Movella™

Document XD0201P, Revision A, July, 2023

Movella Dot SDK Programming Guide for Android

Revision	Date	By	Changes
A	July 2023	ERI, MRA	Movella DOT Rebranding

Table of Contents

List of Tables

List of Figures

1 Introduction

The Movella Dot Android SDK is a software development kit for Android mobile applications. Android developers can use this SDK to build their applications to scan and connect the sensors, get data in real-time streaming or recording, as well as other functions.

This document mainly addresses SDK usage with example codes. It should be used together with *Movella Dot SDK core documentation* with detailed information. Before getting started with the SDK, it's advised to read *Movella Dot User Manual* first to understand the basic functions of the sensor.

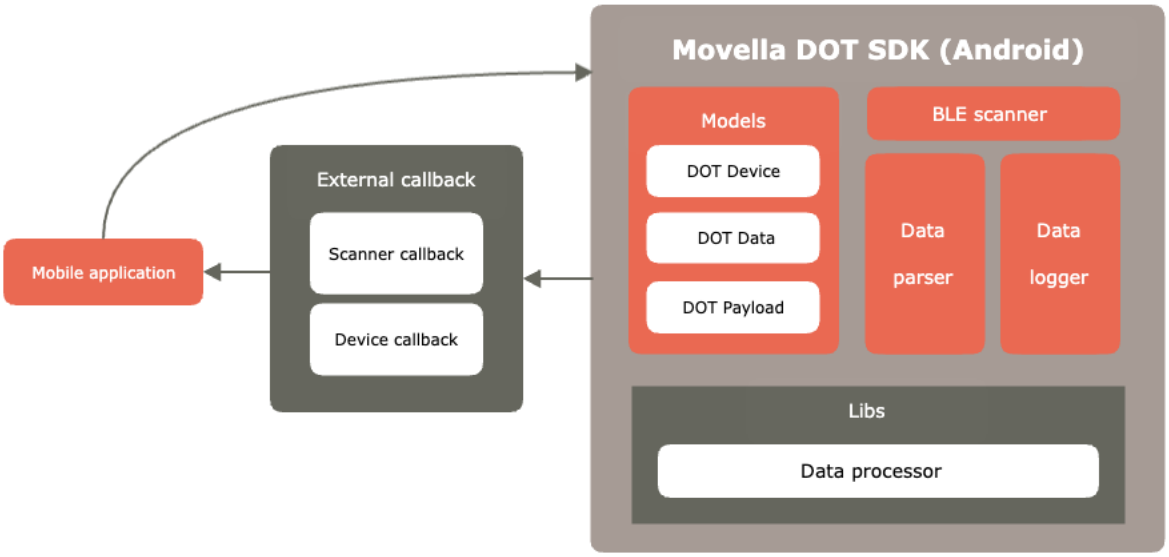

Figure 1: Movella Dot Mobile SDK Architecture

The SDK provides some public classes for developers to facilitate easier integration into specific application. Figure 1 shows the SDK architecture and components. It contains 3 main models to manage the state of device, data payload types and data output. It also contains different classes[1] available for usage. The data processor library is integrated in SDK to process the data from firmware. Other libraries like sensor fusion and calibration libraries are running on Movella Dot firmware.

[1] Not every class can be new or referenced

2 Getting Started

2.1 Platform Requirements

Table 1 shows the Android OS, CPU architecture and Bluetooth requirements for the mobile devices.

Table 1: Platform requirements

Platform requirements

• Android OS 8.0 and above, arm64-v8a ABI (64-bit)
• ARMv8 CPU architecture
• Bluetooth
  ○ Best performance with BLE 5.2 DLE[2] supported
  ○ Compatible with Bluetooth 4.2

2.2 Example code

Refer to this project on GitHub for the Android example code of Movella Dot SDK: https://github.com/MovellaTechnologies/dot_example

2.3 SDK Changelogs

Refer to this BASE article for the Android SDK changelogs.

2.4 Prerequisites for Android Studio Project

This section addresses setup parameters for proper usage of the Movella Dot Android SDK. Make sure the following configurations are met when creating the Android Studio project.

1. Make sure the **`minSdkVersion`** is 26+ (Android 8.0) in the build.gradle (app level) file
2. Use androidx.* artifacts
3. Dependency workmanager: implementation "androidx.work:work-runtime:2.5.0"
4. Dependency lifecycle: implementation "androidx.lifecycle:lifecycle-runtime:2.3.1"

2.5 Import SDK Package

This section addresses setup parameters and some practical considerations for proper usage of the Movella Dot SDK. The following steps describe how to import the SDK object into your Android Studio project.

1. Import the AAR file according to the steps in the Add your AAR or jar as a dependency section from Android developer page.
2. After **`Build finished`**, you can do some basic settings of the SDK as shown below.

```
private void initXsSdk() {
```

[2] Data Length Extension

```
    String version = DotSdk.getSdkVersion();

    DotSdk.setDebugEnabled(true);
    DotSdk.setReconnectEnabled(true);
}
```

2.6 Implement Interface

Developers can implement *DotDeviceCallback* and *DotScannerCallback* in one activity as shown below.

```
public class MainActivity extends AppCompatActivity
                    implements DotDeviceCallback, DotScannerCallback {
  ...
  ...
  ...
}
```

If IDE shows an error message, click the line and press **Alt + Enter** to choose **Implements methods**, the IDE will generate all the required methods that need to be implemented automatically.

3 Classes and Interfaces

The list of classes as part of Movella Dot SDK is shown in Table 2.

