# GridPlus SDK Documentation

***Release 0.0.0***

**Alex Miller**

**Jan 28, 2022**

# Contents

The GridPlus SDK allows any application to establish a connection and interact with a GridPlus Lattice device.

CHAPTER 1

# Installation

This SDK is currently only available as a `node.js` module. You can add it to your project with:

```
npm install gridplus-sdk
```

You can then import a new client with:

```
import { Client } from 'gridplus-sdk';
```

or, for older style syntax:

```
const Sdk = require('gridplus-sdk').Client;
```

CHAPTER 2

# Instantiating a Client

Once imported, you can instantiate your SDK client with a `clientConfig` object, which at minimum requires the name of your app (`name`) and a private key with which to sign requests (`privKey`). The latter is not meant to e.g. hold onto any cryptocurrencies; it is simply a way of maintaining a secure communication channel between the device and your application.

```
const crypto = require('crypto');
const clientConfig = {
    name: 'MyApp',
    crypto: crypto,
    privKey: crypto.randomBytes(32).toString('hex')
}
```

## 2.1 Client options

CHAPTER 3

# Connecting to a Lattice

With the `clientConfig` filled out, you can instantiate a new SDK object:

```
const client = new Client(clientConfig);
```

With the client object, you can make a connection to any Lattice device which is connected to the internet:

```
const deviceId = 'MY_LATTICE';
client.connect(deviceId, (err, isPaired) => {
    ...
});
```

If you get a non-error response, it means you can talk to the device. Note that the response also tells you whether you are paired with the device.

> The `deviceId` is listed on your Lattice under `Settings->Device Info`

## 3.1 Canceling a Pairing Request

If you get `isPaired = false` in the callback, this request will have started the pairing request with the specified device, which will now be showing a random 8 character pairing code for 60 seconds.

If you wish to cancel this request, you may call `pair()` with an empty string `''` as the first argument. This will gracefully cancel the request. You may also call `pair()` with any random string which will also cancel the request, but the Lattice will show an error screen.

CHAPTER 4

# Pairing with a Lattice

This function requires the user to interact with the Lattice. It therefore uses your client's timeout to sever the request if needed.

When `connect` is called, your Lattice will draw a random, six digit secret on the screen. The SDK uses this to "pair" with the device:

```
client.pair('SECRET', (err, hasActiveWallet) => {
    ...
});
```

A non-error response indicates you may now make encrypted requests.

If `hasActiveWallet = false`, it means there was an error fetching the current wallet on the device. This could mean the device has not been set up or that a SafeCard is inserted which has not been set up. It could also mean there was an error with the connection. If you try to get addresses or sign without an active wallet saved (it is saved automatically if `hasActiveWallet = true`), the SDK will automatically retry fetching the active wallet before making the original request.

# CHAPTER 5

## Getting Addresses

If the SDK is connected to the wrong wallet or if the device has no current active wallet, this request will take additional time to complete.

You may retrieve some number of addresses for supported cryptocurrencies. The Lattice uses BIP44-compliant highly-deterministic (HD) wallets for generating addresses. You may request a set of contiguous addresses (e.g. indices 5 to 10 or 33 to 36) based on a currency (`ETH` or `BTC`). *For now, you may only request a maximum of 10 addresses at a time from the Lattice per request.*

NOTE: For BTC, the type of address returned will be based on the user's setting. For example, if the user's latter is configured to return segwit addresses, you will get addresses that start with `3`.

An example request looks like:

```
const HARDENED_OFFSET = 0x80000000;
const req = {
    // -- m/49'/0'/0'/0/0, i.e. first BTC address
    startPath: [HARDENED_OFFSET+49, HARDENED_OFFSET, HARDENED_OFFSET, 0, 0],
    n: 4
};
client.addresses(req, (err, res) => {
    ...
})
```

NOTE: For v1, the Lattice1 only supports `p2sh-p2wpkh` BTC addresses, which require a `49'` purpose, per BIP49. Ethereum addresses use the legacy `44'` purpose.

**Options:**

**Response:**

Returns an array of address strings (if the user's Lattice is configured to return segwit addresses):

```
res = [
    '3PKEDaainApM4u5Tqm1nn3txzZWbtFXUQ2',
    '3He2JrsT33DEnjCgdpPgc6RXD3UogALCNF',
    '3QybQyM8i9YR9e9Tgb1zLsYHHRXWF1eDAR',
    '3PNwCSHKNfCjzvcU8XE9N8wp8DRxrUzsyL'
]
```

# CHAPTER 6

# Requesting Signatures

> This function requires the user to interact with the Lattice. It therefore uses your client's timeout to sever the request if needed. If the SDK is connected to the wrong wallet or if the device has no current active wallet, this request will take additional time to complete.

The Lattice device, at its core, is a tightly controlled, highly configurable, cryptographic signing machine. By default, each pairing (the persistent association between your app and a user's lattice) allows the app an ability to request signatures that the user must manually authorize.

