenticator) or use the raw HID API (SetupDI and CreateFile on the HID device interface) to communicate. For example, the Microsoft documentation suggests using the HID Win32 APIs to talk to 0xF1D0 devices if implementing custom solutions [75]. In summary, **Windows will natively support a correctly-described FIDO HID device** – the main challenge is injecting a virtual device, which typically requires a kernel driver (and thus driver signing).

On **macOS**, similarly, no custom driver is needed for a physical key – macOS's IOKit will detect the HID device. The Safari browser and other apps can use the built-in WebAuthn support or HID APIs. For a virtual device, macOS does not offer a simple userland HID injector either. One could write a User-space DriverKit extension for HID or create a dummy USB device using the USB Gadget (if on macOS with a virtual USB controller, which is uncommon). That said, as long as the device presents with the FIDO usage page, macOS will list it under IOHIDDevice. The open-source **hidapi** library on macOS, for instance, filters devices by usage page 0xf1d0 to enumerate security keys. No special kext is required aside from the generic HID kext. Thus, for both Windows and macOS, **the critical requirement is the correct HID descriptor** (so the OS knows it's a FIDO authenticator) and then the device will be handled by the standard HID stack. The differences lie mostly in how one might implement a software device (straightforward in Linux via UHID, but requiring kernel-level development on Windows/macOS).

Integration with Browsers and libfido2

Modern browsers (Chrome, Firefox) and libraries (like Yubico's **libfido2**) rely on the above descriptors to find and use the authenticator. Here's how the enumeration and I/O typically work:

- **Device Enumeration:** Libfido2 on Linux uses `udev` to scan for hidraw devices and check their descriptors for usage page 0xF1D0 [76] [18]. It calls `ioctl(HIDIOCGRDESC)` to get the report descriptor and then parses it to find the usage page (and usage) [77] [76]. If it matches FIDO, the device is considered a FIDO authenticator. Libfido2 then fetches some device info from sysfs (like the USB `vendor_id` and `product_id`, and manufacturer string) for display or logging [78] [79]. (If

using UHID with a fake bus, note that `udev_device_get_parent_with_subsystem_devtype(... "usb_device")` might fail if the device isn't actually on a USB bus. Newer libfido2 versions have improved support for pure user-space devices by not strictly requiring a USB parent [80].) Once identified, the library opens the hidraw node (just a file open) [81]. On Windows, similarly, libfido2 (or a browser) would use SetupDi to enumerate HID interfaces and filter by the FIDO usage. On macOS, IOKit APIs (IOHIDManager) can filter by usage page 0xF1D0 to find keys. The important point is that **any device with the correct usage page & usage will be picked up** by FIDO client software [1].

- **Device Opening:** When a WebAuthn operation is initiated, the client (browser) will open the device and perform an **INIT handshake** on it. For example, libfido2 will call `fido_hid_open(path)` which on Linux just does an `open("/dev/hidrawX", O_RDWR)` [82]. This triggers UHID_OPEN to your driver as mentioned. After open, the client typically sends a **CTAPHID_INIT command** on the broadcast CID (0xFFFFFFFF) to obtain a channel. Your authenticator must respond with a valid CTAPHID_INIT response (as described above), which includes a newly generated CID [24] [47]. The library or browser will then use that channel ID for all subsequent messages. If your device fails to respond to INIT properly, the client will not proceed further – so getting INIT right is crucial (matching nonce and correct length of 17 bytes in the response) [83] [84].

