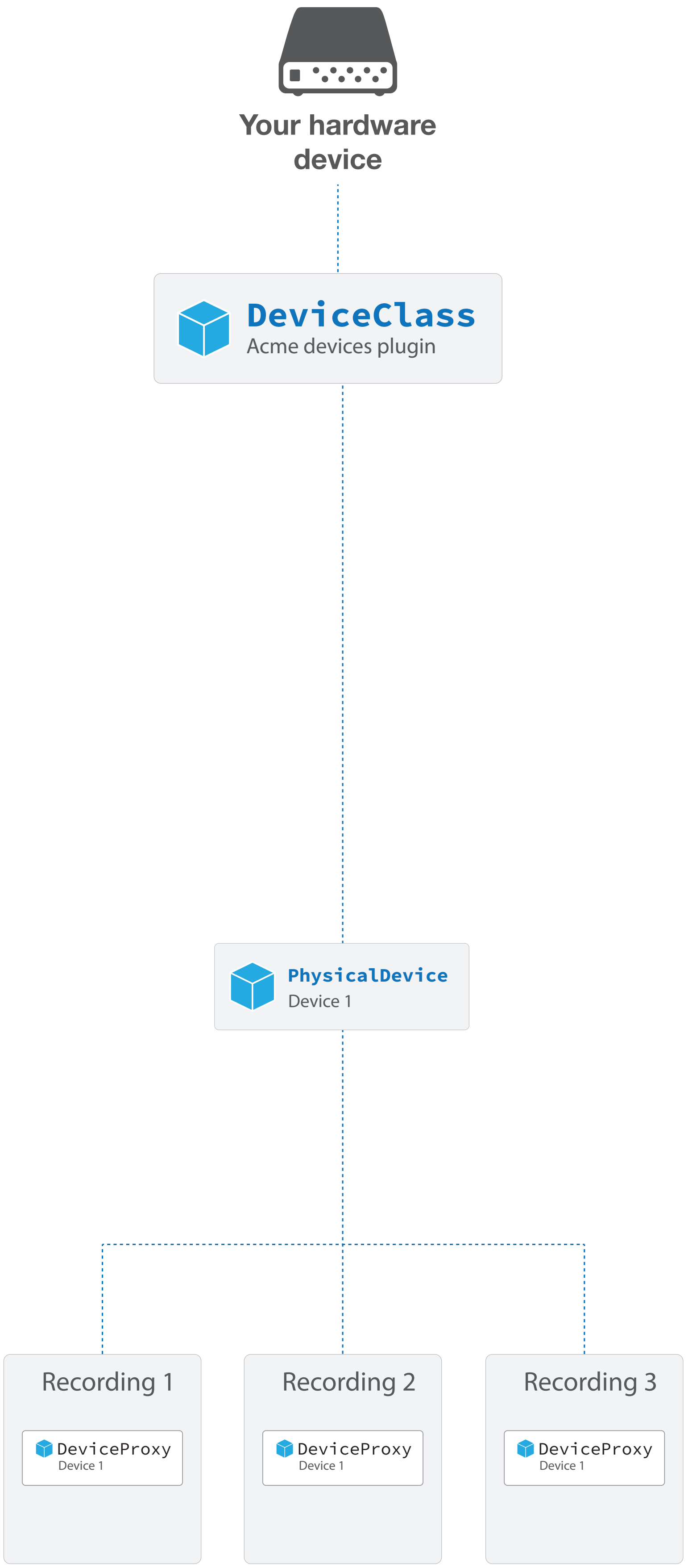
LabChart LIGHTNING

SDK Documentation

v 2.01

Overview (single device)



Your hardware device

DeviceClass
Acme devices plugin

PhysicalDevice
Device 1

Recording 1
DeviceProxy
Device 1

Recording 2
DeviceProxy
Device 1

Recording 3
DeviceProxy
Device 1

The `DeviceClass` is implemented by your plugin code. Each device class object can support a single device, or a range of related devices that can handle the same types of settings.

On launch, LabChart Lightning loads all plugins.

When LabChart Lightning starts, it calls `getDeviceClasses()` on your plugin. Each plugin can, if desired, implement more than one `DeviceClass`. Note that Lightning will not let settings from one `DeviceClass` be applied to a device from a different class (this is essentially the defining feature of a `DeviceClass`).

Typically there would only be one `DeviceClass` per plugin (unless the plugin needs to support devices of very different types).

This must return an object array which Lightning can query by calling `DeviceClass.checkDeviceIsPresent()` for each possible device connection.

