

- [Openloop SDK](#)
 - [Documentation](#)
- [Openloop SDK](#)
 - [Specifications](#)
- [Architecture Overview](#)
 - [What is the Openloop SDK?](#)
 - [Layer Structure](#)
 - [Security Model](#)
 - [Supported Chains](#)
 - [Transport Connection Methods](#)
 - [Package List](#)
 - [APDU Protocol Overview](#)
 - [Next Steps](#)
- [Quick Start](#)
 - [Installation](#)
 - [The Basic Flow](#)
 - [Complete Ethereum Example](#)
 - [Complete Bitcoin Example](#)
 - [Connecting over Bluetooth](#)
 - [Connecting over LocalWS \(all browsers\)](#)
 - [Using the SDK with React](#)
 - [Automatic Reconnection](#)
 - [Next Steps](#)
- [Transport Guide](#)
 - [Package Structure by SDK Language](#)
 - [Platform Support Matrix](#)
 - [Selection Flowchart](#)
 - [Transport Details](#)
 - [Auto-Detection with the OpenloopSDK Class](#)
 - [Reconnection Patterns](#)
 - [Next Steps](#)
- [Bitcoin API](#)
 - [Import](#)
 - [Initialization](#)
 - [Methods](#)
 - [getAddress](#)
 - [getAddressWithPath](#)
 - [getAccountXpub](#)

- [getMasterFingerprint](#)
- [signMessage](#)
- [signPsbtx](#)
- [signPsbtxHex](#)
- [signPsbtxWithPaths](#)
- [PSBT Protocol Details](#)
- [bitcoind-lib Integration](#)
- [Next Steps](#)
- [Ethereum API](#)
 - [Import](#)
 - [Initialization](#)
 - [Methods](#)
 - [getAddress](#)
 - [signPersonalMessage](#)
 - [signTransaction](#)
 - [signTypedData](#)
 - [signTypedDataStreaming](#)
 - [isEip3009](#)
 - [signAuthorization](#)
 - [ethers.js Integration](#)
 - [viem Integration](#)
 - [Next Steps](#)
- [XRP API](#)
 - [Key Differences from Ethereum](#)
 - [Import](#)
 - [Initialization](#)
 - [Methods](#)
 - [getAddress](#)
 - [signTransaction](#)
 - [xrpl.js Integration](#)
 - [Next Steps](#)
- [Solana API](#)
 - [Import](#)
 - [Initialization](#)
 - [Methods](#)
 - [getAddress](#)
 - [signTransaction](#)
 - [signOffchainMessage](#)
 - [@solana/web3.js Integration](#)

- Next Steps
- TRON API
 - Key Differences from Ethereum
 - Import
 - Initialization
 - Methods
 - getAddress
 - signTransaction
 - signMessage
 - TronWeb Integration
 - Next Steps
- WalletConnect Integration
 - The Two Approaches
 - WcTransport (APDU Relay)
 - AppKit (High-Level WC)
 - Choosing Between WcTransport and AppKit
 - Limitations
 - Next Steps
- Error Handling Guide
 - Error Class Hierarchy
 - ApduError
 - TransportError
 - Error Handling Patterns
 - Next Steps
- API Reference
 - Contents
 - Interfaces
 - App Classes
 - Signature Verification
 - Error Classes
 - OpenloopSDK Class
 - WcTransport Class
 - Constants
 - Utility Functions

Openloop SDK

Documentation

Version: 0.4.0

Updated: 2026-06-25

Hardware Wallet SDK for Web & Node.js

Ethereum | Bitcoin | Solana | TRON | XRP



Openloop SDK Specifications

Version v0.6.2 | Last updated 2026-09-12
Haudi Crypto, Inc. President and CEO Kazunori Asada

Architecture Overview

What is the Openloop SDK?

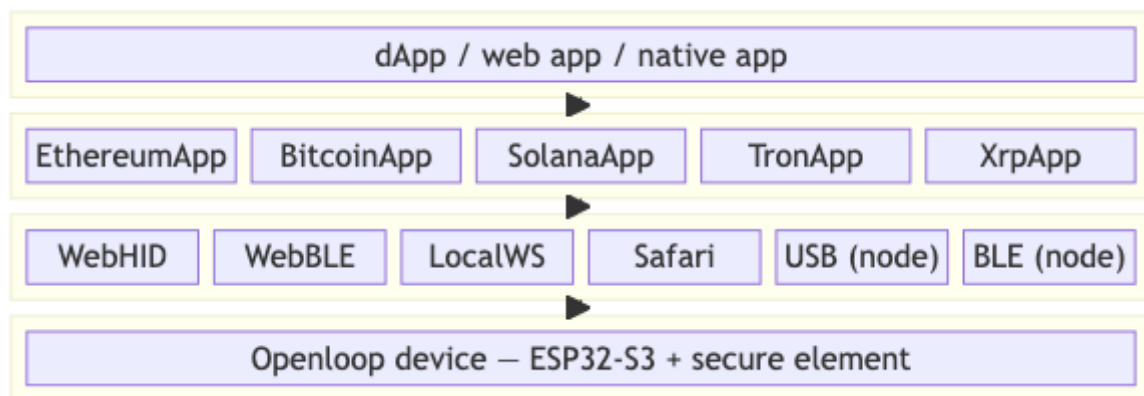
The Openloop SDK is a **multi-language library** for communicating with the Openloop hardware wallet.

Language / format	Main deliverables	Purpose
TypeScript / npm	@openloop/sdk-core , @openloop/transport-*	Web dApps, Node.js / Electron, React Native
Swift / SPM (planned)	—	Native iOS apps
Kotlin / Maven (planned)	—	Native Android
C# / NuGet (planned)	—	Native Windows

About licensing: Use of the Openloop SDK requires a license agreement with Haudi Crypto, Inc. We can also customize the SDK to suit your requirements — please feel free to contact us at info@crypto.haudi.jp.

The single most important principle: private keys stay inside the device and never leave it. The SDK only sends and receives APDU (Application Protocol Data Unit) commands. All signing is performed inside the device's secure element.

Layer Structure



What each layer does

Layer	Role
dApp	User interface, transaction construction
App class	Builds chain-specific APDU commands and parses responses
ITransport	Physical communication with the device (USB / BLE / WebSocket, etc.)
Device	Private key management, signing, user approval

Security Model

What the SDK does

- Builds and sends APDU commands
- Parses responses (addresses, signatures, and so on)
- Manages transport connection and disconnection

What the SDK does not do

- Generate, hold, or operate on private keys
- Perform signing
- Manage seed phrases

Security on the device side

- Private keys are generated and stored inside the device and never leave it

- All signing is performed inside the device
- Transaction details are shown on the device's screen and the user approves them with a physical button
- When rejected or busy, the device returns status word `0x6985` and no signature is produced

Supported Chains

Chain	App class	Elliptic curve	BIP44 coin_type
Ethereum (EVM)	<code>EthereumApp</code>	secp256k1	<code>60'</code>
Bitcoin	<code>BitcoinApp</code>	secp256k1	<code>0'</code> (mainnet), <code>1'</code> (testnet)
Solana	<code>SolanaApp</code>	Ed25519	<code>501'</code>
TRON	<code>TronApp</code>	secp256k1	<code>195'</code>
XRP Ledger	<code>XrpApp</code>	Ed25519 / secp256k1	<code>144'</code>

EVM-compatible chains

`EthereumApp` works not only with Ethereum but with any EVM-compatible chain. Specify the chain through the `chainId` parameter of `signPersonalMessage` and `signTypedData`.

Transport Connection Methods

The SDK offers six connection methods, covering a wide range of platforms.

Transport	Package	Use case
WebHID (USB)	<code>@openloop/transport-webhid</code>	Chrome/Edge on desktop
Web Bluetooth	<code>@openloop/transport-webble</code>	Chrome/Edge + Android
LocalWS	<code>@openloop/transport-local</code>	All browsers (via Connect)
Safari Extension	<code>@openloop/transport-safari</code>	iOS Safari
USB HID (native)	<code>@openloop/transport-usb</code>	Node.js / Electron
BLE (native)	<code>@openloop/transport-ble</code>	Node.js / Electron
WalletConnect	Built into <code>@openloop/sdk-core</code>	All browsers (remote signing)

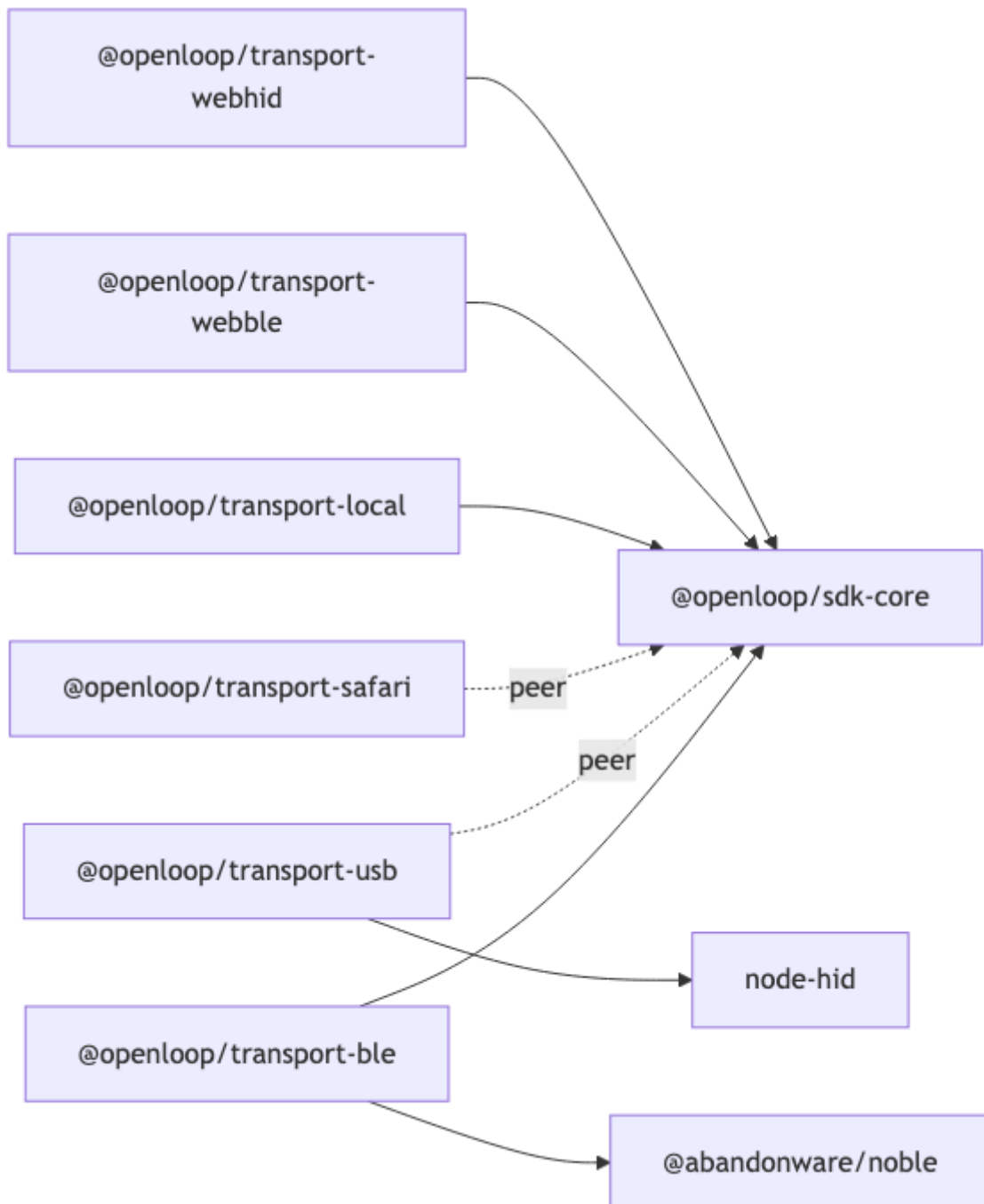
See the [Transport Guide](#) for details.

Package List

npm packages (TypeScript)

Package	Description	Dependencies
<code>@openloop/sdk-core</code>	Core library (App, Types, Errors, Constants, WcTransport)	none
<code>@openloop/transport-webhid</code>	WebHID (USB) transport	sdk-core
<code>@openloop/transport-webble</code>	Web Bluetooth transport	sdk-core
<code>@openloop/transport-local</code>	LocalWS transport	sdk-core
<code>@openloop/transport-safari</code>	Safari extension transport	sdk-core (peer)
<code>@openloop/transport-usb</code>	Native USB HID transport	sdk-core (peer), node-hid
<code>@openloop/transport-ble</code>	Native BLE transport (Node.js / Electron)	sdk-core (peer), noble

Dependency graph



`sdk-core` is a pure TypeScript package with no dependency on browser APIs or native modules. Each transport package supplies the platform-specific implementation. Swift / Kotlin / C# libraries for native apps are planned.

APDU Protocol Overview

Communication between the SDK and the device uses APDU (Application Protocol Data Unit) commands.