- **FIDO2 Operations:** After INIT, the next thing a client usually does is send **CTAPHID_CBOR with an authenticatorGetInfo command**. This asks the device for its basic capabilities (supported versions, extensions, AAGUID, options, maxMsgSize, etc.). Your device should implement the CTAP2 layer to handle this: it needs to return a CBOR map containing at least:

- `versions`: e.g. `["FIDO_2_0"]` (and `"U2F_V2"` if you support U2F) [85]. For maximum browser compatibility, include `"U2F_V2"` in versions if you support the U2F APDU commands via CTAPHID_MSG, otherwise browsers might not permit U2F fallbacks.
- `aaguid`: 16-byte Authenticator Attestation GUID. This should be a fixed UUID for your authenticator model [86]. (If uncertified/self-made, you can use a random GUID or all zeros. All-zero AAGUID is technically allowed and is used by some software authenticators for anonymity, but note that all-zero might also be interpreted by some platforms as "no AAGUID provided". It's better to choose a random constant GUID to identify your virtual authenticator model for testing.)
- `extensions`: list of supported extension identifiers (e.g. `"hmac-secret"`) if any – can be an empty array if none [85].
- `options`: a map of booleans for flags like "plat" (platform device), "rk" (supports resident keys), "up" (User presence required), "uv" (user verification) [87]. For a typical roaming USB key: `"plat": false` (it's removable, not built-in), `"up": true` (it will enforce user presence, e.g. via a touch or a software confirmation), `"uv": false` (if you have no built-in PIN/biometrics), `"rk": false` (if you don't support storing resident credentials). If you do implement client PIN, then also `"clientPin": true/false` and `"uv": true` as appropriate.
- `maxMsgSize`: (optional) max message size in bytes your device can handle [88]. If you stick to 64-byte frames and the standard buffering, you can report `7609` as in the spec example, or omit it, as 7609 is the default for 64-byte frames.
- `pinProtocols`: if you support PIN (CTAP2.1 clientPin), list the protocol versions (e.g. `[1]`). Otherwise it can be omitted or an empty array [88].

Browsers (Chrome/Firefox) will use this info to decide what features to use. For instance, if `uv` (User Verification) is not supported and the site requests UV, the client might know to fall back to just UP or show an error. **Make sure** your GetInfo response is well-formed, as it's often the first real CTAP2 exchange – any parsing error here can cause the browser to mark the device as malfunctioning. Using known-good values (like those from a real YubiKey's GetInfo) as a reference can be helpful.

- **Making Credentials & Assertions:** When the user initiates a registration or login, the browser will prepare a CTAP2 command (MakeCredential or GetAssertion) and send it via CTAPHID_CBOR. Your authenticator application then needs to process that (this involves crypto: creating a key pair, forming attestation, etc., beyond the scope of transport). The key point at the transport level: if an operation involves waiting for user interaction (e.g. "touch the security key"), your device should send periodic **CTAPHID_KEEPALIVE** messages with status `UP_NEEDED` to indicate it's waiting for user touch [42] [89]. For example, Chrome will display a prompt "Touch your security key" and as long as it receives keep-alives, it will keep waiting. If no keepalive or response arrives within a certain time, the browser may timeout the operation. The CTAP2 spec default is 30 seconds for user presence before giving up, but the device should at least send keepalives every 0.5s or so while waiting [90] [91]. Once the user "touches" (or in a virtual case, maybe clicks a UI or times out), the authenticator should complete the operation or send an error. If the user cancels the operation in the browser (e.g. closes dialog), the browser will send CTAPHID_CANCEL; your device should handle UHID_OUTPUT with the cancel command by aborting the current op and perhaps returning `ERR_KEEPALIVE_CANCEL` or a timely error response [92].

- **Browser API Use:** Chrome and Firefox on Linux currently use libfido2 or a similar HID access method under the hood (Chrome has its own HID enumeration code and recently can also use the OS FIDO2 daemon on some platforms). Firefox, as of version 90+, can use either built-in U2F for older keys or direct HID for FIDO2 – ensuring libfido2 is installed often helps Firefox on Linux [93]. On Windows, Chrome and Edge often use the WebAuthn API (which delegates to the system, meaning the OS will talk to the HID device). Firefox on Windows can use either the built-in stack or its own; in either case, the behavior is similar. The important takeaway is that **if your virtual device correctly handles CTAPHID at the transport level and implements the CTAP2 commands properly, the browsers should treat it like a normal hardware key.** They will prompt the user and exchange messages the same way.