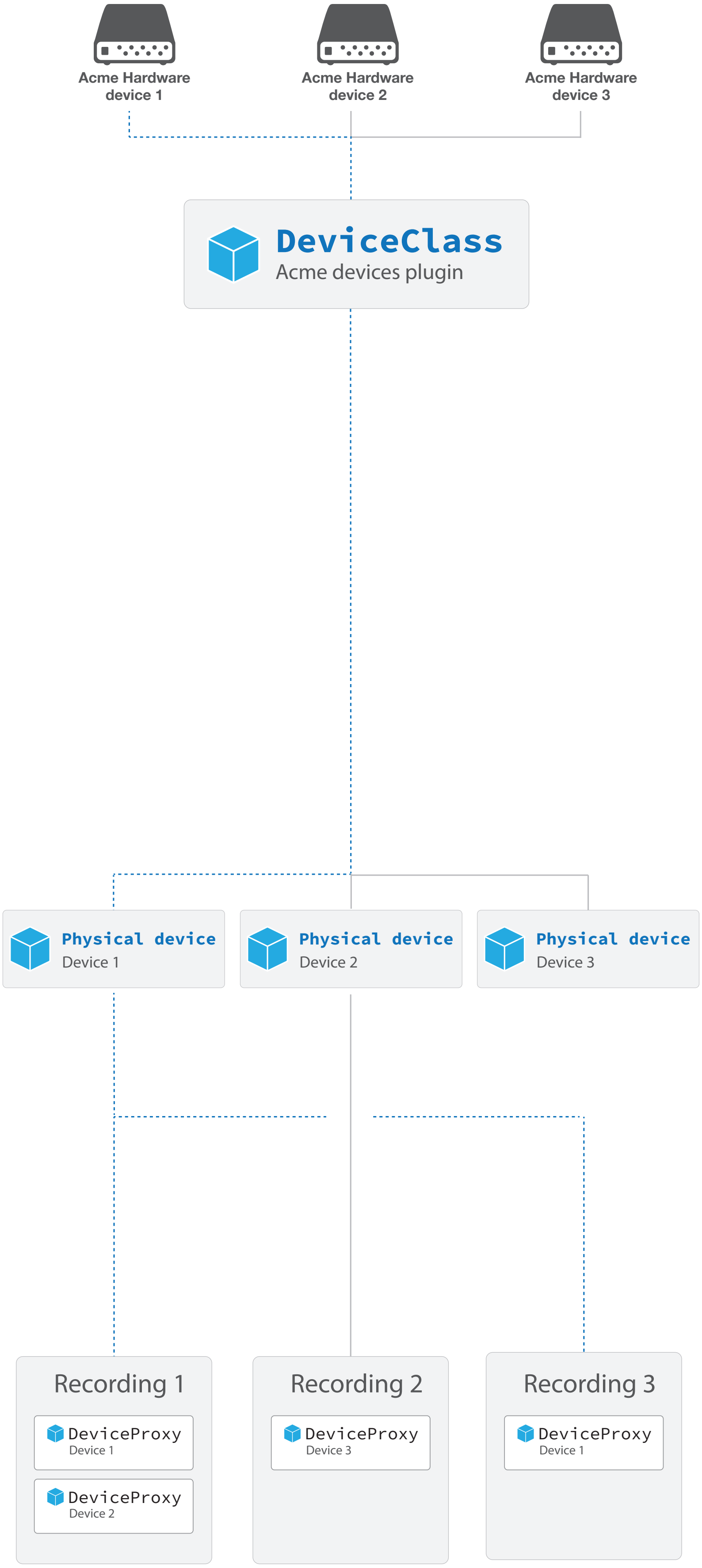
A `PhysicalDevice` object is created by your plugin code and represents an instance of a single, connected hardware device. Your plugin creates `PhysicalDevice` objects for as many devices that are discovered.

Multiple LabChart Lightning recordings can use the same `PhysicalDevice`, but only one can sample with that device at any time.

Each recording in Lightning holds the settings for that device for that recording and allows the getting and setting of device settings specific to the recording.

Lightning uses the `DeviceClass` supplied by the plugin to create these `DeviceProxy` objects as required.

Overview (multiple devices)



The **DeviceClass** is implemented by your plugin code. Each device class object can support a single device, or a range of related devices that can handle the same types of settings.

On launch, LabChart Lightning loads all plugins.

When LabChart Lightning starts, it calls **getDeviceClasses()** on your plugin. Each plugin can, if desired, implement more than one **DeviceClass**. Note that Lightning will not let settings from one **DeviceClass** be applied to a device from a different class (this is essentially the defining feature of a **DeviceClass**).

Typically there would only be one **DeviceClass** per plugin (unless the plugin needs to support devices of very different types).

This must return an object array which Lightning can query by calling **DeviceClass.checkDeviceIsPresent()** for each possible device connection.

A **PhysicalDevice** object is created by your plugin code and represents an instance of a single, connected hardware device. Your plugin creates **PhysicalDevice** objects for as many devices that are discovered.

Multiple LabChart Lightning recordings can use the same **PhysicalDevice**, but only one can sample with that device at any time.

Each recording in Lightning holds the settings for that device for that recording and allows the getting and setting of device settings specific to the recording.