Table 2: Classes in Movella Dot SDK

Class	Description
DotSdk	The SDK main object used for global settings such as enable debug or reconnect features.
DotDevice	Represents a Movella Dot device object, including basic information and operations. Return data by DotDeviceCallback.
DotData	Contains all the measurement data, including acceleration, angular velocity, and mag data, etc.
DotLogger	A class for data logging.
DotParser	A class for parsing data from the device via Bluetooth.
DotScanner	A class for scanning Movella Dot device. Return the scanned device by DotScannerCallback.
XsPayload	For setting different payload types for measurement.
DotRecordingManager	Data recording manager, including data recording, and exporting methods.
DotSyncManager	Synchronization manager for sensors' syncing
DotSettingsManager	Settings Manager primarily handling Factory Reset functionality

3.1 Interfaces

The list of available interfaces as part of Movella Dot SDK is shown in Table 3.

Table 3: Interfaces in Movella Dot SDK

Class	Description
DotDeviceCallback	An interface for notifying device information, measurement data.
DotScannerCallback	An interface for notifying LE scan result
DotMeasurementCallback	An interface for notifying measurement status and data.
DotCiCallback	An interface for notifying firmware crash information result.
DotRecordingCallback	An interface for notifying recording status and data of device.
DotSyncCallback	An interface for synchronization result.
SettingsCallback	An interface for notifying the result of factory resetting of the Dot device.

3.2 Permissions

The permissions used by this SDK are as listed in Table 4. Make sure these permissions are set and are part of AndroidManifest.xml file in your project.

Table 4: Permissions list

Permission	Purpose
BLUETOOTH	For connecting to sensor
BLUETOOTH_ADMIN	For connecting to sensor

ACCESS_FINE_LOCATION	For LE scanning
ACCESS_COARSE_LOCATION	For LE scanning
READ_EXTERNAL_STORAGE	For storing the log file
WRITE_EXTERNAL_STORAGE	For storing the log file

4 SDK Usage with Examples

4.1 Recommended workflow

The Android SDK workflow is shown in Figure 2. This flow process can be used by Android developers after importing SDK library into Android project and creating an SDK object.

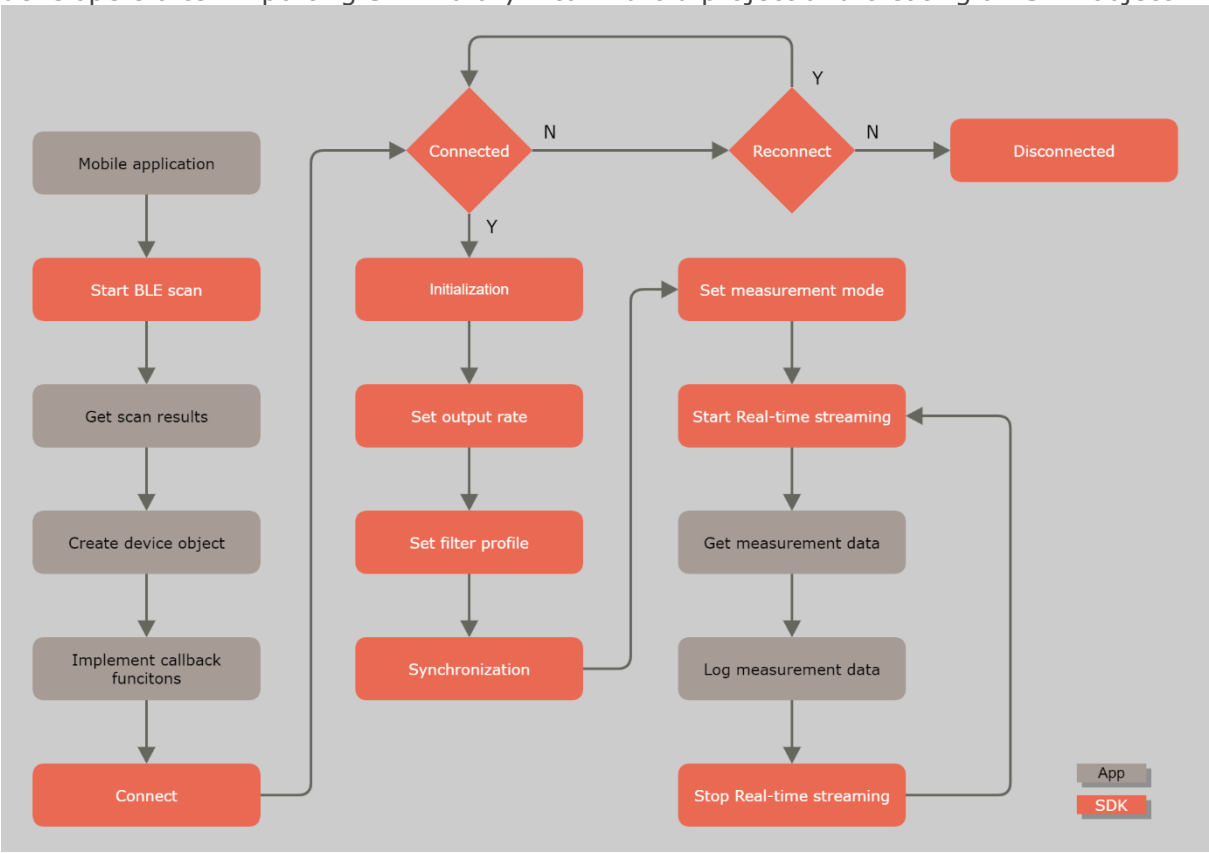

Figure 2: Movella Dot Android SDK Workflow

The first thing is to start BLE scanning. Developers can obtain the scan result from a callback function and use the *BluetoothDevice* object to initialize *DotDevice* class. Most of the operations can be done by making use of this class.