Command structure

```
[CLA] [INS] [P1] [P2] [Lc] [Data...]
```

Field	Size	Description
CLA	1 byte	Class byte (<code>0xE0</code> : Ledger-compatible, <code>0xF0</code> : Openloop-specific)
INS	1 byte	Instruction code
P1	1 byte	Parameter 1 (chunking control, etc.)
P2	1 byte	Parameter 2 (curve selection, etc.)
Lc	1 byte	Data length
Data	Lc bytes	Command data

Response structure

```
[Data...] [SW1] [SW2]
```

Field	Size	Description
Data	variable	Response data
SW1-SW2	2 bytes	Status word (<code>0x9000</code> = success)

CLA bytes

CLA	Purpose
0xE0	Ledger-compatible commands (ETH, BTC getAddress, SOL, TRON, XRP)
0xF0	Openloop-specific commands (chain_id-aware signing, PSBT, BTC xpub)

See [Error Handling](#) for the full list of APDU status words.

Next Steps

- [Quick Start](#) — a getting-started guide for developers
- [Transport Guide](#) — the platform support matrix
- [API Reference](#) — all type definitions and interfaces

Quick Start

This guide walks you through connecting to the hardware wallet with the Openloop SDK, retrieving an address, and signing a transaction.

Installation

Install the core package plus the transport package for the connection method you plan to use.

```
# USB connection (Chrome/Edge on desktop)
npm install @openloop/sdk-core @openloop/transport-webhid

# Bluetooth connection (Chrome/Edge + Android)
npm install @openloop/sdk-core @openloop/transport-webble

# All browsers (via Openloop Connect)
npm install @openloop/sdk-core @openloop/transport-local

# iOS Safari
npm install @openloop/sdk-core @openloop/transport-safari

# Node.js / Electron
npm install @openloop/sdk-core @openloop/transport-usb
```

The Basic Flow

The same four steps apply to every connection method.

```
1. Connect transport → 2. Create app → 3. Operate → 4. Disconnect
```

```
import { EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

// 1. Connect the transport (must be called inside a user gesture)
const transport = await WebHidTransport.connect()

// 2. Create the app
const eth = new EthereumApp(transport)

// 3. Operate
const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
console.log('Address:', address)

// 4. Disconnect
await transport.close()
```

Note: Browser security rules require `WebHidTransport.connect()` and `WebBleTransport.connect()` to be called inside a user gesture, such as a button click.

Complete Ethereum Example

```
import {
  EthereumApp,
  DEFAULT_ETH_PATH,
  type SignatureResult,
} from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

async function main() {
  // Connect
  const transport = await WebHidTransport.connect()
  const eth = new EthereumApp(transport)

  try {
    // Get the address
    const { address, publicKey } = await eth.getAddress(DEFAULT_ETH_PATH)
    console.log('Address:', address)
    console.log('Public Key:', publicKey)

    // Sign a message (EIP-191 personal_sign)
    const msgSig: SignatureResult = await eth.signPersonalMessage(
      DEFAULT_ETH_PATH,
      'Hello, Openloop!',
      1 // chainId: Ethereum Mainnet
    )
    console.log('Message Signature:', { v: msgSig.v, r: msgSig.r, s: msgSig.s })

    // Sign a transaction (pass the RLP-encoded raw tx)
    const rawTx = 'f86c...' // RLP-encoded transaction hex
    const txSig: SignatureResult = await eth.signTransaction(DEFAULT_ETH_PATH, rawTx)
    console.log('TX Signature:', { v: txSig.v, r: txSig.r, s: txSig.s })

    // Sign EIP-712 typed data
    const typedSig: SignatureResult = await eth.signTypedData(
      DEFAULT_ETH_PATH,
      'aabbccdd...', // domainSeparatorHash (32 bytes hex)
      '11223344...', // messageHash (32 bytes hex)
      1
    )
    console.log('TypedData Signature:', typedSig)
  } finally {
    await transport.close()
  }
}
```

Complete Bitcoin Example

```
import {
  BitcoinApp,
  BTC_MAINNET_PATH,
  type BtcMessageSignature,
} from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

async function main() {
  const transport = await WebHidTransport.connect()
  const btc = new BitcoinApp(transport)

  try {
    // Get the address (SegWit bech32)
    const { address, publicKey, chainCode } = await btc.getAddress(false) // mainnet
    console.log('Address:', address) // bc1q...

    // Sign a message (BIP-137)
    const msgSig: BtcMessageSignature = await btc.signMessage(
      'Hello, Bitcoin!',
      BTC_MAINNET_PATH
    )
    console.log('Signature (base64):', msgSig.signature)

    // Sign a PSBT
    const psbtHex = '70736274ff...' // PSBT binary as hex
    const signedPsbtHex = await btc.signPsbtHex(psbtHex)
    console.log('Signed PSBT:', signedPsbtHex)

    // Sign a PSBT with explicit input paths
    const signedWithPaths = await btc.signPsbtWithPaths(psbtHex, [
      { index: 0, path: "84'/0'/0'/0/0" },
      { index: 1, path: "84'/0'/0'/0/1" },
    ])
    console.log('Signed PSBT with paths:', signedWithPaths)
  } finally {
    await transport.close()
  }
}
```

Connecting over Bluetooth

Just swap WebHID for WebBLE — the rest of the code is identical.

```
import { WebBleTransport } from '@openloop/transport-webble'

// Connect over Bluetooth (must be called inside a user gesture)
const transport = await WebBleTransport.connect()

// Everything from here on is the same as WebHID
const eth = new EthereumApp(transport)
const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
```

Connecting over LocalWS (all browsers)

Connect through the Openloop Connect desktop app. This works even in Firefox and Safari.

```
import { LocalWsTransport } from '@openloop/transport-local'

// Check whether the Connect app is running
const available = await LocalWsTransport.isAvailable()
if (!available) {
  console.log('Please start Openloop Connect')
}

// Connect
const transport = new LocalWsTransport()
await transport.open()

// Everything from here on is the same
const eth = new EthereumApp(transport)
```

Using the SDK with React

```
import { useState, useCallback } from 'react'
import { EthereumApp, DEFAULT_ETH_PATH, type ITransport } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

function WalletButton() {
  const [address, setAddress] = useState<string>('')
  const [transport, setTransport] = useState<ITransport | null>(null)

  const connect = useCallback(async () => {
    const t = await WebHidTransport.connect()
    setTransport(t)
    const eth = new EthereumApp(t)
    const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
    setAddress(address)
  }, [])

  const disconnect = useCallback(async () => {
    await transport?.close()
    setTransport(null)
    setAddress('')
  }, [transport])

  return (
    <div>
      {address ? (
        <>
          <p>{address}</p>
          <button onClick={disconnect}>Disconnect</button>
        </>
      ) : (
        <button onClick={connect}>Connect Wallet</button>
      )}
    </div>
  )
}
```

The sample app's `useOpenloop` hook (`packages/sample-app/src/hooks/useOpenloop.ts`) is a worked example that covers switching between all transports, WalletConnect integration, and automatic reconnection.

Automatic Reconnection

WebHID and WebBLE support reconnecting automatically to a device the user has previously authorized.

```
// Reconnect automatically to a previously authorized device (no user gesture needed)
const transport = await WebHidTransport.reconnect()
if (transport) {
  // Reconnected successfully
  const eth = new EthereumApp(transport)
} else {
  // Device not found → pairing via connect() is required
}
```

Next Steps

- [Transport Guide](#) — choose the connection method that fits your environment
- [Ethereum API](#) — ethers.js / viem integration
- [Bitcoin API](#) — the PSBT protocol in detail
- [Error Handling](#) — error-handling patterns

Transport Guide

The Openloop SDK supports several connection methods (transports). Pick the transport that matches your target platform and use case — dApp or native app.

Package Structure by SDK Language

The Openloop SDK ships as a **set of TypeScript packages (npm)**. Swift / Kotlin / C# libraries for native apps are planned.

Language / format	Registry	Purpose
TypeScript / npm	npmjs.com (<code>@openloop/*</code>)	Web dApps, Node.js / Electron, React Native (including Connect Mobile)
Swift / SPM	(planned)	Native iOS apps
Kotlin / Maven	(planned)	Native Android
C# / NuGet	(planned)	Native Windows

Platform Support Matrix

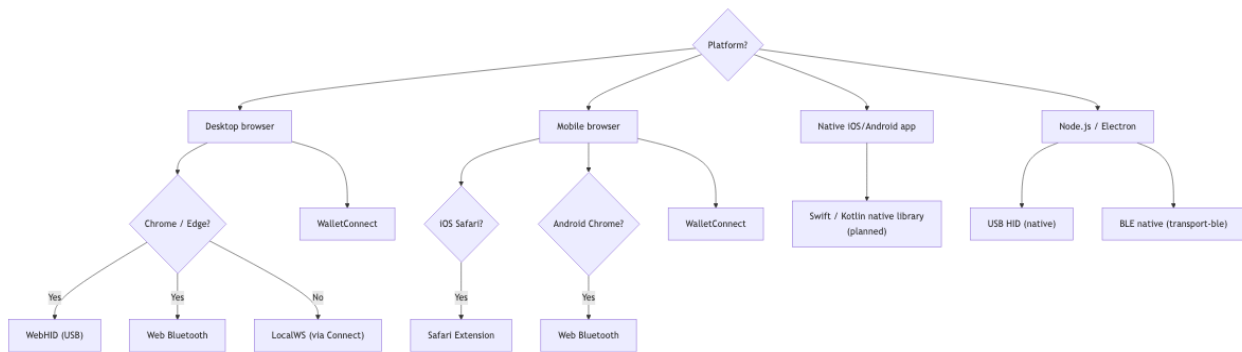
Transport	Chrome/Edge	Firefox	Safari (Desktop)	iOS Safari	Android Chrome	iOS Native	Android Native	Node.js / Electron
WebHID (USB)	✓	✗	✗	✗	✗	✗	✗	✗
Web Bluetooth	✓	✗	✗	✗	✓	✗	✗	✗
LocalWS	✓	✓	✓	✗	✓	✗	✗	✗
Safari Extension	✗	✗	✗	✓	✗	✗	✗	✗
USB HID (native)	✗	✗	✗	✗	✗	✗	✗	✓
BLE (native)	✗	✗	✗	✗	✗	✗	✗	✓
Native lib (Swift / Kotlin — planned)	✗	✗	✗	✗	✗	Planned	Planned	✗
Wallet Connect	✓	✓	✓	✓	✓	△	△	✗

WalletConnect does not talk to the device directly — it is remote signing through the Openloop Connect app. Device operations such as address retrieval happen on the Connect side.

BLE (native): a noble-based BLE transport (`@openloop/transport-ble`) for Node.js / Electron. It is the BLE counterpart of USB HID (native), talking to the device directly without going through a browser.

Native lib (planned): Swift / Kotlin libraries that connect directly from native iOS / Android apps are planned. Until then, use iOS Safari (Safari Extension) or Android Chrome (Web Bluetooth) on mobile.

Selection Flowchart



Transport Details

WebHID (USB)

Package: `@openloop/transport-webhid` **Supported environments:** Chrome 89+ / Edge 89+ (desktop only)

Connects the device directly over a USB cable. This is the fastest and most stable connection method.

Static Methods

Method	Description
<code>isSupported(): boolean</code>	Whether the WebHID API is available
<code>connect(): Promise<WebHidTransport></code>	Shows the device picker and connects (requires a user gesture)
<code>reconnect(): Promise<WebHidTransport null></code>	Reconnects automatically to a previously authorized device

Code Example

```
import { WebHidTransport } from '@openloop/transport-webhid'

// Check support
if (!WebHidTransport.isSupported()) {
  console.log('WebHID is not supported in this browser')
}

// First connection (inside a button click)
const transport = await WebHidTransport.connect()

// Automatic reconnection (on page load)
const transport = await WebHidTransport.reconnect()

// Detect disconnection
transport.onDisconnect(() => {
  console.log('Device disconnected')
})
```

Web Bluetooth

Package: @openloop/transport-webble **Supported environments:** Chrome 56+ / Edge 79+ (desktop), Android Chrome

Connects wirelessly over Bluetooth Low Energy (BLE).

Static Methods

Method	Description
<code>isSupported(): boolean</code>	Whether the Web Bluetooth API is available
<code>connect(): Promise<WebBleTransport></code>	Shows the device picker and connects (requires a user gesture)
<code>reconnect(): Promise<WebBleTransport null></code>	Reconnects to a previously paired device (Chrome 100+)

Code Example

```
import { WebBleTransport } from '@openloop/transport-webble'

if (!WebBleTransport.isSupported()) {
  console.log('Web Bluetooth is not supported')
}

// First connection
const transport = await WebBleTransport.connect()

// Automatic reconnection
const transport = await WebBleTransport.reconnect()
```

Notes

- BLE may disconnect on its own while idle. The transport attempts an automatic reconnection on the next `exchange()` call.
- The MTU (Maximum Transmission Unit) is negotiated with the device, and the frame size is adjusted automatically.
- Default MTU: 155 bytes

LocalWS (All Browsers)

Package: `@openloop/transport-local` **Supported environments:** every browser that supports WebSocket **Prerequisite:** the [Openloop Connect](#) desktop app must be running

Communication goes through the WebSocket server (`ws://127.0.0.1:21320`) provided by the Openloop Connect desktop app. It works even in browsers without WebHID/WebBLE support, such as Firefox and Safari.