Lightning uses the **DeviceClass** supplied by the plugin to create these **DeviceProxy** objects as required.

Plugin location and structure

Location

C:\Users\[USERNAME]\Documents\LabChart Lightning\Plugins\Devices	**Windows**
~/Documents/LabChart Lightning/Plugins/Devices	**macOS**

Structure/contents

- Plugins
  - Devices
    - plugin-name
      - plugin-name.js **(required)**

        The entry point for you plugin’s code. Split up your code into multiple files and import/reference them in this file.
      - Additional files (optional)

        Split up your code into as many (or as few) files as necessary. Import/reference these files in the *plugin-name.js* file.

What your plugin needs to implement

You will need to create the following TypeScript classes as well as their required methods.

`DeviceClass`

(extends `IDeviceClass`)

The **`DeviceClass`** object represents this set of devices and can find and create **`PhysicalDevice`** objects of its class, as well as the **`ProxyDevice`** objects.

All devices in this class can understand the same settings. An important function of the **`DeviceClass`** is enable Lightning to match hardware devices to previous device settings (i.e. **`ProxyDevices`**) when reopening recordings by matching properties such as serial numbers and device capabilities.

The **`DeviceClass`** also contains properties shared by all devices in the class.

`ProxyDevice`

(extends `IProxyDevice`)

The **`ProxyDevice`** object is created for each recording and manages the device settings and sampling for its recording.

`PhysicalDevice`

(extends `OpenPhysicalDevice`)

`PhysicalDevice` is a representation of the connected hardware device. Once created, **`PhysicalDevices`** last for the entire Lightning session.

The **`PhysicalDevice`** holds onto the **`Parser`** and holds the capabilities of its hardware device, e.g. maximum number of inputs, the serial number and other properties that can be used to identify that particular piece of hardware for matching settings to devices when recordings are reopened.

What your plugin needs to implement

Inputs vs streams

Input

Each Input corresponds to a hardware input on the device (e.g. a BNC connector on a PowerLab), so it has a Range, typically in V, mV or uV which depends on the gain of the input (which may be fixed or adjustable).

An input may have other device dependent properties such as AC coupling, Differential (vs single ended), etc, which are stored in the `InputSettings` object.

Stream

Each `ProxyDevice` provides sample data to its recording via one or more data streams. The number of streams can depend both on the capabilities of the hardware and the settings.

The number of streams need not equal the number of hardware inputs, e.g. the device may calculate more than computed data stream from each input.

Required classes:

`DeviceClass`

`(IDeviceClass)`

The `DeviceClass` object represents a set of devices that can share the same types of settings and can find and create `PhysicalDevice` objects of its class, as well as the `ProxyDevice` objects.

`ProxyDevice`

`(IProxyDevice)`

The `ProxyDevice` object is created for each recording that uses the device and manages hardware settings and sampling and access to the `PhysicalDevice` (only one recording can access the `PhysicalDevice` at any one time).

`PhysicalDevice`

`(OpenPhysicalDevice)`

`PhysicalDevice` is a representation of the connected hardware device and owns the parser and the device connection.

`Parser`

A custom object, designed by you, that handles parsing of data returned from the device and adds the resulting samples to the device's output streams (signals).

Other classes:

`InputSettings`

The input corresponds to a hardware input on the device (e.g. a BNC connector on a PowerLab), so it has a Range, typically in V, mV or uV which depends on the gain of the input (which may be fixed or adjustable).

An input may have other device dependent properties such as AC coupling, Differential (vs single ended), etc.

Sometimes a device provide more streams (signals) than it has inputs, generally because it calculates multiple signals from one or more of its inputs.

Often these computed streams have different units or sample rates from the input(s).

`StreamSettings`

`(IDeviceStreamApi)`

The settings for each Stream for each recording are stored in a StreamSettings object. The ProxyDevice holds an array of these.

A `StreamSettings` object can hold arbitrary, device or stream specific settings in addition to the standard ones: "enabled", "samplesPerSec" and "inputSettings" (which is of type `InputSettings`).

`DeviceStreamConfigurationImpl`

`(IDeviceStreamConfiguration)`

This object holds the information Quark needs to know about the units and data format of the Stream. The `unitsInfo` will change if the gain of the stream's input changes.

DeviceClass

(extends `IDeviceClass`)

The **`DeviceClass`** object represents this set of devices and can find and create **`PhysicalDevice`** objects of its class, as well as the **`ProxyDevice`** objects.

Identifying the device

```
getClassId(): string
```

Should return a universally unique identifier (uuid) specific to this device class.

```
getDeviceClassName(): string
```

Should return a human readable name for the **`DeviceClass`** to be shown in Lightning UI.

```
getDeviceConnectionType(): TDeviceConnectionType;
```

Specifies the type of the connections Quark should create and pass to **`checkDeviceIsPresent()`** to support device discovery. E.g. **`kDevConTypeSerialPort`** or **`kDevConTypeSerialOverBluetooth`**.