Developers can call the connect function in *DotDevice* class to connect to the sensors. If the connection process fails, SDK will check if the reconnection feature is enabled or not. If it is enabled, a reconnection will start automatically.

Each step is further explained in the following sections with example code.

4.2 Debugging flag

This is a static function and can be used to enable/disable the debug messages. If set to true, the SDK will output debug message with this tag – DotSdk.

```
DotSdk.setDebugEnabled(true);
```

This setting is disabled by default.

4.3 Reconnection setting

This is a static function and can be used to enable/disable the reconnection feature. If set to true, the SDK will start to reconnect the sensor(s) automatically when the connection is lost.

```
DotSdk.setReconnectEnabled(true);
```

You can cancel the reconnecting by:

```
xsDevice.cancelReconnecting();
```

4.4 BLE scan

To use this, declare a *DotScanner* object and try to initialize. There are two additional parameters that need to be put in the constructor - application context and an instance of *DotScannerCallback* (i.e., an activity that implemented the DotScannerCallback interface).

The mode can be one of these: *SCAN_MODE_BALANCED*, *SCAN_MODE_LOW_LATENCY* or *SCAN_MODE_LOW_POWER*.

```
private DotScanner mXsScanner;

private void initXsScanner() {

    mXsScanner = new DotScanner(mContext, this);
    mXsScanner.setScanMode(ScanSettings.SCAN_MODE_BALANCED);
}
```

To start the LE scanning, the function below should be called.

```
mXsScanner.startScan();
```

The scanned result can be obtained by using the *onDotScanned* callback function. Note that only Movella Dot device is reported.

```
@Override
public void onDotScanned(BluetoothDevice device) {

    String name = device.getName();
    String address = device.getAddress();
    …
}
```

4.5 Connect

Declare a *DotDevice* object and use the following parameters to initialize - the application context, *BluetoothDevice* object and an instance of *DotDeviceCallback* (i.e., an activity that implemented *DotDeviceCallback* interface).

```
DotDevice xsDevice =
new DotDevice(mContext, device, MainActivity.this);
```

Then use the following function to connect to the device.

```
xsDevice.connect();
```

As a best practice, it is preferred to check whether the device's name is null or not before you connect to it. After connecting, the *onDotConnectionChanged* callback function will be triggered. If the state equals *CONN_STATE_CONNECTED*, it means the Bluetooth GATT connection is successful after which all BLE services/characteristics will be discovered automatically.

The state of service discovery can be checked from *onDotServicesDiscovered* callback function.

```
@Override
public void onDotConnectionChanged(String address,
                                   int state) {

    if (state == DotDevice.CONN_STATE_DISCONNECTED) {

        // Update UI
    }
}

@Override
public void onDotServicesDiscovered(String address,
                                    int status) {

    if (status == BluetoothGatt.GATT_SUCCESS) {

        // Update UI
    }
}
```

Once the connection is successful, device information can be obtained using the following methods, including:

- getName
- getAddress
- getConnectionState
- getFirmwareBuildTime
- getFirmwareVersion
- getBatteryState
- getBatteryPercentage
- getMeasurementMode
- getMeasurementState
- getPlotState
- getLogState

- getTag
- getCurrentOutputRate
- getFilterProfileInfoList
- isSynced
- isProductV1
- isProductV2
- …

The following function call can be used to disconnect the device.

```
xsDevice.disconnect();
```

We add product id to distinguish different hardware versions.

```
xsDevice.isProductV1();
xsDevice.isProductV2();
```

4.5.1 Connect multiple sensors

To connect multiple sensors, the *DotDevice* object can be put into a list under one class.

```
private ArrayList<DotDevice> mDeviceLst = new ArrayList<>();
```

After initiating connection to this device, one can add this object to the list to get the connection result from *onDotConnectionChanged* callback function.

```
DotDevice xsDevice = new DotDevice(
                                mContext, device, MainActivity.this);
xsDevice.connect();
mDeviceLst.add(xsDevice);
```

To disconnect one device, use the key variable - **`address`** to get the device object from the list and then call disconnect method. It is very important to make sure that the *DotDevice* will be removed from the list after the device is disconnected. In a similar way, put *DotLogger* object into a list to manage data collecting and logging for multiple devices.

4.6 Initialization

After the sensor connection, an initialization process will start automatically to enable BLE notifications and obtain basic sensor information, including the hardware and firmware version, MAC address, tag name, battery status, synchronization status, filter profile, output rate etc. *onDotInitDone()* is the callback after the initialization is successful.

NOTE:

- Any read or write operation can only be called after a successful initialization.

```
public void onDotInitDone(String address) {
    // get tag name, version, battery info etc.
    DotDevice.getFirmwareVersion();
    DotDevice.getTag();
    DotDevice.getBatteryPercentage();
    ...
    DotDevice.identifyDevice();
}
```

4.7 Filter profile

After the initialization is done, you can get or set the current filter profile for the measurement. Refer to section 3.2 in the User Manual for more information about filter profiles.

4.7.1 Get current filter profile

Get the current filter profile that is applied in the measurement:

```
int profileIndex = DotDevice.getCurrentFilterProfileIndex();
```

The *profileIndex* is the index of the current selected filter profiles.

Table 5: Filter profile index

Index	Filter profile	Description
0	General	This filter profile is the default setting. It assumes moderate dynamics and a homogeneous magnetic field. External magnetic distortion is considered relatively short.
1	Dynamic	This filter profile assumes fast and jerky motions that last for a short time. The dynamic filter uses the magnetometer for stabilization of the heading and assumes very short magnetic distortions. Typical applications are when sensors are applied on persons for sports such as sprinting.