Static Methods

Method	Description
<code>isAvailable(port?, host?): Promise<boolean></code>	Whether the Connect server is running

Code Example

```
import { LocalWsTransport } from '@openloop/transport-local'

// Check whether the Connect app is running
const available = await LocalWsTransport.isAvailable()

// Connect
const transport = new LocalWsTransport({
  port: 21320, // default
  host: '127.0.0.1', // default
})
await transport.open()

// Notification when the device connection state changes
transport.onDeviceChange((connected: boolean) => {
  console.log('Device:', connected ? 'connected' : 'disconnected')
})
```

Protocol

JSON-RPC 2.0 over WebSocket:

Method	Description
<code>openloop_exchange</code>	Sends and receives APDU commands
<code>openloop_lock</code>	Acquires an exclusive lock on the device
<code>openloop_unlock</code>	Releases the exclusive lock on the device
<code>openloop_status</code>	Retrieves the device connection state

Safari Extension (iOS)

Package: `@openloop/transport-safari` **Supported environments:** iOS Safari (with the Openloop Safari Extension installed)

Reaches iOS CoreBluetooth through a Safari Web Extension and talks to the device over BLE.

Static Methods

Method	Description
<code>isAvailable(timeout?): Promise<boolean></code>	Whether the Safari Extension is installed (3-second timeout by default)
<code>scan(duration?): Promise<DeviceInfo[]></code>	Scans for BLE devices
<code>connect(deviceId?): Promise<SafariTransport></code>	Connects to a device

Code Example

```
import { SafariTransport } from '@openloop/transport-safari'

// Check for the extension
const available = await SafariTransport.isAvailable()

// Scan for devices
const devices = await SafariTransport.scan(5000) // scan for 5 seconds
console.log('Found devices:', devices)

// Connect
const transport = await SafariTransport.connect(devices[0]?.deviceId)
```

USB HID (Native)

Package: `@openloop/transport-usb` **Supported environments:** Node.js / Electron
Dependency: `node-hid`

Talks to the USB device directly, without a browser. A good fit for server-side code and desktop apps.

Static Methods

Method	Description
<code>discover(): UsbDeviceInfo[]</code>	Enumerates the connected Openloop/Ledger devices

Code Example

```
import { UsbHidTransport } from '@openloop/transport-usb'

// Detect devices
const devices = UsbHidTransport.discover()
if (devices.length === 0) {
  console.log('No device found')
}

// Connect
const transport = new UsbHidTransport({ path: devices[0].path })
await transport.open()

// Automatic retry (up to 3 times by default)
const transport = new UsbHidTransport({
  path: devices[0].path,
  maxRetries: 3,
  exchangeTimeout: 120000, // 2 minutes
})
```

BLE (Native)

Package: @openloop/transport-ble **Supported environments:** Node.js / Electron

Dependency: @abandonware/noble

Uses noble to talk to the device over BLE directly, without going through a browser. It is the BLE counterpart of USB HID (native), and a good fit for server-side code and desktop apps.

Static Methods

Method	Description
<code>scan(timeout?): Promise<BleDeviceInfo[]></code>	Scans for BLE devices (30 seconds by default)

Code Example

```
import { BleTransport } from '@openloop/transport-ble'

// Scan for devices
const devices = await BleTransport.scan()

// Connect
const transport = new BleTransport({ deviceId: devices[0].id })
await transport.open()

// Detect disconnection
transport.onDisconnect(() => {
  console.log('Device disconnected')
})
```

Notes

- Node.js / Electron only — it does not work in a browser or in React Native
- The MTU is negotiated with the device (155 bytes by default)

WalletConnect

Package: `@openloop/sdk-core` (WcTransport is bundled with sdk-core) **Supported environments:** every browser that supports WebSocket

Communicates with a remote Openloop Connect app through the WalletConnect v2 relay. APDU commands are relayed using the custom method `openloop_exchange`.

See [WalletConnect Integration](#) for details.

Auto-Detection with the OpenloopSDK Class

The `OpenloopSDK` class auto-detects the available transports and connects for you.

```

import { OpenloopSDK, EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'
import { WebBleTransport } from '@openloop/transport-webble'
import { LocalWsTransport } from '@openloop/transport-local'

// Register the transports
OpenloopSDK.registerTransport('webhid', {
  factory: () => WebHidTransport.connect(),
  name: 'WebHID (USB)',
  isAvailable: () => WebHidTransport.isSupported(),
})

OpenloopSDK.registerTransport('webble', {
  factory: () => WebBleTransport.connect(),
  name: 'Web Bluetooth',
  isAvailable: () => WebBleTransport.isSupported(),
})

OpenloopSDK.registerTransport('local', {
  factory: async () => {
    const t = new LocalWsTransport()
    await t.open()
    return t
  },
  name: 'LocalWS (Connect)',
  isAvailable: () => LocalWsTransport.isAvailable(),
})

// Check which transports are available
const transports = await OpenloopSDK.discover()
console.log(transports)
// [
//   { type: 'webhid', name: 'WebHID (USB)', available: true },
//   { type: 'webble', name: 'Web Bluetooth', available: true },
//   { type: 'local', name: 'LocalWS (Connect)', available: false },
// ]

// Connect using a specific transport
const transport = await OpenloopSDK.connect({ transport: 'webhid' })

// Or connect using the first available transport
const transport = await OpenloopSDK.connect()

```

Reconnection Patterns

```
// Reconnect automatically to the previous device on page load
async function autoReconnect(): Promise<ITransport | null> {
  // WebHID: look for a previously authorized device
  if (WebHidTransport.isSupported()) {
    const transport = await WebHidTransport.reconnect()
    if (transport) return transport
  }

  // WebBLE: look for a previously paired device (Chrome 100+)
  if (WebBleTransport.isSupported()) {
    const transport = await WebBleTransport.reconnect()
    if (transport) return transport
  }

  // LocalWS: connect if the Connect app is running
  if (await LocalWsTransport.isAvailable()) {
    const transport = new LocalWsTransport()
    await transport.open()
    return transport
  }

  return null
}
```

Next Steps

- [Ethereum API](#) — every method on EthereumApp
- [Bitcoin API](#) — the PSBT protocol in detail
- [WalletConnect Integration](#) — WcTransport / AppKit

Bitcoin API

The `BitcoinApp` class provides Bitcoin address retrieval, message signing (BIP-137), and PSBT (Partially Signed Bitcoin Transaction) signing.

Import

```
import {
  BitcoinApp,
  BTC_MAINNET_PATH,
  BTC_TESTNET_PATH,
  type BtcMessageSignature,
  type InputSigningPath,
} from '@openloop/sdk-core'
```

Initialization

```
const btc = new BitcoinApp(transport)
```

Methods

Method	Description
<code>getAddress(testnet?)</code>	Retrieves a SegWit address (switches between mainnet and testnet automatically)
<code>getAddressWithPath(path)</code>	Retrieves an address at an arbitrary BIP32 path
<code>getAccountXpub(coinType)</code>	Retrieves the account-level extended public key
<code>getMasterFingerprint()</code>	Retrieves the master key fingerprint (4 bytes hex)
<code>signMessage(message, path)</code>	BIP-137 message signing
<code>signPsbt(psbt)</code>	PSBT signing (binary in and out)
<code>signPsbtHex(psbt)</code>	PSBT signing (hex in and out)
<code>signPsbtWithPaths(psbt, inputPaths)</code>	PSBT signing with explicit input paths

getAddress

Retrieves a SegWit (bech32) address.

```
async getAddress(testnet?: boolean): Promise<{
  publicKey: string // uncompressed public key (hex)
  address: string // bech32 address ("bc1..." or "tb1...")
  chainCode: string // chain code (hex, 32 bytes)
}>
```

Parameters

Name	Type	Default	Description
testnet	boolean	false	true : testnet path (84'/1'/0'/0/0)

Default Paths

Network	Path	Constant
Mainnet	84'/0'/0'/0/0	BTC_MAINNET_PATH
Testnet	84'/1'/0'/0/0	BTC_TESTNET_PATH

Example

```
// Mainnet
const { address } = await btc.getAddress()
// address: "bc1q..."

// Testnet
const { address } = await btc.getAddress(true)
// address: "tb1q..."
```

getAddressWithPath

Retrieves an address at an arbitrary BIP32 path. Use it for multi-account setups and for change addresses.

```

async getAddressWithPath(path: string): Promise<{
  publicKey: string
  address: string
  chainCode: string
}>

```

Example

```

// First address of the second account
const { address } = await btc.getAddressWithPath("84'/0'/1'/0/0")

// Change address
const { address } = await btc.getAddressWithPath("84'/0'/0'/1/0")

// The "m/" prefix is also accepted
const { address } = await btc.getAddressWithPath("m/84'/0'/0'/0/5")

```

getAccountXpub

Retrieves the account-level extended public key (`m/84'/coin'/0'`). Derive child keys from it in software to check balances and generate addresses.

```

async getAccountXpub(coinType: number): Promise<{
  publicKey: string // compressed public key (hex, 33 bytes)
  chainCode: string // chain code (hex, 32 bytes)
}>

```

Parameters

Name	Type	Description
coinType	number	0 : mainnet, 1 : testnet

Example

```

const { publicKey, chainCode } = await btc.getAccountXpub(0) // mainnet

```

getMasterFingerprint

Retrieves the wallet's **BIP32 master key fingerprint** — `hash160(master public key)[:4]`. It is identical to the value the device computes internally, and it is the value to place in a PSBT input's BIP32 derivation (`master_fingerprint`).

```
async getMasterFingerprint(): Promise<string> // 8 lowercase hex characters (big-endian), e.g.
```

When to Use It

In a normal wallet integration you fetch this value once, at account import time, and embed it in the input derivations of every PSBT you build afterwards. It matters most on the **AirGap / watch-only (QR PSBT)** path: the device decides an input is its own by matching this fingerprint, so a PSBT input without the correct `master_fingerprint` is treated as belonging to another wallet and cannot be signed — there is no input-path injection there like `signPsbtWithPaths` over USB.

Internally it fetches the public key at depth 0 (master `m`) and returns the first 4 bytes of its `hash160`. This is the same processing Sparrow / HWI perform.

Example

```
const fp = await btc.getMasterFingerprint() // e.g. "631698e0"  
  
// Embed it in the PSBT input's BIP32 derivation (type 0x06):  
// value = fingerprint(4B) || path(4B LE × N)
```

signMessage

Signs a message in BIP-137 format.

```
async signMessage(  
  message: string,  
  path: string  
) : Promise<BtcMessageSignature>
```

Parameters

Name	Type	Description
<code>message</code>	<code>string</code>	UTF-8 message
<code>path</code>	<code>string</code>	BIP32 path (e.g. <code>"84'/0'/0'/0/0"</code>)

Returns: `BtcMessageSignature`

Field	Type	Description
<code>v</code>	<code>number</code>	Recovery ID (35-38: P2WPKH native SegWit)
<code>r</code>	<code>string</code>	R value (32 bytes hex)
<code>s</code>	<code>string</code>	S value (32 bytes hex)
<code>signature</code>	<code>string</code>	Base64 of the full signature (65 bytes: V R S)

Example

```
const sig = await btc.signMessage('Hello, Bitcoin!', BTC_MAINNET_PATH)
console.log(sig.signature) // Base64 encoded signature
```

signPsbt

Signs a PSBT (Partially Signed Bitcoin Transaction). The paths are detected automatically from the `BIP32_DERIVATION` fields inside the PSBT.

```
async signPsbt(psbt: Uint8Array | string): Promise<Uint8Array>
```

Parameters

Name	Type	Description
<code>psbt</code>	<code>Uint8Array</code> <code>string</code>	PSBT binary data or hex string

Returns

The signed PSBT as binary data (`Uint8Array`)

signPsbtHex

PSBT signing with hex strings — both input and output are hex strings.

```
async signPsbtHex(psbt: string): Promise<string>
```

Example

```
const signedHex = await btc.signPsbtHex('70736274ff...')
```

signPsbtWithPaths

Signs a PSBT with the BIP32 path of each input given explicitly. Use it when the PSBT has no `BIP32_DERIVATION` fields, or when you want to sign only specific inputs.

```
async signPsbtWithPaths(  
  psbt: string,  
  inputPaths: InputSigningPath[]  
) : Promise<string>
```

Parameters

Name	Type	Description
<code>psbt</code>	<code>string</code>	PSBT hex string
<code>inputPaths</code>	<code>InputSigningPath[]</code>	The path for each input