Handling connections

```
createProxyDevice(
        quarkProxy: ProxyDeviceSys | null,
        physicalDevice: OpenPhysicalDevice | null
    ): IProxyDevice
```

```
checkDeviceIsPresent(
        deviceConnection: DuplexDeviceConnection,
        callback: (error: Error | null, device: OpenPhysicalDevice | null) => void
    ): void
```

```
indexOfBestMatchingDevice(
        descriptor: OpenPhysicalDeviceDescriptor,
        availablePhysDevices: OpenPhysicalDeviceDescriptor[]
    ): number
```

Called when deciding (typically when loading a recording) which physical device should be used for the specified device proxy in a recording.

This function should run through all the available **`PhysicalDevices`** comparing them against the capabilities and device id properties passed in the descriptor parameter in order to choose the best match.

If a valid index is returned, that **`PhyiscalDevice`** will be removed from the available **`PhyiscalDevices`** passed in future calls.

Handling errors

`onError(err: Error): void` optional

Called when a sampling or connection error is detected. You can choose to ignore these.

Other

`release?(): void` optional

Called when the app shuts down. Chance to release any resources acquired during this object's lifetime.

PhysicalDevice

(extends `OpenPhysicalDevice`)

`PhysicalDevice` is a representation of the connected hardware device.

`PhysicalDevices` need to own a custom parser object which typically holds onto the `PhysicalDevice's DeviceConnection` so that the parser can always receive any bytes arriving from the device.

What you must implement

`getDeviceName(): string;`

`getNumberOfAnalogInputs(): number;`

`getNumberOfAnalogStreams(): number;`

Commonly, this will be the same as the number of analog inputs, unless the device can generate multiple calculated data streams from its hardware inputs.

`getDescriptor(): OpenPhysicalDeviceDescriptor;`

`release(): void;`

Since the **`PhysicalDevice`** owns its `DeviceConnection`, **`release()`** should call **`onStreamDestroy()`** and then **`release()`** on the **`deviceConnection`**, (which is normally held onto by the **`PhysicalDevice's`** parser).

ProxyDevice

(extends `IProxyDevice`)

The **`ProxyDevice`** object is created for each recording and manages access to the **`PhysicalDevice`** which may be shared by multiple recordings, each having different settings.

What you must implement

{} `getOutBufferInputIndices(): Int32Array;`

{} `getDeviceName(): string;`

{} `getNumberOfAnalogStreams(): number;`

{} `getLastError(): string;`

{} `setPhysicalDevice(physicalDevice: OpenPhysicalDevice): boolean;`

{} `connectToPhysicalDevice(): boolean;`

Called from Quark to allow this proxy to communicate with the device.

{} `disconnectFromPhysicalDevice(): void;`

Called from Quark to stop this proxy communicating with the device to allow another proxy to use the device.

{} `prepareForSampling(bufferSizeInSecs: number): boolean;`

Allocate StreamRingBuffers buffers.

{} `startSampling(): boolean;`

{} `onSamplingStarted(): void;`

{} `onSamplingUpdate(): void;`

{} `stopSampling(): boolean;`

{} `onSamplingStopped(errorMsg: string): void;`

{} `cleanupAfterSampling(): boolean;`

Release buffers and cleanup.

P `isSampling: boolean;`

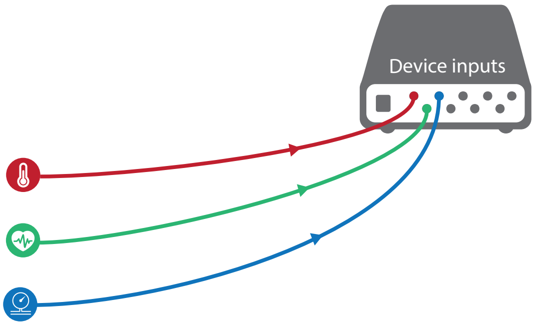
P `outStreamBuffers: StreamRingBuffer[];`

Inputs, streams, and signals.

Inputs

Each Input corresponds to a hardware input on the device (e.g. a BNC connector on a PowerLab), so it has a Range, typically in V, mV or uV which depends on the gain of the input (which may be fixed or adjustable).

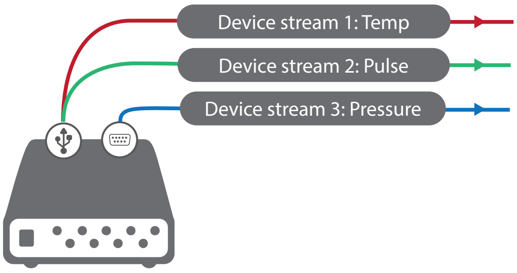
An input may have other device dependent properties such as AC coupling, Differential (vs single ended), etc, which are stored in the InputSettings object.