You can also get the current filter profile from callback:

```
SomeClass implements DotDeviceCallback {

    public void onDotFilterProfileUpdate(String address, int
filterProfileIndex) {
        ...
    }
}
```

4.7.2 Get all the filter profiles

Get all the supported filter profiles through *getFilterProfileInfoList*:

```
ArrayList<FilterProfileInfo> list = DotDevice.getFilterProfileInfoList();
```

You can also get this list from the callback during initialization:

```
SomeClass implements DotDeviceCallback {

    public void onDotGetFilterProfileInfo(String address,
ArrayList<FilterProfileInfo> filterProfileInfoList) {
        ...
    }
}
```

4.7.3 Set a new filter profile

Before setting a new filter profile, get the index first.

```
int profileIndex = list.get(0).getIndex();
int profileIndex = FilterProfileInfo.getIndex();
```

Set current filter profile with the index from the profile list.

```
DotDevice.setFilterProfile(int profileIndex);
```

onDotFilterProfileUpdate() callback will be triggered if the new filter profile is set successfully.

4.8 Output rate

After the initialization is done, you can get or set the output rate for the measurement by *DotDevice* class. Table 6 shows the available output rates during measurements.

Table 6: Output rates

Measurement	Output rates
Real-time streaming	1 Hz, 4 Hz, 10 Hz, 12 Hz, 15 Hz,20 Hz, 30 Hz, 60 Hz
Recording	1 Hz, 4 Hz, 10 Hz, 12 Hz, 15 Hz,20 Hz, 30 Hz, 60 Hz, 120 Hz

Get the current output rate that is applied in the measurement:

```
int outputRate = DotDevice.getCurrentOutputRate();
```

You can also get output rate from the callback during initialization:

```
SomeClass implements DotDeviceCallback {

    public void onDotOutputRateUpdate(String address, int outputRate) {
        ...
    }
}
```

Set a new output rate for the measurement:

```
DotDevice.setOutputRate(int outputRate);
```

onDotOutputRateUpdate() callback will be triggered if the new output rate is set successfully.

4.9 Synchronization

All sensors will be time-synced with each other to a common time base after synchronization. Refer to section 3.3.2 in *Movella Dot User Manual* for more information. Refer to Figure 3 for workflow to start synchronization.

Set the output rate and filter profile before starting the synchronization. Since the sensor will enter measurement mode right after the sync succeeds so it's not possible to change it after sync.