InputSigningPath

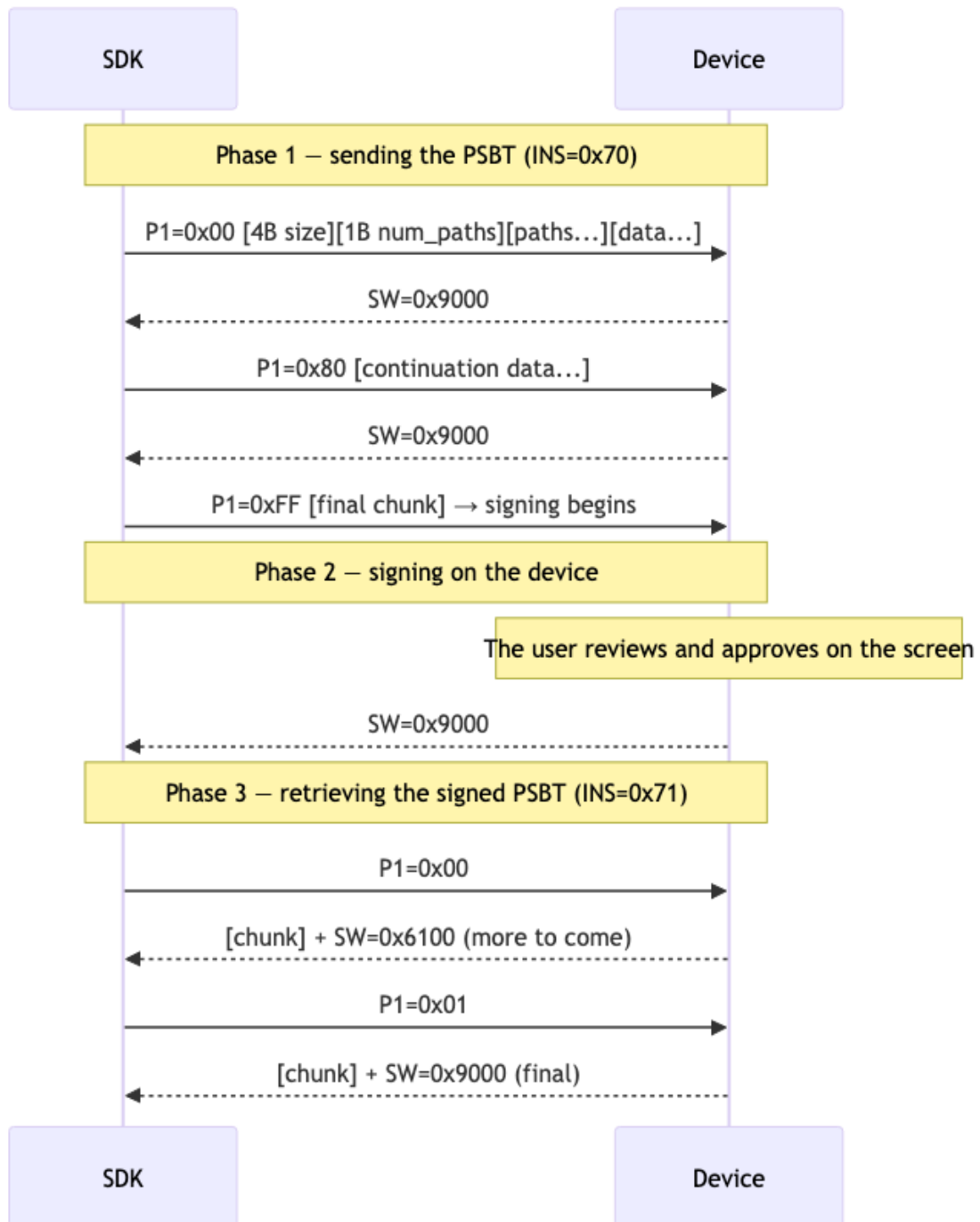
Field	Type	Description
index	number	Input index in the PSBT
path	string	BIP32 path (e.g. "84'/0'/0'/0/5")

Example

```
const signedHex = await btc.signPsbtcWithPaths('70736274ff...', [  
  { index: 0, path: "84'/0'/0'/0/0" },  
  { index: 1, path: "84'/0'/0'/0/3" },  
  { index: 2, path: "84'/0'/0'/1/0" }, // change  
])
```

PSBT Protocol Details

PSBT signing is a three-phase exchange with the device.



Phase 1: Sending the PSBT

Chunk	P1	Data format
First	0x00	[4B total_size][1B num_paths][paths...] [psbt_data...]
Continuation	0x80	[psbt_data...]
Final	0xFF	[remaining_data] → the device starts signing

- `num_paths=0` : detected automatically from `BIP32_DERIVATION` inside the PSBT
- `num_paths>0` : the path for each input is given explicitly ([1B input_index][1B path_len] [4B * path_len elements])

Phase 2: Signing on the Device

The user reviews the transaction details on the device screen and presses the approve button.

Phase 3: Retrieving the Signed PSBT

Response	Status word	Meaning
[signed_psbt_chunk]	0x61XX	More data follows — request the next chunk
[signed_psbt_chunk]	0x9000	Final chunk — retrieval complete

Increment the chunk index from 0 and repeat until `0x9000` comes back.

bitcoinjs-lib Integration

```
import * as bitcoin from 'bitcoinjs-lib'
import { BitcoinApp, BTC_MAINNET_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const btc = new BitcoinApp(transport)

// Get the address
const { address, publicKey } = await btc.getAddress()

// Build the PSBT
const psbt = new bitcoin.Psbt({ network: bitcoin.networks.bitcoin })
psbt.addInput({
  hash: 'txid...',
  index: 0,
  witnessUtxo: {
    script: Buffer.from('0014...', 'hex'),
    value: 50000,
  },
})
psbt.addOutput({
  address: 'bc1q...',
  value: 40000,
})

// Sign the PSBT on the device
const psbtHex = psbt.toHex()
const signedHex = await btc.signPsbtWithPaths(psbtHex, [
  { index: 0, path: BTC_MAINNET_PATH },
])

// Load the signed PSBT
const signedPsbt = bitcoin.Psbt.fromHex(signedHex)
signedPsbt.finalizeAllInputs()

// Raw transaction ready to broadcast
const rawTx = signedPsbt.extractTransaction().toHex()
```

Next Steps

- Solana API
- Error Handling — handling rejections and busy devices

Ethereum API

The `EthereumApp` class provides Ethereum address retrieval, message signing, transaction signing, EIP-712 typed data signing, and EIP-7702 authorization signing.

Import

```
import {
  EthereumApp,
  DEFAULT_ETH_PATH,
  type SignatureResult,
} from '@openloop/sdk-core'
```

Initialization

```
const eth = new EthereumApp(transport)
```

`transport` is any transport instance that implements the `ITransport` interface.

Methods

Method	Description
<code>getAddress(path)</code>	Retrieves the address and public key
<code>signPersonalMessage(path, message, chainId?)</code>	EIP-191 <code>personal_sign</code>
<code>signTransaction(path, rawTx)</code>	Signs an RLP-encoded transaction
<code>signTypedData(path, domainHash, msgHash, chainId?)</code>	Signs EIP-712 typed data
<code>signAuthorization(path, authorizationRlp)</code>	Signs an EIP-7702 authorization
<code>signTypedDataStreaming(path, typedData, chainId?)</code>	Sign EIP-712 typed data by streaming it (the device can show the contents)
<code>isEip3009(typedData)</code>	Whether typed data is EIP-3009 (used to choose the path)

getAddress

Retrieves the Ethereum address and public key from the device.

```
async getAddress(path: string): Promise<{
  publicKey: string // uncompressed public key (hex, 65 bytes)
  address: string   // checksummed address ("0x...")
}>
```

Parameters

Name	Type	Description
path	string	BIP44 path (e.g. "44'/60'/0'/0/0")

Example

```
import { DEFAULT_ETH_PATH } from '@openloop/sdk-core'

const { address, publicKey } = await eth.getAddress(DEFAULT_ETH_PATH)
// address: "0x742d35Cc6634C0532925a3b844Bc9e7595f2bD18"
// publicKey: "04a1b2c3d4..."
```

Default Path

```
DEFAULT_ETH_PATH = "44'/60'/0'/0/0"
```

signPersonalMessage

Signs a message with EIP-191 `personal_sign`. It uses the Openloop-specific extended protocol that is aware of the chain ID.

```
async signPersonalMessage(
  path: string,
  message: string,
  chainId?: number
): Promise<SignatureResult>
```

Parameters

Name	Type	Default	Description
<code>path</code>	<code>string</code>	—	BIP44 path
<code>message</code>	<code>string</code>	—	The message — a UTF-8 string or <code>0x</code> -prefixed hex
<code>chainId</code>	<code>number</code>	<code>1</code>	EVM chain ID

Returns: `SignatureResult`

Field	Type	Description
<code>v</code>	<code>number</code>	Recovery ID (27 or 28)
<code>r</code>	<code>string</code>	R value (32 bytes hex, no <code>0x</code>)
<code>s</code>	<code>string</code>	S value (32 bytes hex, no <code>0x</code>)

Example

```
const sig = await eth.signPersonalMessage(  
  DEFAULT_ETH_PATH,  
  'Hello, Openloop!',  
  1  
)  
// sig.v = 27  
// sig.r = "a1b2c3..."  
// sig.s = "d4e5f6..."
```

Note: Long messages are split into chunks automatically before they are sent to the device — 150 bytes for the first chunk, then 255 bytes each.

signTransaction

Signs an RLP-encoded Ethereum transaction.

```
async signTransaction(  
  path: string,  
  rawTx: string  
): Promise<SignatureResult>
```

Parameters

Name	Type	Description
path	string	BIP44 path
rawTx	string	RLP-encoded transaction (hex, with or without the <code>0x</code> prefix)

Example

```
const sig = await eth.signTransaction(  
  DEFAULT_ETH_PATH,  
  'f86c0a8502540be400825208...'  
)
```

signTypedData

Signs EIP-712 typed data. Pass the pre-hashed domainSeparator and message.

```
async signTypedData(  
  path: string,  
  domainSeparatorHash: string,  
  messageHash: string,  
  chainId?: number  
): Promise<SignatureResult>
```

Parameters

Name	Type	Default	Description
<code>path</code>	<code>string</code>	—	BIP44 path
<code>domainSeparatorHash</code>	<code>string</code>	—	EIP-712 domain separator hash (32 bytes hex)
<code>messageHash</code>	<code>string</code>	—	EIP-712 message hash (32 bytes hex)
<code>chainId</code>	<code>number</code>	<code>1</code>	EVM chain ID

Example

```
const sig = await eth.signTypedData(
  DEFAULT_ETH_PATH,
  'aabbccdde...', // domainSeparatorHash
  '1122334455...', // messageHash
  1
)
```

signTypedDataStreaming

Signs EIP-712 typed data by sending **the data itself instead of its hashes**, so the device can show what is actually being signed.

```
async signTypedDataStreaming(
  path: string,
  typedData: TypedDataLike,
  chainId?: number
): Promise<SignatureResult>
```

Why there are two

	What reaches the device	Device screen
<code>signTypedData</code>	Two hashes	Hashes only
<code>signTypedDataStreaming</code>	The typed data	The contents (supported types only)

`signTypedData` is enough when the dApp shows the same hash (Safe, for example) — the user compares the two and is done.

For a gasless stablecoin transfer (EIP-3009) the signer is an ordinary customer, and asking them to compare 64 hex characters is not realistic: they would sign without ever seeing the amount or the recipient. That is what this method is for.

Parameters

Name	Type	Default	Description
<code>path</code>	<code>string</code>	—	BIP44 path
<code>typedData</code>	<code>TypedDataLike</code>	—	<code>{ domain, types, primaryType, message }</code>
<code>chainId</code>	<code>number</code>	<code>1</code>	Chain ID

Notes

- **Openloop devices only.** Ledger devices throw
- **Nested structs and arrays are rejected.** EIP-3009 has neither, and partial support is where silent mistakes hide
- **About 25 APDU round trips.** Fine over USB; 0.5–1.2 s over BLE, so show progress
- The device renders the contents **only for types it supports**; anything else stays on the hash display (the signature still succeeds)

Example

```
import { isEip3009 } from '@openloop/sdk-core'

// Only route payloads worth showing through the streaming path
if (isEip3009(typedData)) {
  const sig = await eth.signTypedDataStreaming(DEFAULT_ETH_PATH, typedData, 137)
} else {
  const sig = await eth.signTypedData(DEFAULT_ETH_PATH, domainHash, messageHash, 137)
}
```

isEip3009

Returns whether typed data is EIP-3009 (`TransferWithAuthorization` / `ReceiveWithAuthorization`).

```
function isEip3009(typedData: TypedDataLike): boolean
```

It checks the type name, the field order, and the `message` contents, and returns `true` **only on an exact match**. One difference anywhere gives `false`.

The rule is **identical to the device's**. If the two disagreed, the app could promise a readable screen the device does not deliver, or the other way round.

