PLUX Java Application Programming Interface

Documentation – Android API

Contents

1. General Information

Interface Specification Document			
Plux Java Application Programming Interface			
		Date	2016-06-02
Current Revision	2	**Author**	Gonçalo Telo

Change History			
Revision Number	**Author**	**Date**	**Change**
1	Gonçalo Telo	2016-01-20	Initial Document for API 0.0.5
2	Gonçalo Telo	2016-09-27	Update for API 0.0.9

Glossary	
BLE	Bluetooth Low Energy
BTH	Bluetooth

2. Introduction

The PLUX Java Application Programming Interface (API) brings to java android applications all the functionalities of PLUX devices.

The API is implemented based on the utilization of the factory design pattern which provides the developer with a simple way to choose between the different types of communication available.

The class *BiopluxCommunication* contains the methods required to communicate with a device of the biosignalsplux range, as well as receiving information on their state. This device can be accessed through a classic Bluetooth connection or using Bluetooth Low Energy. The class *BiopluxException* implements the exception which is thrown by the API when an error condition is met while calling any of the API functions or class methods.

The API consists of a jar file (pluxapi-0.0.9.jar, where 0.0.9 is the code version name) containing the API object code.

3. Main Objects

There are four types of objects that are important to make a proper use of the PLUX's API:

3.1 . Class EventData

Object that has the event data received from the bluetooth device. There are two types of events: the battery event and the onBody event.

3.1.1 . Methods:

getIdentifier()

parameters:

[out]

[String] identifier - returns the MAC address of the device that sent the event

getEventDescription()

parameters:

[out]

[String] eventDescription - returns the description of the event sent

getBatteryLevel()

parameters:

[out]

[int] batteryLevel - returns the battery level of the bluetooth device

getOnBody()

parameters:

[out]

[int] onBody - returns 1 if device is on body, returns 0 otherwise

3.2 . Class BiopluxFrame

Object that has the data received from the bluetooth device.

3.2.1 . Methods:

getIdentifier()

parameters:

[out]

[String] identifier – returns the MAC address of the device that sent the frame

getSequence()

parameters:

[out]

[int] sequence – returns the sequence number of the frame

getAnalogData()

parameters:

[out]

[int[]] analogData – returns an array with all the channel values of the frame

getDigitalInput()

parameters:

[out]

[int] digitalInput – returns the value of the digital input

3.3 . Class PluxDevice

Object that has the general information about the PLUX's device.

3.3.1 . Methods:

getProductName()

parameters:

[out]

[String] productName - returns the device's name

getFirmwareVersion()

parameters:

[out]

[int] firmwareVersion - returns the device's firmware version

getHardwareVersion()

parameters:

[out]

[int] harwareVersion - returns the device's hardware version

getProductIdentifier()

parameters:

[int]

[String] productIdentifier - returns the device's product identifier

3.4 . Class Source

Source(int port, int nBits, byte channelMask, int freqDivisor)

Object that allows the user to set the device's channel acquisition parameters.

3.4.1 . Arguments:

- port – sensor port number;
- nBits – sensor samples resolution;
- channelMask – sensor channels to acquire in a bitmask
- freqDivisor – sensor frequency divisor

3.4.2 . Description:

The freqDivisor parameter is the size of the window used for the envelope calculation and its respective subsampling.

4. Main Classes

4.1. Class BiopluxCommunicationFactory

4.1.1 . Overview

This class allows the user to select from the possible types of communication.

4.1.2 . Methods

getCommunication (Communication type, Context context)

parameters:

[in]

type – type of communication selected (BTH or BLE)

context – application's activity

[out]

biopluxCommunication – BiopluxCommunication object, with the selected type of communication associated to it.

description:

Returns the object with the chosen type of communication.

getCommunication (Communication type, Context context, OnBITalinoDataAvailable callback)

parameters:

[in]

type – type of communication selected (BTH or BLE)

context – application's activity

callback – callback that receives the BITalino frames (BTH only)

[out]

bitalinoCommunication – BITalinoCommunication object, with the selected type of communication associated to it.

Description:

Returns the object with the chosen type of communication.

4.2 . Class BiopluxCommunication

4.2.1 . Overview

This class includes all the methods that compose the communication channel with the bioplux device. It is through this object that the developer can call the methods to set the acquisition parameters or start and stop an acquisition using a bioplux device.

The communication between the android device and the bioplux device is asynchronous, using a local broadcast or a callback to send the received data and several events from a lower level to the higher level (like an activity of the app).

4.2.2 . Methods

void scan()

description:

Scans for PLUX's devices in the near area. This method has an asynchronous response that returns an object (BluetoothDevice) with the device's name and MAC address.

boolean connect(String address)

parameters:

[in]

address – Media Access Control (MAC) address, the unique identifier of the device.