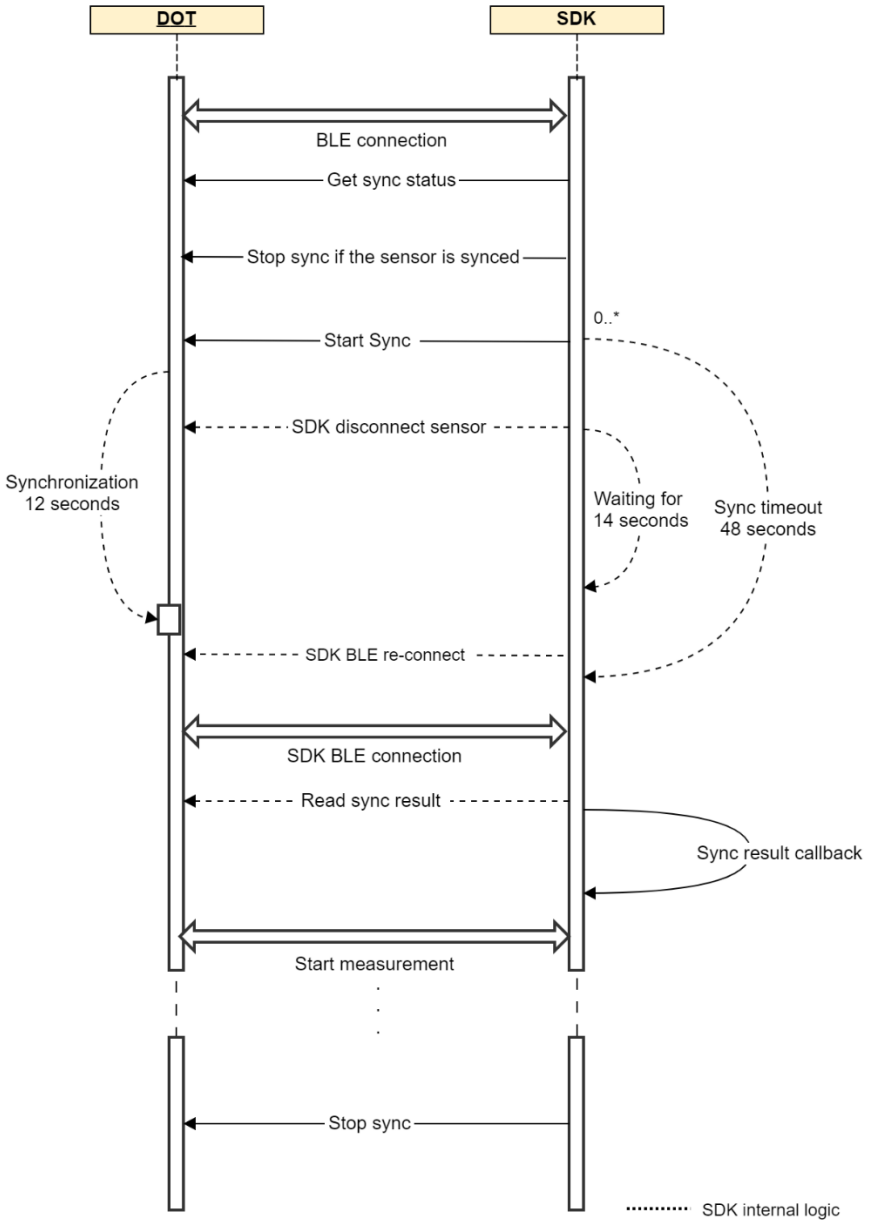

Figure 3: Synchronization workflow

First implement the *DotSyncCallback* interface.

```
public class RecordingFragment implements DotSyncCallback {

    public void onSyncingProgress(int progress, int requestCode) {
    }

    public void onSyncingResult(String address, boolean isSuccess, int
requestCode)  {
    }

    public void onSyncingDone(HashMap<String, Boolean> syncingResultMap,
boolean isSuccess, int requestCode) {
    }

    public void onSyncingStopped(String address, boolean isSuccess, int
requestCode) {
    }
}
```

4.9.1 Get sync status

Before starting the synchronization, check the synchronization status of the target sensors and make sure they are not synced. Stop the synced sensor before starting a new synchronization to prevent error status.

After the initialization is done, you can get sync status by DotDevice class:

```
boolean isSynced = DotDevice.isSynced();
```

You can also get sync status from the callback function during initialization:

```
SomeClass implements DotDeviceCallback {
    public void onSyncStatusUpdate(String address, boolean isSynced) {
        ...
    }
}
```

4.9.2 Start sync

One of the sensors must be set as the root sensor before starting synchronization, while the remaining sensors will be the scanners:

```
mSelectedDeviceList.get(0).setRootDevice(true);
```

Start the synchronization. *mSelectedDeviceList* is the list of sensors that need to be synchronized.

```
DotSyncManager.getInstance(this).startSyncing(mSelectedDeviceList,
SYNCING_REQUEST_CODE)
```

onSyncingProgress(int progress, int requestCode) is the callback during synchronization. The sync process is updated via "progress".

SDK will disconnect the sensor after starting the synchronization. The synchronization of 5 sensors would take about 12 seconds so SDK will try to reconnect after 14 seconds.

NOTE:

- Do not interrupt during the synchronization process.

4.9.3 Get sync results

onSyncingResult() function will be called if one sensor has finished the synchronization. If all the sensors finish the synchronization, *onSyncingDone()* function will be called back. *syncingResultMap* contains the sync results of the device list in *startSyncing()*. *isSuccess* represents whether the synchronization is successful or not.

Once the sync succeeds, sensor will enter measurement mode. You can then choose to do real-time streaming or recording with the synced sensors.

If the sensor is not reconnected within 48 seconds, the sync is considered as failed. The sync is also failed if the result shows fail. In that case, SDK will stop those sensors that have been successfully synced. Refer to this BASE article for more tips about synchronization.

4.9.4 Stop sync

Stop sync is required after the measurement. Otherwise, the sensor will stay in measurement mode and the battery will run out soon.

You can stop the synchronization for all the synced sensors:

```
DotSyncManager.getInstance(this).stopSyncing();
```

Or stop the synchronization for specific sensors.

```
DotSyncManager.getInstance(this).stopSyncing(dotDevices);
```

After one sensor is stopped, you will get the callbacks to indicate the synchronization is stopped and the sync status is updated.

```
    public void onSyncingStopped(String address, boolean isSuccess,
int requestCode) {
        ...
    }

    public void onSyncStatusUpdate(String address, boolean isSynced) {
        ...
    }
```

4.10 Real-time streaming

In real-time streaming, motion data is streamed and logged to the central device via a constant Bluetooth connection. You can set measurement mode, start/stop measurement and log the data to csv files with the SDK.

Figure 4 shows the workflow to start and stop real-time streaming.

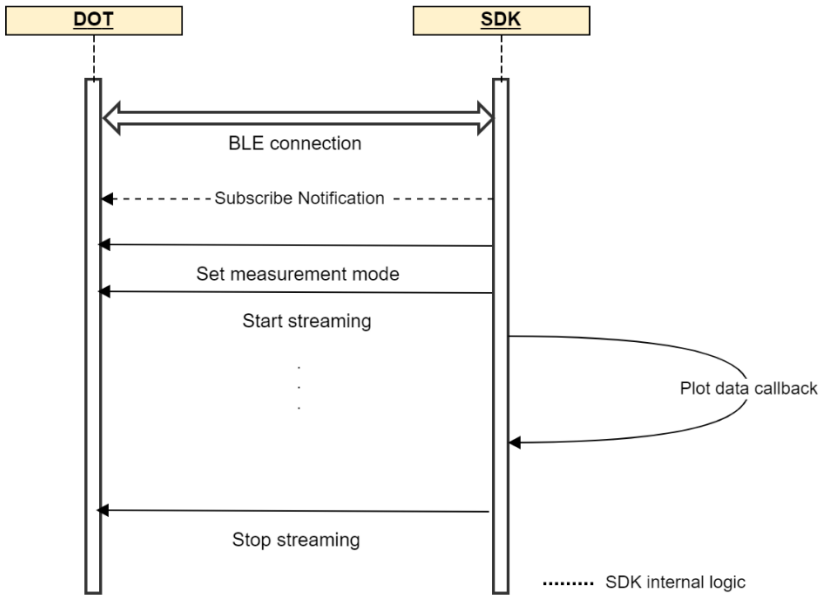

Figure 4: Workflow to start and stop real-time streaming

The *DotDevice* can report sensor data via *onDotDataChanged* callback function. To use this, notify the sensor to enter the measurement mode, then start the measurement by following the steps below.