Streams

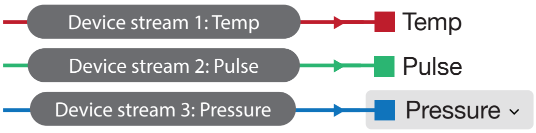
Each ProxyDevice provides sample data to its recording via one or more data streams. The number of streams can depend both on the capabilities of the hardware and the settings.

The number of streams need not equal the number of hardware inputs, e.g. the device may calculate more than computed data stream from each input.



Signals

Lightning maps the data streams to signals and exposes these signals to users.



The connection process

Lightning is launched.

On launch, LabChart Lightning loads all plugins by calling `getDeviceClasses()` on each plugin. This returns an instance of the plugin `DeviceClass` implementation. e.g. `[return new DeviceClass()];`

Lightning scans for hardware.

For each device connection that is not already being used by a physical device:

```
checkDeviceIsPresent(deviceConnection, callback)
```

is called on the `DeviceClass` implementation.

It is up to the `checkDeviceIsPresent` function within the script to decide whether its hardware is present, e.g. the device might use a generic serial to USB chip such as FTDI such that the vendor, product and manufacturer information (vid, pid etc) are not sufficient to identify the actual device type, in which case it will need to try communicating with it.

If a device fo the right type is detected, a new `PhysicalDevice` object is instantiated and returned to LabChart Lightning via callback.

DeviceClass
Acme devices plugin

PhysicalDevice objects created.

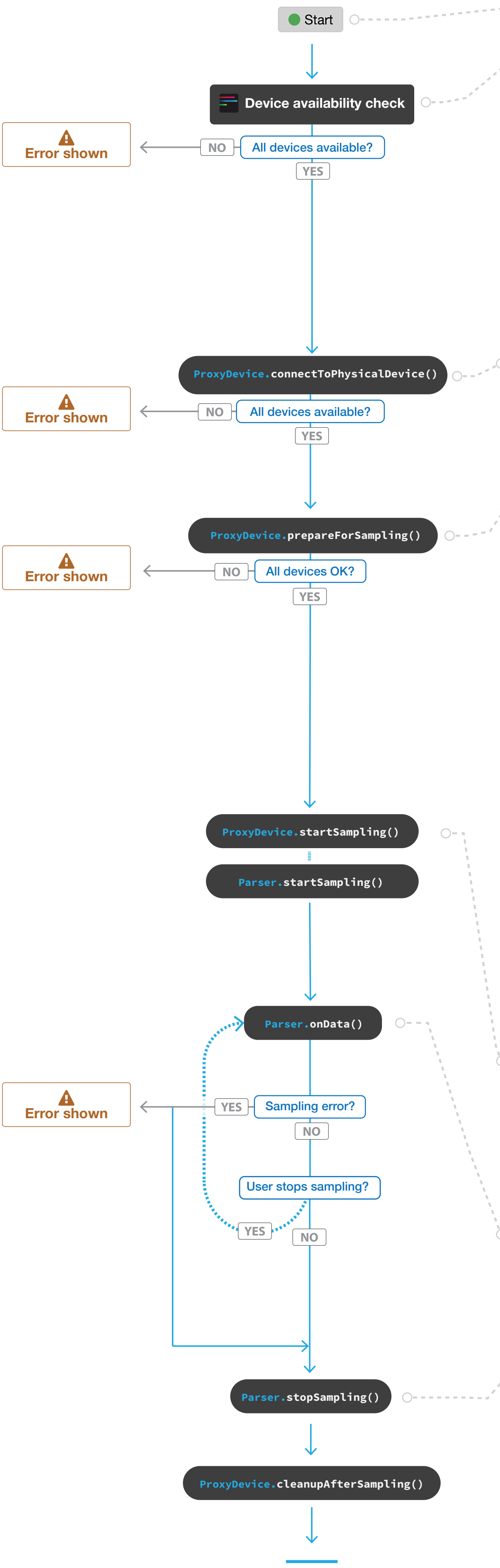
When physical devices are created during search for devices, any recordings using devices of that class are notified so that they can connect any proxies of that class to the physical devices.

When recordings are created, one of two things happen:

(1) If a recording is cerated with default settings, a `DeviceProxy` is created for the default device for use by the recording.

(2) If a recording is cloned from an existing recording, the `DeviceProxies` within the existing recording are cloned for useby the new recording.

The Sampling Process



Start

Device availability check

All devices available? NO → Error shown; YES

ProxyDevice.connectToPhysicalDevice()

All devices available? NO → Error shown; YES

ProxyDevice.prepareForSampling()

All devices OK? NO → Error shown; YES

ProxyDevice.startSampling()

Parser.startSampling()

Parser.onData()

Sampling error? YES → Error shown; NO

User stops sampling? YES; NO

Parser.stopSampling()

ProxyDevice.cleanupAfterSampling()

1 The User clicks on *Start Sampling*.

2 Enabled proxy devices used in the recording are checked.

Lightning first checks that each enabled ProxyDevice used by the recording has at least one enabled stream and that it's PhysicalDevice (or hardware) is not currently being used by another recording (or another application such as LabChart8 or Lt).