## 6.1 Request Types

The following types of requests are currently supported by the Lattice. These correspond to the `currency` param in the `sign` options (`signOpts` below)

## 6.2 `ETH` (Ethereum transaction)

Ethereum transactions consist of six fields. An example payload looks as follows:

```
const data = {
    nonce: '0x01',
    gasLimit: '0x61a8,
    gasPrice: '0x2540be400,
    to: '0xe242e54155b1abc71fc118065270cecaaf8b7768',
    value: 0,
    data: '0x12345678'
    // -- m/44'/60'/0'/0/0
    signerPath: [HARDENED_OFFSET+44, HARDENED_OFFSET+60, HARDENED_OFFSET, 0, 0],
    chainId: 'rinkeby',
    useEIP155: false,
}
const signOpts = {
    currency: 'ETH',
    data: data,
}
```

### 6.2.1 Chain ID

The `chainId` param is used to provide replay protectin for most Ethereum-based chains. We allow several ways to specify this:

1. A "named" chain, with options being: `mainnet`, `ropsten`, `rinkeby`, `kovan`, `goerli`
2. An integer (only recommended for small numbers – see below section)
3. A hex string (e.g. `0x1234`)

**Hex strings are strongly recommended**

Generally, we recommend **not** using Javascript integers and **never** using them for fields that may contain large values, such as `value` (which is measured in units of wei, where 10**18 wei = 1 ether). We recommend using hex strings instead, as shown in the example above. Consider the following dummy code in `node.js`:

```
> new bn(2).pow(64).toString(16)
'10000000000000000'
> (2**64).toString(16)
'10000000000000000'
> (2**64-2).toString(16)
'10000000000000000'
> new bn(2**64).toString(16)
'10000000000000180'
> 2**64
18446744073709552000
> new bn(18446744073709552000-2).toString(16)
'10000000000000180'
```

As you can see, all sorts of problems arise from large Javascript integers. Don't use them!

Note that in the `gridplus-sdk`, all numerical inputs are converted to big numbers, but we still recommend avoiding them.

**"Named" `chainIds`**

We support a hand full of human-readable strings for specifying a network. These include the Ethereum mainnet and current widely used testnets. It is important to note that **some networks use EIP155 by default and others don't**. You can, of course, specify whether you want to use EIP155 or not explicitly using the `eip155` param. Please see the following table for EIP155 defaults:

## 6.3 `ETH_MSG` (Ethereum message)

In addition to transactions, we support signing ETH messages, e.g.:

```
const data = {
    protocol: 'signPersonal',
    payload: '0xdeadbeef',
    signerPath: [HARDENED_OFFSET+44, HARDENED_OFFSET+60, HARDENED_OFFSET, 0, 0],
}
const signOpts = {
    currency: 'ETH_MSG',
    data: data,
}
```

### 6.3.1 Supported ETH_MSG protocols

- `signPersonal`: ETH personalSign (EIP191)

## 6.4 BTC (Bitcoin transaction)

Bitcoin transactions are constructed by referencing a set of inputs to spend and a recipient + output value. You should also specify a change address path (defaults to `m/44'/0'/0'/1/0`):

```
const data = {
    prevOuts: [
        {
            txHash:
↪'08911991c5659349fa507419a20fd398d66d59e823bca1b1b94f8f19e21be44c',
            value: 3469416,
            index: 1,
            signerPath: [HARDENED_OFFSET+49, HARDENED_OFFSET, HARDENED_OFFSET, 1,␣
↪0],
        },
        {
            txHash:
↪'19e7aa056a82b790c478e619153c35195211b58923a8e74d3540f8ff1f25ecef',
            value: 3461572,
            index: 0,
            signerPath: [HARDENED_OFFSET+49, HARDENED_OFFSET, HARDENED_OFFSET, 0,␣
↪5],
        }
    ],
    recipient: 'mhifA1DwiMPHTjSJM8FFSL8ibrzWaBCkVT',
    value: 1000,
    fee: 1000,
    isSegwit: true,
    changePath: [HARDENED_OFFSET+49, HARDENED_OFFSET, HARDENED_OFFSET, 1, 1],
=}
const signOpts = {
    currency: 'BTC',
    data: data,
}
```

## 6.5 Requesting the Signature

Once you build the data needed, you can request a signature using the following pattern:

```
client.sign(signOpts, (err, signedTx) => {

})
```

**Response**

The returned `signedTx` object has the following properties:

CHAPTER 7

# Getting Active Wallets

The Lattice1 has two wallet "slots": an internal wallet that is always the same for a given device and an external slot for SafeCard wallets. When a SafeCard is inserted or removed, the external slot is updated. If a wallet is present in a given slot, the device will allow paired requesters to get the "wallet UID", against which addresses or signatures may be requested. This UID is a permanent identifier for a given wallet (i.e. every SafeCard, once setup, will have a permanent UID that maps directly to a wallet seed and, therefore, to a set of addresses).

Although these requests are abstracted from the user of this SDK, you may look at the active wallets currently known by the SDK. This may be useful for determining if there is a SafeCard inserted.

```
const wallet = client.getActiveWallet();
```

This will return an object containing:

```
uid           // 32 byte buffer id
name          // 20 char (max) string
capabilities  // 4 byte flag
external      // boolean
```

Where `uid` is a 32-byte buffer containing the wallet UID discussed above and `external` is `true` if the active wallet is a SafeCard. **NOTE: If a SafeCard is inserted, this will be the data returned from `getActiveWallet()`. When it is removed, you will get the internal wallet data. Currently, `name` and `capabilities` are not used.

## 7.1 Detecting Card Insertion/Removal

When a card is inserted or removed, this will affect the active wallet of the device. If you want to stay up to date on the latest wallet state, you will need to *refresh* the active wallet. You can do this by "re-connecting":

```
client.connect((err) => {
    activeWallet = client.getActiveWallet();
})
```

Note that you may only call `connect` with one argument once a `deviceID` has been saved, i.e. after you've called `connect` once with the device ID as the first argument.