4.10.1 Set measurement mode

There are 17 measurement modes with different payload modes. Refer to Appendix for data outputs of different modes. Section 4.2 in *Movella Dot User Manual* also gives detailed explanation about output values.

```
xsDevice.setMeasurementMode(XsPayload.PAYLOAD_TYPE_HIGH_FIDELITY_NO_MAG);
```

4.10.2 Start measurement

Then call this function to start measuring.

```
xsDevice.startMeasuring();
```

The measuring data can be received from *onDotDataChanged* callback function.

```
@Override
public void onDotDataChanged(String address,
```

```
                                    DotData DotData) {
}
```

The *address* variable can be used to help identify the device for data association. The *DotData* object contains all measuring data, timestamp, and the packet counter information. The following methods from *DotData* object can be used to get these data outputs according to the measurement mode.

- getAcc()
- getGyr()
- getDq()
- getDv()
- getMag()
- getEuler()
- getQuat()
- getSampleTimeFine()
- getPacketCounter()
- …

The *DotData* object has implemented the `Parcelable` object from Java, so this object can be passed to another class by *Broadcast* event.

4.10.3 Stop measurement

The following function call can be used to stop the measurement.

```
xsDevice.stopMeasuring();
```

4.10.4 Data logging

The *DotLogger* class provides a way to log measurement data to the SD card of mobile devices. Try to initialize this object with the full file path. After this object is created, it will write a default title string of each column and save to csv file.

```
DotLogger xsLogger = new DotLogger(
        Environment.getExternalStorageDirectory() + "/YOUR_DIR/");
```

The following function can be used to update the file content:

```
public void update(DotData xsData)
```

Make sure the data output stream is closed before you stop measuring. You can call this function to flush and close the stream.

```
xsLogger.stop();
```

4.10.5 High fidelity modes

In high fidelity mode, higher frequency (800 Hz) information is preserved with lower output data rate (60 Hz), even with transient data loss. There are 3 measurement modes containing high fidelity inertial data in the SDK:

- PAYLOAD_TYPE_HIGH_FIDELITY_WITH_MAG
- PAYLOAD_TYPE_HIGH_FIDELITY_NO_MAG

- PAYLOAD_TYPE_CUSTOM_MODE_4

To parse the high fidelity inertial data to to delta_q, delta_v or calibrated angular velocity and acceleration, you need to select the above measurement modes with high fidelity inertial data. After starting the measurement, you can get the values with getAcc(), getGyr(), getDq(), getDv() methods from *DotData* object.

```
DotDevice DotDevice = ...;
//set measurement mode
DotDevice.setMeasurementMode(XsPayload.PAYLOAD_TYPE_HIGH_FIDELITY_WITH
_MAG);

//start measurement
DotDevice.startMeasuring();

public void onDotDataChanged(String address, DotData data) {
    final double[] acc = data.getAcc();
    final double[] gyr = data.getGyr();
    final double[] dq = data.getDq();
    final float[] dv = data.getDv();
    ...
}
```

4.10.6 Data conversions

Data conversion functions are provided in Movella Dot SDK. Developers can make use of these conversion functions to get the measurement quantities as required in their applications.

Convert dq, dv to angular velocity and acceleration

You can get dq and dv outputs from onDotDataChanged callback in some measurement modes (e.g. *PAYLOAD_TYPE_DELTA_QUANTITIES_WITH_MAG*).

You can also set other values to dq and dv as following:

```
DotData xsData = ...;
xsData.setDq(…);
xsData.setDv(…);
```

If there are dq and dv in *DotData* object, the default data processor can be used to convert dq, dv to angular velocity and acceleration.

```
private DataProcessor mDataProcessor =
DotParser.getDefaultDataProcessor();
```

```
XsDataPacket packet = DotParser.getXsDataPacket(mDataProcessor,
xsData.getDq(), xsData.getDv());
```

Then use the output packet to get the angular velocity and acceleration.

```
final double[] acc = DotParser.getCalibratedAcceleration(packet);
final double[] gyr = DotParser.getCalibratedGyroscopeData(packet);
```

Convert quaternion to Euler angles

quaternion2Euler() method is provided in *DotParser* class to convert quaternion values to Euler angles.

```
final float[] quats = DotData.getQuat();
final double[] eulerAngles = DotParser.quaternion2Euler(quats);
```

Calculation of free acceleration

You can get the free acceleration from orientation and acceleration as mentioned in this BASE article.

In real-time streaming, *getCalFreeAcc* function is provided to help you omit the mathematical calculations.

As this function requires both orientation (in quaternion) and acceleration as input, it can currently on be used with PAYLOAD_TYPE_CUSTOM_MODE_4. Custom mode 4 is the only mode that can output these two quantities at the same time.

```
DotData.getCalFreeAcc();
```

The default gravity is 9.8127 m/s^2. You can set a custom gravity vector (for example 9.82 m/s^2) by defining its value in the following way:

```
DotData.getCalFreeAcc(double localGravity);
```

4.11 Heading Reset

Heading reset function allows user to align heading outputs among all sensors and with the object they are connected to. Performing a heading reset will determine the orientation and free acceleration data with respect to a different earth-fixed local frame (L'), which defines the L' frame by setting the X-axis of L' frame while maintaining the Z-axis along the vertical. It computes L' such that Yaw becomes 0 deg.

The heading reset function must be executed during real-time streaming and with measurement mode including orientation output. The reset orientation is maintained between measurement start/stop and connection/disconnection but will be lost after a device reboot.

Figure 5 shows the workflow to do the heading reset.