Lightning then checks that each device's connection to the hardware is ok. Because all current device classes use connections created in C++ by Lightning, these checks are done in Quark C++ code (e.g. `CanStartSampling()`).

In the future it is likely that Lightning will add support for plugins creating connections (e.g. for networked devices) in Typescript code. In that case, the code for ensuring that only one ProxyDevice can use a hardware device at a time will be written in Typescript code provided by the SDK library.

3 Lightning connects to physical devices.

Lightning calls the JavaScript `ProxyDevice.`**`connectToPhysicalDevice()`** method. Because a ProxyDevice from another recording (or application such as Lt) may have been using the hardware, this method should send all the proxy's settings down to the hardware.

4 Prepare for sampling.

Lightning calls:

```
ProxyDevice.prepareForSampling(bufferSizeInSecs: number):
boolean
```

for each enabled device used by the recording.

If any of these calls result in error or warning events, these are displayed in the UI.

ProxyDevices should allocate **`SharedArrayBuffer`** for each of their Streams with a size sufficient to hold at least bufferSizeInSecs worth of samples for the sample rate for each Stream. If any `prepareForSampling()` fails, Lightning will call:

`ProxyDevice.`**`cleanupAfterSampling()`**

on each ProxyDevice that has had **`prepareForSampling()`** called on it.

If the JavaScript code detects any issues that the user needs to be aware of, it should call `onDeviceEvent` on the Quark proxy device, e.g.:

```
this.proxyDeviceSys?.onDeviceEvent(
    DeviceEvent.kDeviceStartSamplingUserQuery,
    this.getDeviceName(),
    'Sampling will inflate finger cuffs. They can be damaged if
     inflated while empty.',
    {
       onceOnly: 'check-finger-cuffs',
       severity: TMessageSeverity.kMessageWarn
    }
);
```

5 Start sampling.

If all the `ProxyDevice.`**`prepareForSampling()`** calls succeed, Lightning calls `ProxyDevice.`**`startSampling()`** on each enabled ProxyDevice. This should call `parser.`**`startSampling()`** so that the parser knows to start writing samples that arrive into each enabled Stream's SharedArrayBuffer, thereby allowing the sample to be stored by Lightning.

6 Data received.

Whether or not Lightning is sampling, each DeviceConnection will call the **`onData()`** callback that has been set on it whenever any new bytes arrive from the hardware device.

7 Sampling stops

When the user clicks Stop or in future, if a fixed duration recording ends, Quark calls `ProxyDevice.`**`stopSampling()`** on each enabled ProxyDevice. This should call `parser.`**`stopSampling()`**.

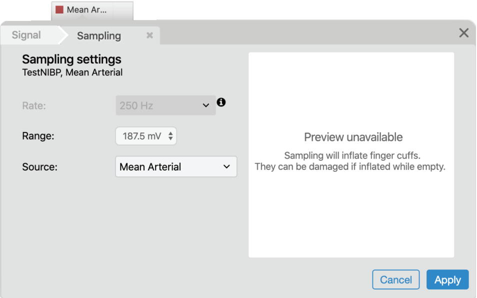
If the sampling session is then finished, because it was a user stop or a repetitive fixed-duration sampling session finished repeating, Quark calls `ProxyDevice.`**`cleanupAfterSampling()`** on each enabled ProxyDevice. This should release references to the SharedArrayBuffers held by the ProxyDevice.

Places your plugin can present UI

Configuring signals

Each signal from your device can be configured using the Signal properties popover in the ChartView.

Your plugin script can define custom UI for this popover.



Sampling settings
TestNIBP, Mean Arterial

Rate: 250 Hz

Range: 187.5 mV

Source: Mean Arterial

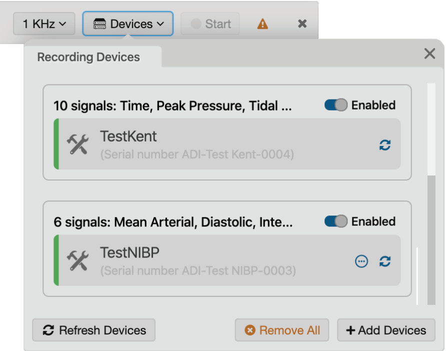
Preview unavailable

Sampling will inflate finger cuffs.
They can be damaged if inflated while empty.

Device connection

Devices handled by your plugin script will be shown in either the Add Device list (if it is not used by the active recording) or the Recording Devices list (if it is being used).

Name, status, and button to link to custom user interface (if any exists) are shown.