- **libfido2 usage:** You can also test your device with libfido2's tools. For example, the `fido2-token -L` command will list all FIDO devices; your device should appear there with the vendor/product name (if retrievable – with UHID you might not have a USB serial, but libfido2 will list something like "vendor 0xffff product 0xffff (FIDO2)"). The `fido2-token -I <device>` will invoke GetInfo on it, and `fido2-token -G/-M` can exercise credential generation and assertion. Another useful test is Python's `fido2` library (Yubico's python-fido2) which has an interactive test (`ctap_test.py`) that can send various commands to a CTAPHID device. Also, websites like **webauthn.io** or **WebAuthn.dev** can be used in a browser to test registration and authentication flows with your virtual authenticator.

Security and Certification Considerations

Even though this is a software authenticator, you should follow the security requirements of the CTAP specification to ensure compatibility and safety:

- **Channel nonce:** Always verify that the 8-byte nonce in the CTAPHID_INIT response matches the one the host sent, to confirm the channel setup [49] [25]. This prevents any mix-up if multiple apps tried to INIT concurrently. The host will check this and ignore the response if it's wrong.
- **Unique Channel IDs:** Generate truly random channel IDs (32-bit). Do not reuse IDs too frequently or use trivial values, to avoid collisions. Never use 0 or 0xFFFFFFFF as an allocated channel (those are reserved: 0 is not valid, and 0xFFFFFFFF is the broadcast).
- **Timeouts and Sequencing:** Implement a message timeout (e.g. if a continuation frame doesn't arrive within a few seconds, reset state) and sequence enforcement (if a wrong sequence number comes or a continuation for an unknown channel, you may drop it or send an error). This ensures the device isn't stuck due to host-side issues. Use `ERR_INVALID_SEQ` (0x04) if out-of-order packets are received [94], `ERR_MSG_TIMEOUT` (0x05) if an incomplete message timed out [94], etc.
- **User presence**: If your authenticator is purely software, define how "user presence" is achieved (maybe a console prompt or automatic approval). For WebAuthn compliance, user presence (a test of user interaction) is required for registration and authentication unless `"up": false` is in options (platform authenticators can be always-on, but roaming ones generally require a touch). If you automatically simulate user presence, be aware this might not be acceptable for real security key usage (it could be okay for testing). At minimum, return the UP flag in authenticator data as true to indicate user presence was delivered.
- **AAGUID and Attestation:** Decide on an **AAGUID** and attestation strategy. If this is for personal/testing use, you can use a dummy self-signed attestation certificate and a random AAGUID. If you want to integrate with the FIDO Metadata Service (so that the authenticator is recognized by its AAGUID and attestation can be validated by relying parties), you would need to obtain an official AAGUID from the FIDO Alliance and produce attestation certs (this typically requires having a proper secure key storage, etc.). For early development, many implementations just use "**self-attestation**" (where the attestation key is generated on the fly, effectively making attestation IDs meaningless). In such a case the AAGUID can be set to all zeros to indicate no specific device info – but note that some RPs might treat all-zero AAGUID as a special case. Alternatively, use a constant GUID and document it as a test device.
- **PIN/UV Support:** If you plan to support PIN UV (user verification via PIN), you will need to implement the authenticatorClientPIN command and related state (PIN token, retries, etc.). This is optional, but without it, your device cannot satisfy `uv=true` requests. Browsers will fall back to just requiring user touch in that case. It's fine to start without PIN (just ensure options `"clientPin": false` and omit pinProtocols).
- **FIDO2 Certification:** If this were to be a certifiable authenticator, there are additional requirements (like specific error behaviors, UV modality if any, clone detection for resident credentials, etc.). Since this is a software device, certification might not be intended – but adhering to spec as closely as possible will ensure broad compatibility.