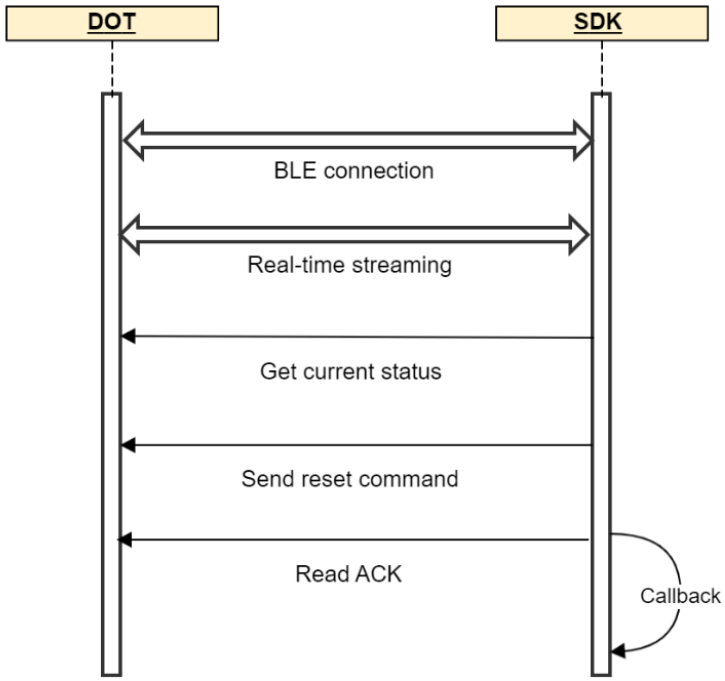

Figure 5: Workflow for heading reset

4.11.1 Heading reset status

Get the heading reset status of the sensor.

```
DotDevice.getHeadingStatus();
```

Refer to Table 7 for the heading reset status of the sensor.

Table 7: Heading status

Heading reset status	Description
HEADING_STATUS_XRM_HEADING	Heading has been reset

HEADING_STATUS_XRM_DEFAULT_ALIGNMENT	Heading has been reverted to default status
HEADING_STATUS_XRM_NONE	Default status

When the sensor is initially powered on, it is *HEADING_STATUS_XRM_NONE* by default。

4.11.2 Reset heading

Use following functions to perform heading reset.

```
DotDevice DotDevice = ...;
DotDevice.setDotMeasurementCallback(this);
//set to one sensor funsion mode
//start measurement
DotDevice.resetHeading();
```

You need to implement *DotMeasurementCallback* in one class and override the following two functions to obtain the result of heading reset.

```
public class MeasurementFragment implements DotMeasurementCallback {

      ...

      @Override
      public void onDotHeadingChanged(String address, int status, int
result) {

      }

      @Override
      public void onDotRotLocalRead(String address, float[] quaternions)
{

      }
}
```

If the heading reset is successful, status should be *HEADING_STATUS_XRM_HEADING*, and the result is *HEADING_SUCCESS* when *onDotHeadingChanged* is triggered.

4.11.3 Revert heading

Then revert heading to original value by calling this function.

```
DotDevice.revertHeading();
```

After reset the heading, a revert is required before conducting a new reset.

4.12 Recording

In recording mode, motion data is stored in the sensor internal storage and can be exported for post processing. Bluetooth connection is not required during recording. With the SDK, you can start/stop recording, set timed recording and export recording data.

Figure 6 shows the recommended workflow to start and stop recording with Android SDK.

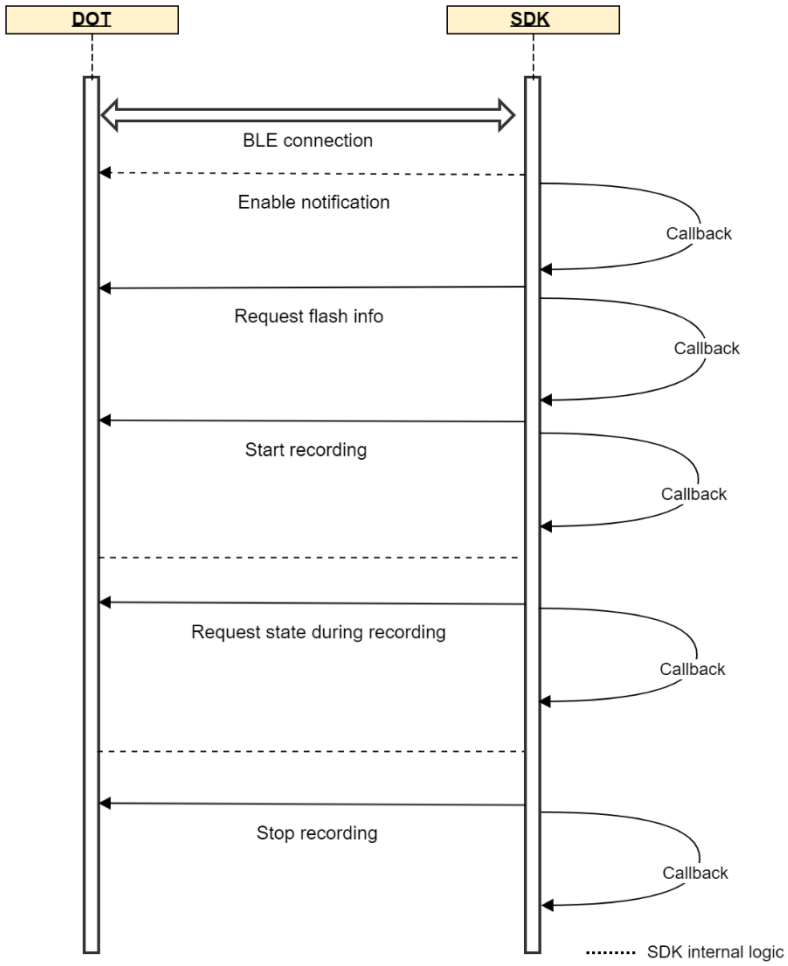

Figure 6: Workflow to start and stop recording

To perform the recording function, an *DotRecordingManager* needs to be instantiated. In most cases, one *DotDevice* uses one *DotRecordingManager*.