10 signals: Time, Peak Pressure, Tidal ... Enabled

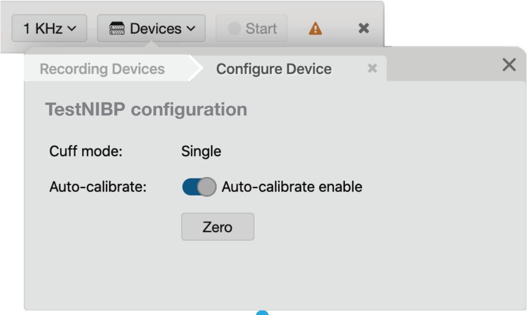
TestKent
(Serial number ADI-Test Kent-0004)

6 signals: Mean Arterial, Diastolic, Inte... Enabled

TestNIBP
(Serial number ADI-Test NIBP-0003)

Device configuration

If your device has custom user interface (defined in the Device plugin script), it will be shown in the configure devices panel.



TestNIBP configuration

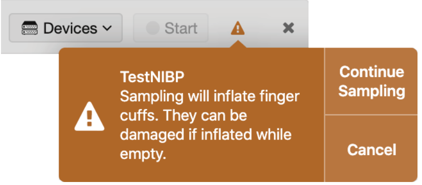
Cuff mode: Single

Auto-calibrate: Auto-calibrate enable

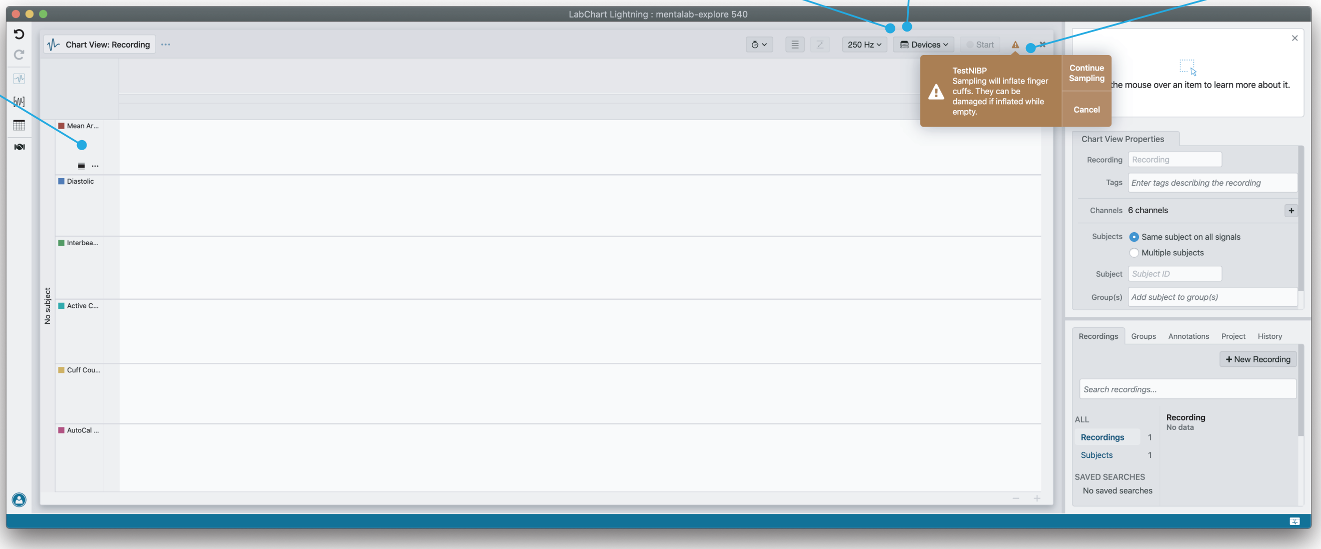
Pre-sampling warnings

When a user selects *Start sampling*, your plugin can temporarily halt the start sampling process to display a message.

In the example below, a plugin is telling the user that their device may be damaged if not used correctly. The user has the option of continuing (*Continue Sampling*), or canceling sampling.



TestNIBP
Sampling will inflate finger cuffs. They can be damaged if inflated while empty.



TestNIBP
Sampling will inflate finger cuffs. They can be damaged if inflated while empty.

the mouse over an item to learn more about it.

Recording: Recording

Tags: *Enter tags describing the recording*

Channels: 6 channels

Subjects: ● Same subject on all signals ○ Multiple subjects

Subject: *Subject ID*

Group(s): *Add subject to group(s)*

Search recordings...

ALL
Recordings 1
Subjects 1

Recording
No data

SAVED SEARCHES
No saved searches

Your hardware device.

Your hardware device provides one or more data streams to the host computer.

The data streams may arrive in Lightning via USB, Bluetooth, serial, or other connection type.

Your plugin.

Your plugin is responsible for presenting the device streams to Lightning.