[out]

[boolean] true if the connection to the device is successful, false otherwise

description:

Tries to connect to the device with the given MAC address.

void disconnect()

parameters:

[out]

[boolean] true if the device is disconnected successfully, false otherwise

description:

Disconnects the device and closes the connection channel created.

boolean start(float baseFreq, List<Source> sources)

parameters:

[in]

baseFreq – sample frequency that is intended to the signal acquisition

sources – acquisition parameters

[out]

[boolean] true if the command is sent successfully to the bluetooth device, false otherwise

description:

Starts the acquisition mode of the device. An exception is thrown if the device is already acquiring. The baseFreq parameter must be between 1 Hz and 1000 Hz.

On acquisition mode, the frames sent by the bluetooth device are received asynchronously by the phone and then sent to the local broadcast receiver, using the *BiopluxFrame* object.

boolean stop()

parameters:

[out]

[boolean] true if the command is sent successfully, false otherwise

description:

Stops the acquisition mode in the device. An exception is throw if the acquisition mode is not active.

boolean getDescription()

parameters:

[out]

[boolean] true if the command is sent successfully, false otherwise

description:

Get the connected device description.

boolean getVersion()

parameters:

[out]

[boolean] true if the command is sent successfully, false otherwise

description:

Get the connected device's firmware and hardware version.

boolean reset()

parameters:

[out]

[boolean] true if the command is sent successfully, false otherwise

description:

Calls for a programmatical reset of the bluetooth device.

5. Data and Events Asynchronous Communication

All the information needed on the client side can be received by the user with a Broadcast Receiver as follows:

```
private final BroadcastReceiver mUpdateReceiver = new BroadcastReceiver() {
     @Override
     public void onReceive(Context context, Intent intent) {
         final String action = intent.getAction();
         if (Constants.ACTION_STATE_CHANGED.equals(action)) {
             String identifier = intent.getStringExtra(Constants.IDENTIFIER);
             Constants.States state =
   Constants.States.getStates(intent.getIntExtra(Constants.EXTRA_STATE_CHANGED,0);
             Log.i(TAG, "Device " + identifier + ": " + state.name());
         } else if (Constants.ACTION_DATA_AVAILABLE.equals(action)) {
             BiopluxFrame frames = intent.getParcelableExtra(Constants.EXTRA_DATA);
             Log.d(TAG, "BiopluxFrame: " + frames.toString());
         } else if (Constants.ACTION_COMMAND_REPLY.equals(action)) {
             String identifier = intent.getStringExtra(Constants.IDENTIFIER);
             Parcelable parcelable = intent.getParcelableExtra(Constants.EXTRA_COMMAND_REPLY);

             if(parcelable.getClass().equals(PluxDevice.class)){
                 Log.d(TAG, "PluxDevice: " + parcelable.toString());
             } else if(parcelable.getClass().equals(EventData.class)){
                 Log.d(TAG, "EventData: " + parcelable.toString());
             } else if(parcelable.getClass().equals(CommandReplyString.class)){
                 Log.d(TAG, "CommandReplyString: " + CommandReplyString.toString());
             }
         } else if (Constants.ACTION_EVENT_AVAILABLE.equals(action)) {
             EventData event = intent.getParcelableExtra(Constants.EXTRA_EVENT);
             String str = '';
             if(event.eventDescription.equals(Constants.ON_BODY_EVENT)){
             Log.i(TAG, "[" + event.identifier + "] " + "OnBody: " + false);
             }
             if(event.eventDescription.equals(Constants.BATTERY_EVENT)){
                 Log.i(TAG, str + "Battery Level: " + event.batteryLevel);
             }
         } else if (Constants.ACTION_DEVICE_READY.equals(action)) {
             String identifier = intent.getStringExtra(Constants.IDENTIFIER);
             PluxDevice pluxDevice = intent.getParcelableExtra(Constants.PLUX_DEVICE);
             Log.d(TAG, pluxDevice.toString());
             Toast.makeText(getApplicationContext(), "PluxDevice " + identifier + ": READY",
   Toast.LENGTH_LONG).show();
         } else if (Constants.ACTION_MESSAGE_SCAN.equals(action)){
             BluetoothDevice device = intent.getParcelableExtra(Constants.EXTRA_DEVICE_SCAN);
```

```
        }
    }
};
protected static IntentFilter updateIntentFilter() {
      final IntentFilter intentFilter = new IntentFilter();
      intentFilter.addAction(Constants.ACTION_STATE_CHANGED);
      intentFilter.addAction(Constants.ACTION_DATA_AVAILABLE);
      intentFilter.addAction(Constants.ACTION_COMMAND_REPLY);
      intentFilter.addAction(Constants.ACTION_EVENT_AVAILABLE);
      intentFilter.addAction(Constants.ACTION_DEVICE_READY);
      intentFilter.addAction(Constants.ACTION_MESSAGE_SCAN);
 return intentFilter;
}
```

The following table contains the descriptions of all the intents regarding the data and event communication, as well as their occurring scenarios.