Testing & Debugging: Tools like Yubico's `fido2-tests` (if available), and the FIDO Alliance conformance test suite (CTAP2 Tests) can validate your implementation against many corner cases. At a minimum, test with multiple browsers (Chrome, Firefox) and the command-line tools. Use a USB packet analyzer or software logs to verify frames are parsed and formed correctly. Each CTAPHID packet should be exactly 64

bytes; remember that on Linux hidraw, when writing, you may need to prepend a dummy report ID byte if the kernel expects one. In our case with no report IDs, Linux's hidraw will use 64-byte writes/reads directly (the `libfido2` code uses 65-byte buffers but sets the first byte to 0 and uses the rest 64 for data) [95] [96]. Ensuring your UHID implementation mirrors this (likely you'll receive 64 bytes in UHID_OUTPUT and should send 64 in UHID_INPUT2) is important.

By fulfilling all the above – correct descriptor, handling of UHID events, proper CTAPHID framing and command responses, and matching the spec's expectations – your software authenticator should appear and function indistinguishably from a hardware FIDO2 key. It will be enumerated by the OS as a HID FIDO device, detected by libfido2 and browsers, and able to perform WebAuthn operations (registering new credentials, authenticating, etc.) just like a physical authenticator. Good luck with implementation and happy testing!

Sources: FIDO Alliance CTAP2 Specification (HID transport) [97] [98] [19], U2F HID definitions [2] [26], Linux UHID documentation [61] [66], and libfido2 source [76] [18], among others, have been referenced to compile these requirements.

---

[1] [3] [4] [5] [11] [12] [19] [20] [21] [24] [25] [28] [29] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [45] [46] [47] [48] [49] [50] [51] [52] [53] [57] [58] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] [97] [98] Client to Authenticator Protocol (CTAP)
https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-client-to-authenticator-protocol-v2.0-id-20180227.html

[2] [22] [23] [26] [27] [30] [44] [54] [55] [56] [94] u2f_hid.h - pub/scm/linux/kernel/git/jejb/fido2-ctap-gadget - Git at Google
https://kernel.googlesource.com/pub/scm/linux/kernel/git/jejb/fido2-ctap-gadget/+/65be37a056e80f59bcd49c28a77bdefa871d857a/u2f_hid.h

[6] [7] [8] [9] [10] [13] CTAPHID report size incorrectly limited · Issue #335 · Yubico/libfido2 · GitHub
https://github.com/Yubico/libfido2/issues/335

[14] [15] [16] [17] [74] FIDO HID report descriptor
https://groups.google.com/a/fidoalliance.org/g/fido-dev/c/0lQb6nlOM48

[18] [76] [77] [78] [79] [81] [82] [95] [96] libfido2.git -
https://git.cryptonomic.net/libfido2.git/diff/src/hid_linux.c?id=c79050aa44b8836d836c5dd22a383a073c28b74b

[59] [61] [62] [66] [67] [70] [71] UHID - User-space I/O driver support for HID subsystem — The Linux Kernel documentation
https://docs.kernel.org/hid/uhid.html

[60] [63] [64] [65] [68] [69] include/uapi/linux/uhid.h - kernel/common - Git at Google
https://android.googlesource.com/kernel/common/+/bce1305c0ece3/include/uapi/linux/uhid.h?autodive=0%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F

[72] [73] Yubico Forum • View topic - Universal udev rules
https://forum.yubico.com/viewtopic4a7c.html?p=6441

[75] Recommended API for accessing HID USB Interface for FIDO device
https://learn.microsoft.com/en-us/answers/questions/294658/recommended-api-for-accessing-hid-usb-interface-fo

[80] Release Notes - Yubico Developers
https://developers.yubico.com/libfido2/Release_Notes.html

93 Universal 2nd Factor - ArchWiki
https://wiki.archlinux.org/title/Universal_2nd_Factor