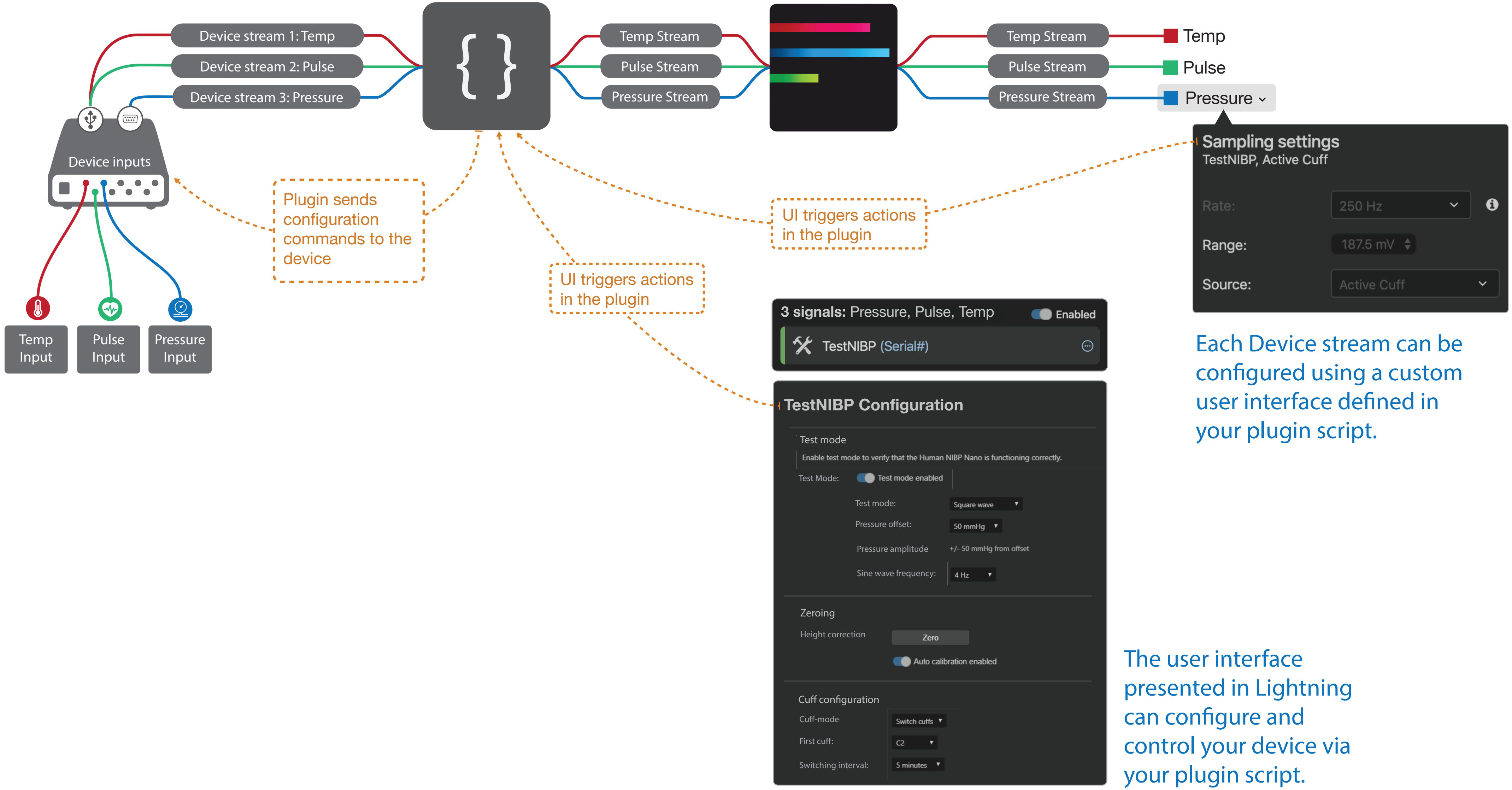
It can allow users to configure your device and device streams through a graphical user interface that is defined in the plugin itself.

Lightning shows your device to the user in the devices list.

Lightning maps the data streams to signals and exposes these signals to users.

Users are shown a graphical representation of your hardware device in the devices list.

The signals are displayed in Lightning channels.



Plugin sends configuration commands to the device

UI triggers actions in the plugin

UI triggers actions in the plugin

Sampling settings
TestNIBP, Active Cuff

Rate: 250 Hz

Range: 187.5 mV

Source: Active Cuff

3 signals: Pressure, Pulse, Temp — Enabled

TestNIBP (Serial#)

TestNIBP Configuration

Test mode

Enable test mode to verify that the Human NIBP Nano is functioning correctly.

Test Mode: Test mode enabled

Test mode: Square wave ▾

Pressure offset: 50 mmHg ▾

Pressure amplitude +/- 50 mmHg from offset

Sine wave frequency: 4 Hz ▾

Zeroing

Height correction [Zero]

Auto calibration enabled

Cuff configuration

Cuff-mode: Switch cuffs ▾

First cuff: C2 ▾

Switching interval: 5 minutes ▾

Each Device stream can be configured using a custom user interface defined in your plugin script.

The user interface presented in Lightning can configure and control your device via your plugin script.