This decides **which path to send on** — never whether a signature is allowed. Typed data that is not EIP-3009 still signs exactly as before.

signAuthorization

Signs an EIP-7702 authorization — Set Code for an EOA.

```
async signAuthorization(  
  path: string,  
  authorizationRlp: string  
): Promise<SignatureResult>
```

Parameters

Name	Type	Description
<code>path</code>	<code>string</code>	BIP44 path
<code>authorizationRlp</code>	<code>string</code>	Hex of <code>RLP([chain_id, address, nonce])</code>

Example

```
const sig = await eth.signAuthorization(  
  DEFAULT_ETH_PATH,  
  'c3010a...' // RLP([1, "0x...", 0])  
)
```

ethers.js Integration

```
import { ethers } from 'ethers'
import { EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const eth = new EthereumApp(transport)

// Get the address
const { address } = await eth.getAddress(DEFAULT_ETH_PATH)

// Provider (connection to an Ethereum node)
const provider = new ethers.JsonRpcProvider('https://eth.llamarpc.com')

// Build the transaction
const tx = {
  to: '0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045',
  value: ethers.parseEther('0.01'),
  gasLimit: 21000n,
  gasPrice: (await provider.getFeeData()).gasPrice,
  nonce: await provider.getTransactionCount(address),
  chainId: 1n,
}

// RLP-encode and sign
const unsignedTx = ethers.Transaction.from(tx).unsignedSerialized
const sig = await eth.signTransaction(DEFAULT_ETH_PATH, unsignedTx)

// Assemble the signed transaction
const signedTx = ethers.Transaction.from({
  ...tx,
  signature: {
    v: sig.v,
    r: '0x' + sig.r,
    s: '0x' + sig.s,
  },
})

// Broadcast
const txResponse = await provider.broadcastTransaction(signedTx.serialized)
```

viem Integration

```
import { createPublicClient, http, parseEther } from 'viem'
import { mainnet } from 'viem/chains'
import { EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const eth = new EthereumApp(transport)

const { address } = await eth.getAddress(DEFAULT_ETH_PATH)

// personal_sign
const sig = await eth.signPersonalMessage(
  DEFAULT_ETH_PATH,
  'Hello from viem!',
  1
)

// Convert to viem's signature format
const signature = `0x${sig.r}${sig.s}${sig.v.toString(16)}` as `0x${string}`
```

Next Steps

- [Bitcoin API](#) — PSBT signing
- [Error Handling](#) — handling rejections, busy devices and similar cases

XRP API

The `XrpApp` class provides XRP Ledger address retrieval and transaction signing. It supports both the Ed25519 (default) and secp256k1 elliptic curves.

Key Differences from Ethereum

Aspect	Ethereum	XRP
Address format	<code>0x...</code> (hex)	<code>r...</code> (Base58Check)
Default curve	secp256k1	Ed25519
Ed25519 public key	—	Carries a <code>0xED</code> prefix
Signature format	v/r/s	raw bytes (Ed25519: 64B, secp256k1: DER)
BIP44 path	<code>44'/60'/...</code>	<code>44'/144'/...</code> (Ed25519: fully hardened)

Import

```
import {
  XrpApp,
  DEFAULT_XRP_PATH,
  type XrpSignatureResult,
  type XrpCurve,
} from '@openloop/sdk-core'
```

Initialization

```
const xrp = new XrpApp(transport)
```

Methods

Method	Description
<code>getAddress(path, curve?)</code>	Get the XRP address and the public key
<code>signTransaction(path, txBlob, curve?)</code>	Sign a serialized transaction

getAddress

Get the XRP Ledger address from the device.

```
async getAddress(  
  path: string,  
  curve?: XrpCurve  
): Promise<{  
  publicKey: string // Public key (hex, Ed25519: 33B with 0xED prefix, secp256k1: 33B compressed)  
  address: string // XRP address ("r...")  

```

Parameters

Name	Type	Default	Description
<code>path</code>	<code>string</code>	—	BIP44 path
<code>curve</code>	<code>XrpCurve</code>	<code>'ed25519'</code>	<code>'ed25519'</code> or <code>'secp256k1'</code>

`XrpCurve` Type

```
type XrpCurve = 'ed25519' | 'secp256k1'
```

Default Path

`DEFAULT_XRP_PATH = "44'/144'/0'/0'/0'"` (Ed25519: every element hardened)

Example

```
// Ed25519 (the default)
const { address, publicKey } = await xrp.getAddress(DEFAULT_XRP_PATH)
// address: "rHb9CJAWyB4rj91VRWn96DkukG4bwdtyTh"
// publicKey: "ed01a2b3c4..." (0xED prefix)

// secp256k1
const { address, publicKey } = await xrp.getAddress(
  "44'/144'/0'/0/0", // secp256k1 also accepts a non-hardened path
  'secp256k1'
)
```

Curve Selection and P2 Values

Curve	P2 value	Description
'ed25519'	0x80	Ed25519 (default, recommended)
'secp256k1'	0x40	secp256k1 (Ethereum-compatible)

signTransaction

Sign a serialized XRP transaction.

```
async signTransaction(
  path: string,
  txBlob: string,
  curve?: XrpCurve
): Promise<XrpSignatureResult>
```

Parameters

Name	Type	Description
<code>path</code>	<code>string</code>	BIP44 path
<code>txBlob</code>	<code>string</code>	Serialized transaction (hex, with or without <code>0x</code>)
<code>curve</code>	<code>XrpCurve</code>	Curve (defaults to <code>'ed25519'</code>). Following the Ledger-compatible specification, it is sent in P2 as a <code>curveMask</code> (<code>secp256k1=0x40</code> / <code>ed25519=0x80</code>). Match the curve you used with <code>getAddress</code>

Returns: `XrpSignatureResult`

Field	Type	Description
<code>signature</code>	<code>string</code>	Signature as hex (Ed25519: 64 bytes, secp256k1: variable-length DER encoding)

Example

```
// Ed25519 (the default when omitted)
const { signature } = await xrp.signTransaction(
  DEFAULT_XRP_PATH,
  '120000228...' // serialized transaction blob
)

// secp256k1 (pass the same value you used with getAddress if you chose 'secp256k1')
const { signature: sig2 } = await xrp.signTransaction(
  DEFAULT_XRP_PATH,
  '120000228...',
  'secp256k1'
)
```

xrpl.js Integration

```
import { Client, Wallet, encode, decode } from 'xrpl'
import { XrpApp, DEFAULT_XRP_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const xrp = new XrpApp(transport)

// Get the address
const { address, publicKey } = await xrp.getAddress(DEFAULT_XRP_PATH)

// Connect to the XRP Ledger
const client = new Client('wss://xrplcluster.com/')
await client.connect()

// Build the transaction
const prepared = await client.autofill({
  TransactionType: 'Payment',
  Account: address,
  Amount: '1000000', // 1 XRP = 1,000,000 drops
  Destination: 'rPT1Sjq2YGrBMTttX4GZHjKu9dyfzbpAYe',
})

// Serialize and sign on the device
const txBlob = encode(prepared)
const { signature } = await xrp.signTransaction(DEFAULT_XRP_PATH, txBlob)

// Add the signature to the transaction
const signedTx = {
  ...prepared,
  TxnSignature: signature.toUpperCase(),
  SigningPubKey: publicKey.toUpperCase(),
}

// Broadcast
const result = await client.submitAndWait(encode(signedTx))
console.log('TX Result:', result.result.meta.TransactionResult)

await client.disconnect()
```

Next Steps

- WalletConnect Integration
- Error Handling

Solana API

The `SolanaApp` class provides Solana address retrieval, transaction signing, and off-chain message signing. It uses an APDU protocol compatible with the Ledger Solana app.

Import

```
import {
  SolanaApp,
  DEFAULT_SOL_PATH,
} from '@openloop/sdk-core'
```

Initialization

```
const sol = new SolanaApp(transport)
```

Methods

Method	Description
<code>getAddress(path)</code>	Get the Ed25519 public key and the Base58 address
<code>signTransaction(path, txBytes)</code>	Sign a transaction
<code>signOffchainMessage(path, messageBytes)</code>	Sign an off-chain message

getAddress

Get the Solana address — the Base58 encoding of the Ed25519 public key — from the device.

```

async getAddress(path: string): Promise<{
  publicKey: string // Ed25519 public key (hex, 32 bytes)
  address: string    // Base58 address
}>

```

Parameters

Name	Type	Description
path	string	BIP44 path (e.g. "44'/501'/0'/0'")

Default Path

```
DEFAULT_SOL_PATH = "44'/501'/0'/0'"
```

Note: Solana uses Ed25519, so every element of the path is hardened.

Example

```

const { address, publicKey } = await sol.getAddress(DEFAULT_SOL_PATH)
// address: "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU"

```

signTransaction

Produce an Ed25519 signature over the binary data of a Solana transaction.

```

async signTransaction(
  path: string,
  txBytes: Uint8Array
): Promise<string> // 64-byte Ed25519 signature (hex)

```

Parameters

Name	Type	Description
<code>path</code>	<code>string</code>	BIP44 path
<code>txBytes</code>	<code>Uint8Array</code>	Serialized transaction (raw binary)

Returns

A 64-byte Ed25519 signature as a hex string.

Example

```
const signatureHex = await sol.signTransaction(  
  DEFAULT_SOL_PATH,  
  transactionBytes  
)
```

Note: Large transactions are automatically split and sent using the Ledger P2 chunking protocol.

signOffchainMessage

Sign an off-chain message, such as Sign-In With Solana.

```
async signOffchainMessage(  
  path: string,  
  messageBytes: Uint8Array  
): Promise<string> // 64-byte Ed25519 signature (hex)
```

Parameters

Name	Type	Description
<code>path</code>	<code>string</code>	BIP44 path
<code>messageBytes</code>	<code>Uint8Array</code>	Raw bytes of the message

Example

```
const message = new TextEncoder().encode('Hello, Solana!')
const signatureHex = await sol.signOffchainMessage(DEFAULT_SOL_PATH, message)
```

@solana/web3.js Integration

```
import {
  Connection,
  PublicKey,
  SystemProgram,
  Transaction,
} from '@solana/web3.js'
import { SolanaApp, DEFAULT_SOL_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const sol = new SolanaApp(transport)

// Get the address
const { address } = await sol.getAddress(DEFAULT_SOL_PATH)
const publicKey = new PublicKey(address)

// Build the transaction
const connection = new Connection('https://api.mainnet-beta.solana.com')
const recentBlockhash = (await connection.getLatestBlockhash()).blockhash

const tx = new Transaction()
tx.add(
  SystemProgram.transfer({
    fromPubkey: publicKey,
    toPubkey: new PublicKey('...'),
    lamports: 1000000, // 0.001 SOL
  })
)
tx.recentBlockhash = recentBlockhash
tx.feePayer = publicKey

// Sign on the device
const txBytes = tx.serializeMessage()
const signatureHex = await sol.signTransaction(DEFAULT_SOL_PATH, txBytes)

// Add the signature to the transaction
const signatureBytes = Uint8Array.from(
  signatureHex.match(/.{2}/g)!.map(b => parseInt(b, 16))
)
tx.addSignature(publicKey, Buffer.from(signatureBytes))

// Broadcast
const txId = await connection.sendRawTransaction(tx.serialize())
console.log('TX:', txId)
```

| Next Steps

- TRON API
- XRP API

TRON API

The `TronApp` class provides TRON address retrieval, transaction signing, and message signing. It uses the same INS codes as Ethereum, but `coin_type=195` in the BIP44 path puts the firmware into TRON mode.

Key Differences from Ethereum

Aspect	Ethereum	TRON
Address format	<code>0x...</code> (hex)	<code>T...</code> (Base58Check)
TX hash	Keccak-256	SHA-256
Signature response	$V(1) + R(32) + S(32)$	$R(32) + S(32) + V(1)$
V value	27 or 28	0 or 1 (recovery ID)

Import

```
import {
  TronApp,
  DEFAULT_TRON_PATH,
  type SignatureResult,
} from '@openloop/sdk-core'
```

Initialization

```
const tron = new TronApp(transport)
```

Methods

Method	Description
<code>getAddress(path)</code>	Get the TRON address (<code>T...</code>) and the public key
<code>signTransaction(path, rawDataHex)</code>	Sign the <code>raw_data</code> of a transaction
<code>signMessage(path, message)</code>	Sign a message

getAddress

Get the TRON address — Base58Check format, `T...` — from the device.

```
async getAddress(path: string): Promise<{
  publicKey: string // Uncompressed public key (hex, 65 bytes)
  address: string   // Base58Check address ("T...")
}>
```

Parameters

Name	Type	Description
<code>path</code>	<code>string</code>	BIP44 path (e.g. <code>"44'/195'/0'/0/0"</code>)

Default Path

```
DEFAULT_TRON_PATH = "44'/195'/0'/0/0"
```

Example

```
const { address } = await tron.getAddress(DEFAULT_TRON_PATH)
// address: "TLsV62...txnV1Fm"
```

signTransaction

Sign the `raw_data` of a TRON transaction.

```

async signTransaction(
  path: string,
  rawDataHex: string
): Promise<SignatureResult>

```

Parameters

Name	Type	Description
path	string	BIP44 path
rawDataHex	string	Hex of the protobuf-encoded <code>raw_data</code> (with or without <code>0x</code>)

Returns: `SignatureResult`

Field	Type	Description
v	number	Recovery ID (0 or 1)
r	string	R value (32 bytes hex)
s	string	S value (32 bytes hex)

Example

```

const sig = await tron.signTransaction(
  DEFAULT_TRON_PATH,
  'a1b2c3d4...' // raw_data hex
)

```

signMessage

Sign a TRON message. The firmware reads `coin_type=195'` from the BIP44 path and applies the TRON message format.

```

async signMessage(
  path: string,
  message: string
): Promise<SignatureResult>

```

Parameters

Name	Type	Description
path	string	BIP44 path
message	string	UTF-8 message string

Example

```
const sig = await tron.signMessage(DEFAULT_TRON_PATH, 'Hello, TRON!')
```

TronWeb Integration

```
import TronWeb from 'tronweb'
import { TronApp, DEFAULT_TRON_PATH } from '@openloop/sdk-core'
import { WebHidTransport } from '@openloop/transport-webhid'

const transport = await WebHidTransport.connect()
const tron = new TronApp(transport)

// Get the address
const { address } = await tron.getAddress(DEFAULT_TRON_PATH)

// Build the transaction with TronWeb
const tronWeb = new TronWeb({ fullHost: 'https://api.trongrid.io' })

const transaction = await tronWeb.transactionBuilder.sendTrx(
  'TLsV62...', // Recipient
  1000000,      // 1 TRX = 1,000,000 SUN
  address
)

// Convert raw_data to hex and sign on the device
const rawDataHex = Buffer.from(
  JSON.stringify(transaction.raw_data)
).toString('hex')

const sig = await tron.signTransaction(DEFAULT_TRON_PATH, rawDataHex)

// Add the signature to the transaction
const signedTx = {
  ...transaction,
  signature: [`${sig.r}${sig.s}0${sig.v}`],
}

// Broadcast
const result = await tronWeb.trx.sendRawTransaction(signedTx)
```

Next Steps

- XRP API
- Error Handling

WalletConnect Integration

The Openloop SDK supports remote device access over WalletConnect v2 in two ways.