Intent Action	Scenario
Constants.ACTION_STATE_CHANGED	Bluetooth connection state changed, this intent has the MAC Address of the device as an extra. *EXTRAS:* [String] Constants.IDENTIFIER [Constants.States] Constants.EXTRA_STATE_CHANGED
Constants.ACTION_DATA_AVAILABLE	This intent is sent when the android device receives the acquired data from the bluetooth device and then the API decodes it and sends an object (BiopluxFrame) as an extra. This object has also the device's MAC Address. EXTRAS: [Parcelable] Constants.EXTRA_DATA
Constants.ACTION_COMMAND_REPLY	This intent is sent when the android device receives a reply from the bluetooth device commands through the API. This intent has two extras, the device's MAC Address and an object (Parcelable) that can be a PluxDevice object, a EventData object or a CommandReplyString object. EXTRAS: [String] Constants.IDENTIFIER [Parcelable] Constants.EXTRA_COMMAND_REPLY
Constants.ACTION_EVENT_AVAILABLE	This intent is sent when the android device receives an

	event from the bluetooth device and then interprets it and sends an object (EventData) as an extra. This object has also the device's MAC Address. EXTRAS: [Parcelable] Constants.EXTRA_Event
Constants.ACTION_DEVICE_READY	This intent arrives as soon as the device is ready to receive commands through the API. This intent has two extras, the device's MAC Address and an object (PluxDevice) that has the version of the device. EXTRAS: [String] Constants.IDENTIFIER [Parcelable] Constants.PLUX_DEVICE
Constants.ACTION_MESSAGE_SCAN	After the call of the API method scan, the available PLUX's devices will be sent with this intent. This intent has an object as an extra (BluetoothDevice). EXTRAS: [BluetoothDevice] Constants.EXTRA_DEVICE_SCAN

6. Implementation Example [BLE]

1. Initialize the PLUX API:

```
BiopluxCommunication bioplux = new BiopluxCommunicationFactory
().getCommunication(Communication.BLE, getBaseContext());
```

2. Register your broadcast receiver:

```
registerReceiver(mUpdateReceiver, updateIntentFilter());
```

3. Connect to the selected device:

```
bioplux.connect(AA:BB:CC:DD:EE:FF);
```

4. After the ready event is received on the broadcast receiver, you can call the other API methods. There is an example below:

```
//Begin acquisition with a base frequency of 1000 Hz and a frequency divisor of 100 Hz

List<Source> sources = new ArrayList<>();

Source emgSource = new Source(1, 16, (byte)0x01, 100);
Source inertialSource = new Source (2, 16, (byte)0x3F, 100);
sources.add(emgSource);
sources.add(inertialSource);

bioplux.start(1000f, sources);
try
{
   Thread.sleep(10000);
} catch (Exception e){
   e.printStackTrace();
}
bioplux.stop();
bioplux.disconnect();
```

7. Implementation Example [BTH]

1. Initialize the PLUX API:

```
BiopluxCommunication bioplux = new BiopluxCommunicationFactory
().getCommunication(Communication.BTH, getBaseContext(), new OnBiopluxDataAvailable(){
    @Override
    public void onBiopluxDataAvailable(BiopluxFrame biopluxFrame) {
        Log.d(TAG, "BiopluxFrame: " + biopluxFrame.toString());
    }
});
```

2. Register your broadcast receiver:

```
registerReceiver(mUpdateReceiver, updateIntentFilter());
```

3. Connect to the selected device:

```
bioplux.connect(AA:BB:CC:DD:EE:FF);
```

4. After the ready event is received on the broadcast receiver, you can call the other API methods. There is an example below:

```
//Begin acquisition with a base frequency of 1000 Hz and a frequency divisor of 100 Hz

List<Source> sources = new ArrayList<>();

Source emgSource = new Source(1, 16, (byte)0x01, 100);
Source inertialSource = new Source (2, 16, (byte)0x3F, 100);
sources.add(emgSource);
sources.add(inertialSource);

bioplux.start(1000f, sources);
try
{
   Thread.sleep(10000);
} catch (Exception e){
   e.printStackTrace();
}
bioplux.stop();
bioplux.disconnect();
```