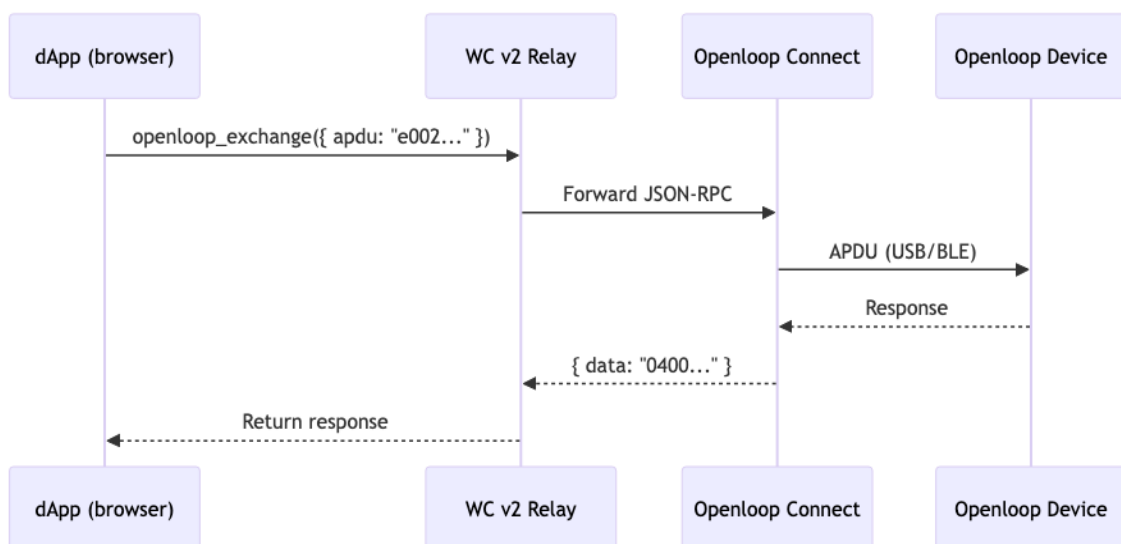
The Two Approaches

Approach	Level	Description
WcTransport	Low-level	Relays APDU commands through the WalletConnect relay
AppKit	High-level	Standard multi-chain WC connection via Reown AppKit

WcTransport (APDU Relay)

`WcTransport` implements `ITransport` and sends APDU commands to a remote Openloop Connect app through the WalletConnect v2 relay.

How It Works



Hex-encoded APDUs are sent and received over the custom JSON-RPC method `openloop_exchange`.

Setup

```
import { WcTransport, EthereumApp, DEFAULT_ETH_PATH } from '@openloop/sdk-core'
import SignClient from '@walletconnect/sign-client'

// Initialize the WalletConnect SignClient
const client = await SignClient.init({
  projectId: 'your-project-id',
})

// Pair and establish a session
const { uri, approval } = await client.connect({
  requiredNamespaces: {
    eip155: {
      methods: ['openloop_exchange', 'openloop_lock', 'openloop_unlock'],
      chains: ['eip155:1'],
      events: [],
    },
  },
})

// Show the URI as a QR code → the user scans it with Openloop Connect
console.log('Pairing URI:', uri)

// Wait for session approval
const session = await approval()

// Create the WcTransport
const transport = new WcTransport({
  client,
  session: { topic: session.topic },
  chainId: 'eip155:1', // optional (default: 'eip155:1')
})

await transport.open()

// The regular App classes work as-is
const eth = new EthereumApp(transport)
const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
```

WcTransport Type Definitions

```
interface WcTransportOptions {
  client: IWcSignClient // WalletConnect SignClient instance
  session: WcSession     // Active session { topic: string }
  chainId?: string       // Chain ID (default: "eip155:1")
}

interface IWcSignClient {
  request<T>(params: {
    topic: string
    chainId: string
    request: { method: string; params: unknown }
  }): Promise<T>
}

interface WcSession {
  topic: string
}
```

Supported Methods

Method	Description
<code>openloop_exchange</code>	Send and receive APDU commands
<code>openloop_lock</code>	Acquire the exclusive device lock
<code>openloop_unlock</code>	Release the exclusive device lock

Key Features

- Signing works even in environments with no direct access to the device
- Implements the `ITransport` interface, so every App class works as-is
- Device locking (`lock` / `unlock`) also works over the relay

AppKit (High-Level WC)

With Reown AppKit you sign using WalletConnect's standard methods — `eth_signTransaction`, `personal_sign`, and so on. This approach never handles APDUs directly; Openloop Connect drives the device internally.

Supported Chains and Methods

EVM (eip155)

Chains: - `eip155:1` — Ethereum Mainnet - `eip155:11155111` — Sepolia Testnet

Methods: | Method | Description | |———|——| | `eth_sendTransaction` | Send a transaction |
| `eth_signTransaction` | Sign a transaction | | `eth_sign` | Sign data | | `personal_sign` |
Sign an EIP-191 message | | `eth_signTypedData` | Sign EIP-712 typed data | |
`eth_signTypedData_v3` | EIP-712 v3 | | `eth_signTypedData_v4` | EIP-712 v4 | |
`wallet_getCapabilities` | Get the wallet's capabilities |

Bitcoin (bip122)

Chains: - `bip122:000000000019d6689c085ae165831e93` — Bitcoin Mainnet -
`bip122:0000000000933ea01ad0ee984209779ba` — Bitcoin Testnet3

Methods: | Method | Description | |———|——| | `signPsbt` | Sign a PSBT | |
`getAccountAddresses` | Get the account addresses | | `signMessage` | Sign a message | |
`sendTransfer` | Send a transfer |

Solana

Chains: - `solana:5eykt4UsFv8P8NJdTREpY1vzqKqZKvdp` — Mainnet Beta -
`solana:EtWTRABZaYq6iMfeYKouRu166VU2xqa1` — Devnet

Methods: | Method | Description | |———|——| | `solana_signTransaction` | Sign a
transaction | | `solana_signAllTransactions` | Sign multiple transactions | |
`solana_signAndSendTransaction` | Sign and send | | `solana_signMessage` | Sign a message |

TRON

Chains: - `tron:0x2b6653dc` — TRON Mainnet - `tron:0x94a9059e` — TRON Shasta Testnet

Methods: | Method | Description | |———|——| | `tron_signTransaction` | Sign a transaction |
| `tron_signMessage` | Sign a message |

AppKit Configuration Example

```
import { createAppKit } from '@reown/appkit/react'
import { WagmiAdapter } from '@reown/appkit-adapter-wagmi'
import { SolanaAdapter } from '@reown/appkit-adapter-solana'
import { BitcoinAdapter } from '@reown/appkit-adapter-bitcoin'
import { TronAdapter } from '@reown/appkit-adapter-tron'
import { mainnet, sepolia } from '@reown/appkit/networks'

const projectId = 'your-walletconnect-project-id'

// Create the adapters
const wagmiAdapter = new WagmiAdapter({ projectId, networks: [mainnet, sepolia] })
const solanaAdapter = new SolanaAdapter()
const bitcoinAdapter = new BitcoinAdapter()
const tronAdapter = new TronAdapter()

// Create the AppKit instance
const modal = createAppKit({
  projectId,
  adapters: [wagmiAdapter, solanaAdapter, bitcoinAdapter, tronAdapter],
  networks: [mainnet, sepolia],
  metadata: {
    name: 'My dApp',
    description: 'My dApp with Openloop',
    url: 'https://my-dapp.com',
    icons: ['https://my-dapp.com/icon.png'],
  },
  customWallets: [
    {
      id: 'openloop-connect',
      name: 'Openloop Connect',
      homepage: 'https://crypto.haudi.jp/openloop/',
      mobile_link: 'openloop://',
      desktop_link: 'openloop://',
      image_url: 'https://crypto.haudi.jp/openloop/images/logo/openloop-icon.png',
    },
  ],
})
```

Choosing Between WcTransport and AppKit

Criterion	WcTransport	AppKit
APDU-level control	✓ Full control	✗ Abstracted away
Ease of adoption	Somewhat involved	Easy
UI components	None (build your own)	Modal included
Multi-chain	Manual switching	Automatic switching
EIP-7702	✓ Supported	✗ Not supported
Supported wallets	Openloop Connect only	Every WC-compatible wallet

Recommendation: - Integrating with the Openloop device is the main goal → **WcTransport** -
You want to support multiple wallets → **AppKit**

Limitations

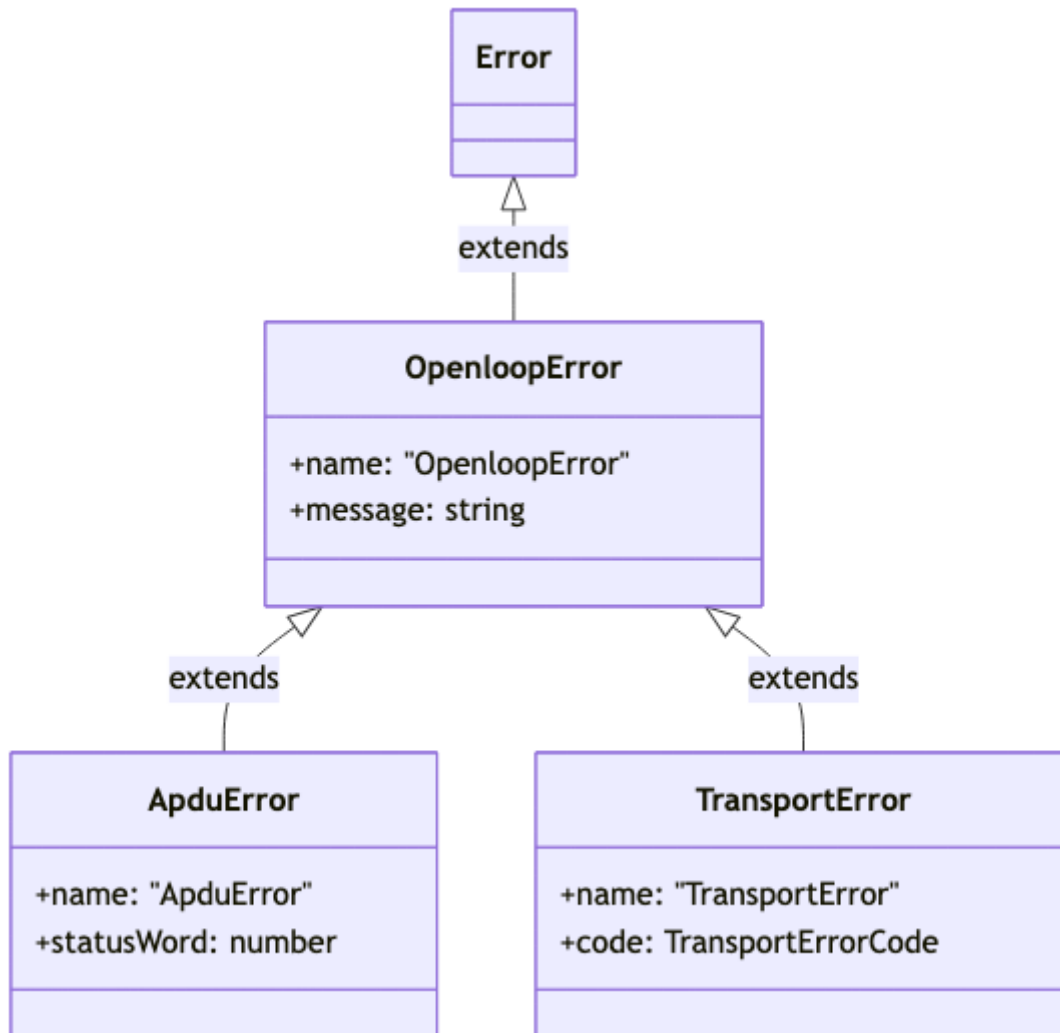
- WalletConnect always goes through a remote Openloop Connect app, so the device must be connected to that app
- EIP-7702 authorization is supported only over WcTransport — AppKit does not support it
- XRP is not currently supported over WalletConnect
- The `exchange` timeout of `WcTransport` depends on the constraints of the WalletConnect relay

Next Steps

- [Error Handling](#)
- [API Reference](#)

Error Handling Guide

Error Class Hierarchy



Import

```
import { OpenloopError, ApduError, TransportError } from '@openloop/sdk-core'
```

ApduError

Thrown when the device returns an error status word — anything other than `0x9000`.

```
class AduError extends OpenloopError {
  readonly statusWord: number // e.g. 0x6985
}
```

APDU Status Word Table

Status word	Constant name	Description	Common cause
0x9000	—	Success	—
0x6985	Conditions not satisfied	Rejected, or the device is busy	The user pressed the reject button, or a request arrived while another confirmation was on screen (answered immediately on device firmware v1.1.5 and later)
0x6a80	Invalid data	Invalid data	A malformed BIP44 path or transaction
0x6a82	App not found	App not found	The matching app is not open on the device
0x6d00	Instruction not supported	Instruction not supported	The device firmware version is too old
0x6e00	CLA not supported	CLA not supported	Invalid class byte
0x6f00	Internal error	Internal error	An unexpected error inside the device
0x61XX	More data	More data available	Intermediate response while fetching a PSBT (not an error)

Example

```
import { ApduError } from '@openloop/sdk-core'

try {
  const sig = await eth.signPersonalMessage(DEFAULT_ETH_PATH, 'Hello')
} catch (err) {
  if (err instanceof ApduError) {
    if (err.statusWord === 0x6985) {
      console.log('Rejected on the device, or the device is busy. Check the screen and try again')
    } else {
      console.log(`APDU error: 0x${err.statusWord.toString(16)}`)
    }
  }
}
```

TransportError

Thrown when a communication error occurs in the transport layer — USB, BLE, WebSocket, and so on.

```
class TransportError extends OpenloopError {
  readonly code: TransportErrorCode // added in v0.6.0
  // message carries English detail – for logs, not for display
}
```

Branch on `code` (added in v0.6.0)

`message` is free-form English, so **do not branch on the string**. Branch on `code`.

code	Meaning	What to do
not-supported	This runtime cannot talk to the device at all (no WebHID / Web Bluetooth)	Point the user at a supported browser
not-found	No device matched, or the chooser was dismissed	Ask the user to pick again
not-connected	There is no open connection (never opened, or already closed)	Call <code>open()</code> before using it
timeout	The device did not answer in time	Ask the user to check the device — it may be waiting for approval
disconnected	The link dropped while in use	Try reconnecting
io	Anything else (framing, write failure, unexpected payload)	Log <code>message</code>

`code` defaults to `io`. Handle unknown values in a `default` branch so new codes cannot break you.

Errors the device returned are `ApduError`

In v0.6.0 the BLE, Safari and USB transports stopped **discarding the status word** and now throw `ApduError`. Previously these surfaced as `TransportError`, so callers could not branch on `statusWord`. Handle device-side failures (rejection, conditions not satisfied, and so on) in the `ApduError` branch.

| Error Handling Patterns

The Basic Pattern

```

import { OpenloopError, ApduError, TransportError } from '@openloop/sdk-core'

try {
  const { address } = await eth.getAddress(DEFAULT_ETH_PATH)
} catch (err) {
  if (err instanceof ApduError) {
    // Error response from the device
    switch (err.statusWord) {
      case 0x6985:
        showMessage('Rejected on the device, or the device is busy. Check the screen and try again')
        break
      case 0x6a80:
        showMessage('Invalid data')
        break
      default:
        showMessage(`Device error: ${err.message}`)
    }
  } else if (err instanceof TransportError) {
    // Communication error – branch on code, not on message
    switch (err.code) {
      case 'not-supported':
        showMessage('This browser is not supported.')
        break
      case 'not-found':
        showMessage('No device was found.')
        break
      case 'timeout':
        showMessage('The device is not responding. Please check its screen.')
        break
      case 'disconnected':
      case 'not-connected':
        showMessage('Lost the connection to the device. Please reconnect.')
        break
      default:
        showMessage(`Communication error: ${err.message}`)
    }
  } else if (err instanceof OpenloopError) {
    // Any other SDK error
    showMessage(`Error: ${err.message}`)
  } else {
    // Unexpected error
    throw err
  }
}

```

Retry Pattern With Reconnect

```
async function withRetry<T>(  
  fn: () => Promise<T>,  
  reconnect: () => Promise<void>,  
  maxRetries: number = 1  
) : Promise<T> {  
  for (let i = 0; i <= maxRetries; i++) {  
    try {  
      return await fn()  
    } catch (err) {  
      if (err instanceof TransportError && i < maxRetries) {  
        await reconnect()  
        continue  
      }  
      throw err  
    }  
  }  
  throw new Error('Unreachable')  
}  
  
// Usage example  
const { address } = await withRetry(  
  () => eth.getAddress(DEFAULT_ETH_PATH),  
  async () => {  
    transport = await WebHidTransport.reconnect()  
    eth = new EthereumApp(transport!)  
  }  
)
```

Detecting a Rejection or a Busy Device

```
function isRejectedOrBusy(err: unknown): boolean {
  return err instanceof ApduError && err.statusWord === 0x6985
}

try {
  const sig = await eth.signTransaction(DEFAULT_ETH_PATH, rawTx)
} catch (err) {
  if (isRejectedOrBusy(err)) {
    // Rejected or busy – check the screen and try again
    showMessage('Rejected on the device, or the device is busy. Check the screen and try again')
    return
  }
  // Show every other error
  showError(err)
}
```

Next Steps

- [API Reference](#) — every type definition

API Reference

A complete list of type definitions, interfaces, classes, and constants. Unless stated otherwise, everything here is exported from `@openloop/sdk-core`.

Contents

- Interfaces
- App Classes
- Signature Verification — `@openloop/sdk-core/verify`
- Error Classes
- OpenloopSDK Class
- WcTransport Class
- Constants
- Utility Functions

Interfaces

ITransport

The interface that abstracts communication with the device. Every transport implements it.

```
interface ITransport {  
  open(): Promise<void>  
  close(): Promise<void>  
  isConnected(): boolean  
  exchange(apdu: Uint8Array): Promise<Uint8Array>  
  lock(): Promise<void>  
  unlock(): Promise<void>  
}
```

Method	Description
<code>open()</code>	Open the connection to the device
<code>close()</code>	Close the connection
<code>isConnected()</code>	Whether a connection is active
<code>exchange(apdu)</code>	Send an APDU command and receive the response
<code>lock()</code>	Acquire exclusive access to the device (for multi-step operations)
<code>unlock()</code>	Release exclusive access

TransportStatus

```
interface TransportStatus {
  connected: boolean
  deviceName?: string
  deviceId?: string
}
```

TransportEvent

```
type TransportEventType = 'connected' | 'disconnected' | 'error'

interface TransportEvent {
  type: TransportEventType
  error?: Error
}

type TransportEventHandler = (event: TransportEvent) => void
```

TransportInfo

The transport information returned by `OpenloopSDK.discover()`.

```
interface TransportInfo {
  type: string // Registered name (e.g. 'webhid')
  name: string // Display name (e.g. 'WebHID (USB)')
  available: boolean // Whether it is available
}
```

TransportFactory

```
type TransportFactory = () => Promise<ITransport>
```

SignatureResult

The ECDSA signature result for Ethereum / TRON.

```
interface SignatureResult {  
  v: number    // Recovery ID  
  r: string    // R value (32 bytes hex, no 0x prefix)  
  s: string    // S value (32 bytes hex, no 0x prefix)  
}
```

BtcMessageSignature

The Bitcoin BIP-137 message signature result.

```
interface BtcMessageSignature {  
  v: number    // Recovery ID (35-38: P2WPKH native SegWit)  
  r: string    // R value (32 bytes hex)  
  s: string    // S value (32 bytes hex)  
  signature: string // Base64 of the full signature (V || R || S, 65 bytes)  
}
```

InputSigningPath

The per-input BIP32 path for a Bitcoin PSBT.

```
interface InputSigningPath {  
  index: number // Input index within the PSBT  
  path: string  // BIP32 path (e.g. "84'/1'/0'/0/5")  
}
```

DeviceInfo

```
interface DeviceInfo {
  connected: boolean
  path?: string
  vendorId?: number
  productId?: number
}
```

XrpSignatureResult

The XRP signature result.

```
interface XrpSignatureResult {
  signature: string // Signature hex (Ed25519: 64B, secp256k1: DER, variable length)
}
```

XrpCurve

```
type XrpCurve = 'ed25519' | 'secp256k1'
```

WcSession

```
interface WcSession {
  topic: string
}
```

IWcSignClient

```
interface IWcSignClient {
  request<T>(params: {
    topic: string
    chainId: string
    request: { method: string; params: unknown }
  }): Promise<T>
}
```

WcTransportOptions

```
interface WcTransportOptions {  
  client: IWcSignClient  
  session: WcSession  
  chainId?: string // Default: "eip155:1"  
}
```

App Classes

EthereumApp

```
class EthereumApp {  
  constructor(transport: ITransport)  
  
  getAddress(path: string): Promise<{ publicKey: string; address: string }>  
  
  signPersonalMessage(path: string, message: string, chainId?: number): Promise<SignatureResult>  
  
  signTransaction(path: string, rawTx: string): Promise<SignatureResult>  
  
  signTypedData(  
    path: string, domainSeparatorHash: string, messageHash: string, chainId?: number  
  ): Promise<SignatureResult>  
  
  signAuthorization(path: string, authorizationRlp: string): Promise<SignatureResult>  
}
```

BitcoinApp

```
class BitcoinApp {
  constructor(transport: ITransport)

  getAddress(testnet?: boolean): Promise<{ publicKey: string; address: string; chainCode: string }>
  getAddressWithPath(path: string): Promise<{ publicKey: string; address: string; chainCode: string }>
  getAccountXpub(coinType: number): Promise<{ publicKey: string; chainCode: string }>
  getMasterFingerprint(): Promise<string> // hash160(master pubkey)[:4], 8-char hex (e.g. "6310")
  signMessage(message: string, path: string): Promise<BtcMessageSignature>
  signPsbt(psbt: Uint8Array | string): Promise<Uint8Array>
  signPsbtHex(psbt: string): Promise<string>
  signPsbtWithPaths(psbt: string, inputPaths: InputSigningPath[]): Promise<string>
}
```

SolanaApp

```
class SolanaApp {
  constructor(transport: ITransport)

  getAddress(path: string): Promise<{ publicKey: string; address: string }>
  signTransaction(path: string, txBytes: Uint8Array): Promise<string>
  signOffchainMessage(path: string, messageBytes: Uint8Array): Promise<string>
}
```

TronApp

```
class TronApp {
  constructor(transport: ITransport)

  getAddress(path: string): Promise<{ publicKey: string; address: string }>

  signTransaction(path: string, rawDataHex: string): Promise<SignatureResult>

  signMessage(path: string, message: string): Promise<SignatureResult>
}
```

XrpApp

```
class XrpApp {
  constructor(transport: ITransport)

  getAddress(path: string, curve?: XrpCurve): Promise<{ publicKey: string; address: string }>

  signTransaction(path: string, txBlob: string): Promise<XrpSignatureResult>
}
```

OpenloopApp

The class for utility commands specific to the Openloop firmware. It provides the Openloop-specific commands exposed under CLA=0xF0 (CLA_OPENLOOP).

```
class OpenloopApp {
  constructor(transport: ITransport)

  /**
   * Sends GET_DEVICE_INFO (F0 A0) and retrieves identity, state, and feature information.
   *
   * Return value:
   *   - Returns null for older Openloop firmware (earlier than v0.91.13) and for
   *     devices that do not support the command
   *   - Throws on transport-layer errors
   */
  getDeviceInfo(): Promise<OpenloopDeviceInfo | null>
}
```

OpenloopDeviceInfo

```
interface OpenloopDeviceInfo {  
  modelId: number // Hardware SKU ID (e.g. 0x0001 = Openloop V1)  
  hwRevision: number // PCB revision  
  mainFwVersion: { major: number; minor: number; patch: number }  
  recoveryFwVersion: { major: number; minor: number; patch: number }  
  state: number // 16-bit state bitmask (raw value)  
  stateBits: OpenloopDeviceState // Decoded view of state  
  features: number // 16-bit feature bitmask (implementation detail, subject to change)  
}
```

Field	Type	Description
modelId	number	Hardware SKU ID. Increments with every new SKU
hwRevision	number	PCB revision number
mainFwVersion	{major,minor,patch}	Version of the main firmware currently running
recoveryFwVersion	{major,minor,patch}	Version of the recovery (factory) partition. All zeros when it cannot be read
state	number	16-bit bitmask representing the current state (raw value)
stateBits	OpenloopDeviceState	<code>state</code> decoded into an object
features	number	16-bit bitmask representing the hardware features (raw value). The bit layout is an implementation detail of the firmware and may change between versions. Host apps should only store and forward the raw value

OpenloopDeviceState

The `state` field decoded into a human-readable object.

```
interface OpenloopDeviceState {
  walletProvisioned: boolean
  language: number           // 0=English, 1=Japanese, 2-7=reserved
  usbEnabled: boolean
  usbVidCompat: boolean      // false=Native (0x303A), true=Compatible (0x2C97)
  bleEnabled: boolean
  blePaired: boolean
  fidoEnabled: boolean
  pivEnabled: boolean
}
```

state bitmask layout (finalized in FW v0.91.14+):

bit	Name	Meaning
0	WALLET_PROVISIONED	A wallet has been created
1-3	LANG (3 bit)	Language index (0=EN, 1=JA, ...)
4	USB_ENABLED	USB communication on
5	USB_VID_COMPAT	USB VID = compatible mode (0x2C97). Native (0x303A) when 0
6	BLE_ENABLED	BLE communication on
7	BLE_PAIRIED	A paired BLE device exists
8	FIDO_ENABLED	FIDO / passkey feature on
9	PIV_ENABLED	PIV / PKCS#11 feature on
10-15	reserved	Reserved for future expansion, currently 0

More bits may be added in the future. To read a bit that `stateBits` does not expose, mask the raw `state` value directly.

Example

```
import { OpenloopApp, formatOpenloopState } from '@openloop/sdk-core'

const app = new OpenloopApp(transport)
const info = await app.getDeviceInfo()

if (info === null) {
  console.log('GET_DEVICE_INFO not supported (old firmware)')
} else {
  console.log(`Model: 0x${info.modelId.toString(16).padStart(4, '0')}`)
  console.log(`FW: ${info.mainFwVersion.major}.${info.mainFwVersion.minor}.${info.mainFwVersion.patch}`)
  console.log(`State: ${formatOpenloopState(info.stateBits)}`)
  if (!info.stateBits.walletProvisioned) {
    // Branch for when no wallet has been created yet
  }
}
```

Signature Verification

The `@openloop/sdk-core/verify` subpath. Utilities that verify a signature returned by the device against a reference implementation (viem / @noble). Transport-independent — USB, BLE, AirGap, and WalletConnect all work. It is not part of the base `@openloop/sdk-core` (viem is never forced on you); it ships as a subpath instead.

```
import { verifyPersonalSign } from '@openloop/sdk-core/verify'

const result = verifyPersonalSign(message, sig, address)
// result: { recovered: string; expected?: string; ok: boolean }
```

Function	Chain	Verifies
<code>verifyPersonalSign(message, sig, expected?)</code>	EVM	EIP-191 personal_sign
<code>verifyTypedDataJson(typedDataJson, sig, expected?)</code>	EVM	EIP-712 (full JSON)
<code>verifyTypedDataHashes(domainHash, msgHash, sig, expected?)</code>	EVM	EIP-712 (pre-hashed)
<code>verifyTransaction(unsignedTxHex, sig, expected?)</code>	EVM	legacy / EIP-2718 typed tx (takes <code>{v,r,s}</code>)
<code>verifySignedTransaction(signedTxHex, expected?)</code>	EVM	Recovers the signer from a signed serialized tx (the return value of <code>WalletConnect.eth_signTransaction</code>)
<code>verifyAuth7702(authRlpHex, sig, expected?)</code>	EVM	EIP-7702 authorization
<code>verifyBtcMessage(message, sig, identity)</code>	BTC	BIP-137 message (<code>identity</code> = pubkey hex or BTC address)
<code>verifyBtcPsbt(signedPsbtHex)</code>	BTC	Signed PSBT (BIP-143 P2WPKH; the pubkey comes from the PSBT)
<code>verifySolMessage(message, sigHex, pubkeyHex)</code>	SOL	ed25519 off-chain message
<code>verifySolTransaction(txHex, sigHex, pubkeyHex)</code>	SOL	ed25519 transaction
<code>verifyTronMessage(message, sig, identity)</code>	TRON	keccak256 TRON prefix (<code>identity</code> = pubkey hex or TRON address)
<code>verifyTronTransaction(rawDataHex, sig, identity)</code>	TRON	sha256(raw_data) (<code>identity</code> = pubkey hex or TRON address)
<code>verifyXrpTransaction(txBlobHex, curve, result, pubkeyHex)</code>	XRP	ed25519 / secp256k1 (for-signing blob + explicit pubkey)
<code>verifyXrpSignedTx(signedTxHex)</code>	XRP	Self-contained verification from a signed tx blob (extracts <code>SigningPubKey</code> / <code>TxnSignature</code> ; no pubkey needed)

Every one of them returns a `VerifyResult`: `{ recovered: string; expected?: string; ok: boolean }`. Pass `expected` and the check goes one step further, reporting whether the recovered signer matches (`ok`).

About `identity` (pubkey or address): for BTC / TRON the shape of `identity` decides automatically whether to compare the public key recovered from the signature directly, or to derive a chain address from the recovered key and compare that. On routes where no public key is available — WalletConnect, for example — you can **match by address**. `verifySignedTransaction` (EVM) and `verifyXrpSignedTx` (XRP) recover the signer / public key from the signed data itself, so there is no pubkey to pass in from outside.

Error Classes

```
class OpenloopError extends Error {
  name: 'OpenloopError'
}

class AduError extends OpenloopError {
  name: 'AduError'
  readonly statusWord: number
  constructor(statusWord: number)
}

type TransportErrorCode =
  | 'not-supported' // the runtime cannot talk to the device
  | 'not-found'     // no device matched / the chooser was dismissed
  | 'not-connected' // there is no open connection
  | 'timeout'       // no answer in time
  | 'disconnected'  // the link dropped while in use
  | 'io'            // anything else

class TransportError extends OpenloopError {
  name: 'TransportError'
  readonly code: TransportErrorCode // added in v0.6.0
  constructor(message: string, code?: TransportErrorCode) // defaults to 'io'
}
```

OpenloopSDK Class

The class that manages transport registration, discovery, and connection. Every method is static.

```
class OpenloopSDK {
  static registerTransport(type: string, config: {
    factory: TransportFactory
    name: string
    isAvailable: () => boolean | Promise<boolean>
  }): void

  static discover(): Promise<TransportInfo[]>

  static connect(opts?: { transport?: string }): Promise<ITransport>

  static getRegisteredTransports(): string[]

  static clearTransports(): void
}
```

WcTransport Class

```
class WcTransport implements ITransport {
  constructor(options: WcTransportOptions)
  open(): Promise<void>
  close(): Promise<void>
  isConnected(): boolean
  lock(): Promise<void>
  unlock(): Promise<void>
  exchange(apdu: Uint8Array): Promise<Uint8Array>
}
```

Constants

USB Identifiers

```
const OPENLOOP_VENDOR_ID = 0x303a // Espressif VID
const OPENLOOP_PRODUCT_ID = 0x8341 // Openloop PID
const LEDGER_VENDOR_ID = 0x2c97 // Ledger SAS VID
const LEDGER_PRODUCT_ID = 0x1011 // Nano S compatible
```

BIP44 Default Paths

```
const DEFAULT_ETH_PATH = "44'/60'/0'/0/0"
const BTC_MAINNET_PATH = "84'/0'/0'/0/0"
const BTC_TESTNET_PATH = "84'/1'/0'/0/0"
const DEFAULT_SOL_PATH = "44'/501'/0'/0'"
const DEFAULT_TRON_PATH = "44'/195'/0'/0/0"
const DEFAULT_XRP_PATH = "44'/144'/0'/0'/0'"
```

BLE Constants

```
const BLE_SERVICE_UUID = '13d63400-2c97-0004-0000-4c6564676572'
const BLE_NOTIFY_UUID = '13d63400-2c97-0004-0001-4c6564676572'
const BLE_WRITE_UUID = '13d63400-2c97-0004-0002-4c6564676572'
const BLE_DEVICE_NAME_PREFIX = 'Openloop'
const BLE_DEFAULT_MTU = 155
```

WalletConnect Constants

```
const WALLETCONNECT_PROJECT_ID = 'b3d610140c2d0d39f50150a030c7c985'
```

WalletConnect Supported Chains (CAIP-2)

```
const SUPPORTED_EVM_CHAINS = [
  'eip155:1', // Ethereum Mainnet
  'eip155:11155111', // Sepolia Testnet
] as const

const SUPPORTED_BIP122_CHAINS = [
  'bip122:000000000019d6689c085ae165831e93', // Bitcoin Mainnet
  'bip122:0000000000933ea01ad0ee984209779ba', // Bitcoin Testnet3
] as const

const SUPPORTED_SOLANA_CHAINS = [
  'solana:5eykt4UsFv8P8NJdTREpY1vzqKqZKvdp', // Mainnet Beta
  'solana:EtWTRABZaYq6iMfeYKouRu166VU2xqa1', // Devnet
] as const

const SUPPORTED_TRON_CHAINS = [
  'tron:0x2b6653dc', // TRON Mainnet
  'tron:0x94a9059e', // TRON Shasta Testnet
] as const
```

WalletConnect Supported Methods

```
const SUPPORTED_EVM_METHODS = [
  'eth_sendTransaction', 'eth_signTransaction', 'eth_sign',
  'personal_sign', 'eth_signTypedData', 'eth_signTypedData_v3',
  'eth_signTypedData_v4', 'wallet_getCapabilities',
] as const

const SUPPORTED_BIP122_METHODS = [
  'signPsbt', 'getAccountAddresses', 'signMessage', 'sendTransfer',
] as const

const SUPPORTED_SOLANA_METHODS = [
  'solana_signTransaction', 'solana_signAllTransactions',
  'solana_signAndSendTransaction', 'solana_signMessage',
] as const

const SUPPORTED_TRON_METHODS = [
  'tron_signTransaction', 'tron_signMessage',
] as const

const SUPPORTED_EVENTS = ['chainChanged', 'accountsChanged'] as const
```

APDU Instruction Set

```
const APDU = {  
  // Class bytes  
  CLA_ETHEREUM: 0xe0,      // Ledger-compatible  
  CLA_OPENLOOP: 0xf0,     // Openloop-specific  
  
  // Shared INS (routed automatically by coin_type)  
  INS_GET_PUBLIC_KEY: 0x02, // Get the address / public key  
  INS_SIGN_TX: 0x04,       // Sign a transaction  
  INS_SIGN_MESSAGE: 0x08,  // Sign a message  
  INS_SIGN_EIP712: 0x0c,   // EIP-712 typed data  
  INS_SIGN_AUTHORIZATION: 0x34, // EIP-7702  
  
  // Bitcoin PSBT (Openloop-specific)  
  INS_BTC_SIGN_PSBT: 0x70, // Send the PSBT  
  INS_BTC_GET_SIGNED_PSBT: 0x71, // Get the signed PSBT  
  INS_BTC_GET_WALLET_PUBLIC_KEY: 0x40, // Get the BTC public key / address  
  INS_BTC_GET_ACCOUNT_XPUB: 0x72, // Get the account xpub  
  INS_BTC_SIGN_MESSAGE: 0x73, // Sign a BTC message  
  
  // Solana (Ledger Solana compatible)  
  INS_SOL_GET_PUBKEY: 0x05, // Get the Ed25519 public key  
  INS_SOL_SIGN_MESSAGE: 0x06, // Sign a transaction / message  
  INS_SOL_SIGN_OFFCHAIN_MSG: 0x07, // Sign an off-chain message  
} as const
```

Utility Functions

```
// Build an APDU command
function buildApdu(cla: number, ins: number, p1: number, p2: number, data?: Uint8Array): Uint8Array

// Convert a BIP32 path to a buffer
function pathToBuffer(path: string): Uint8Array

// Convert a BIP32 path to an array of elements
function pathToElements(path: string): number[]

// Concatenate Uint8Arrays
function concat(...arrays: Uint8Array[]): Uint8Array

// Convert a hex string to a Uint8Array (with or without the 0x prefix)
function hexToBytes(hex: string): Uint8Array

// Convert a Uint8Array to a hex string (no 0x prefix)
function bytesToHex(bytes: Uint8Array): string

// Base58 encode / decode
function base58Encode(bytes: Uint8Array): string
function base58Decode(str: string): Uint8Array

// Format an OpenloopDeviceState into a single-line log string
// e.g. "wallet=N lang=0 USB(native) ble=off FIDO PIV"
function formatOpenloopState(s: OpenloopDeviceState): string
```