First you need to implement the *DotRecordingCallback* interface:

```
public class RecordingFragment implements DotRecordingCallback {

    public void onDotRecordingNotification(String address, boolean
isEnabled) {

    }
```

```
    public void onDotEraseDone(String address, boolean isSuccess) {

    }

    public void onDotRequestFlashInfoDone(String address, int
usedFlashSpace, int totalFlashSpace) {

    }

    public void onDotRecordingAck(String address, int recordingId,
boolean isSuccess, DotRecordingState recordingState) {

    }

    public void onDotGetRecordingTime(String address, int
startUTCSeconds, int totalRecordingSeconds, int
remainingRecordingSeconds) {

    }

    public void onDotRequestFileInfoDone(String address,
ArrayList<DotRecordingFileInfo> list, boolean isSuccess) {

    }

    public void onDotDataExported(String address, DotRecordingFileInfo
fileInfo, DotData exportedData) {

    }

    public void onDotDataExported(String address, DotRecordingFileInfo
fileInfo) {

    }

    public void onDotAllDataExported(String address) {

    }

    public void onDotStopExportingData(String address) {

    }

}
```

Then instantiate an *DotRecordingManager*, for example:

```
private DotRecordingManager mManager;
mManager = DotRecordingManager(context, DotDevice,
RecordingFragment.this);
```

Before performing recording-related operations, you need to enable Notification:

```
mManager.enableDataRecordingNotification();
```

Then wait for the callback, isEnabled indicates whether the notification is enabled or not.

4.12.1 Get flash information

Flash information refers to recording flash size and its usage. The flash size that can be used for recording accounts for about 90% of the total size (16 MB for v1 sensor, 64MB for v2 sensor). So firstly, we need to get the available flash space and the remaining recording time before start recording.

If the activation of notification is successful, you can get the flash info:

```
public void onDotRecordingNotification(String address, boolean
isEnabled) {

    if (isEnabled) {
        mManager.requestFlashInfo();
    }
}
```

Waiting for the callback to obtain the used space and the total space size:

```
public void onDotRequestFlashInfoDone(String address, int
usedFlashSpace, int totalFlashSpace) {

 // get usedFlashSpace & totalFlashSpace, if the available flash space
<= 10%, it cannot start recording

}
```

If the recording storage space is insufficient, clear flash storage space is needed. You can call *mManager.eraseRecordingData()* method and wait for the callback:

```
public void onDotEraseDone(String address, boolean isSuccess) {
    // do somethings
}
```

4.12.2 Start/stop recording

After getting the flash information of recording, you can call *mManager.startRecording()* to start recording. Timed recording is also supported with *mManager.startTimedRecording*. *recordingTimeSeconds* is the timer for timed recording and the unit is second. It should not exceed the maximum recording time (88 minutes).

Call *mManager.stopRecording()* method to stop recording. Recording will also stop automatically in the following situations:

- power button is pressed over 1 second.
- time is up for timed recording.
- flash memory is over 90%.

After start and stop, you need to wait for the callback result:

```
public void onDotRecordingAck(String address, int recordingId, boolean
isSuccess, DotRecordingState recordingState) {
```

```
    if (recordingId ==
DotRecordingManager.RECORDING_ID_START_RECORDING) {
        // start recording result, check recordingState, it should be
success or fail.
        ...
    } else if (recordingId ==
DotRecordingManager.RECORDING_ID_STOP_RECORDING) {
        // stop recording result, check recordingState, it should be
success or fail.
        ...
    }
}
```

4.12.3 Get recording status

You can check the recording status by calling *mManager.requestRecordingState()* and through *onDotRecordingAck()* callback.

```
public void onDotRecordingAck(String address, int recordingId, boolean
isSuccess, DotRecordingState recordingState) {

    if (recordingId == DotRecordingManager.RECORDING_ID_GET_STATE) {
        if (recordingState == DotRecordingState.onErasing
        || recordingState == DotRecordingState.onExportFlashInfo
        || recordingState == DotRecordingState.onRecording
        || recordingState ==
DotRecordingState.onExportRecordingFileInfo
        || recordingState ==
DotRecordingState.onExportRecordingFileData) {
            ...
        }
    }
}
```

Table 8: Recording status

Recording status	Description
XSRecordingIsIdle	Idle status
XSRecordingIsRecording	Sensor is recording
XSRecordingIsRecordingStopped	Recording is stopped
XSRecordingIsErasing	Erasing recording data
XSRecordingIsFlashInfo	Sensor is getting flash information

4.12.4 Get recording time

You can check how long the sensor has been recording in normal or timed recording by calling mManager.requestRecordingTime(), via onDotGetRecordingTime() callback.

```
public void onDotGetRecordingTime(String address, int startUTCSeconds,
int totalRecordingSeconds, int remainingRecordingSeconds) {
    // startUTCSeconds is used for normal and timed recoding, and
timestamp when recording starts in seconds
```

```
    // For timing recording
    // totalRecordingSeconds returns the total recording time
    // remainingRecordingSeconds is the remaining time of the timed
recording
}
```

4.13 Recording data export

A stand-alone application – Movella Dot Data Exporter is provided to export the recording data to PC via USB cable. You can download Windows or MacOS version in developers page.

Figure 7 shows the recommended workflow to start and stop recording with Android SDK.

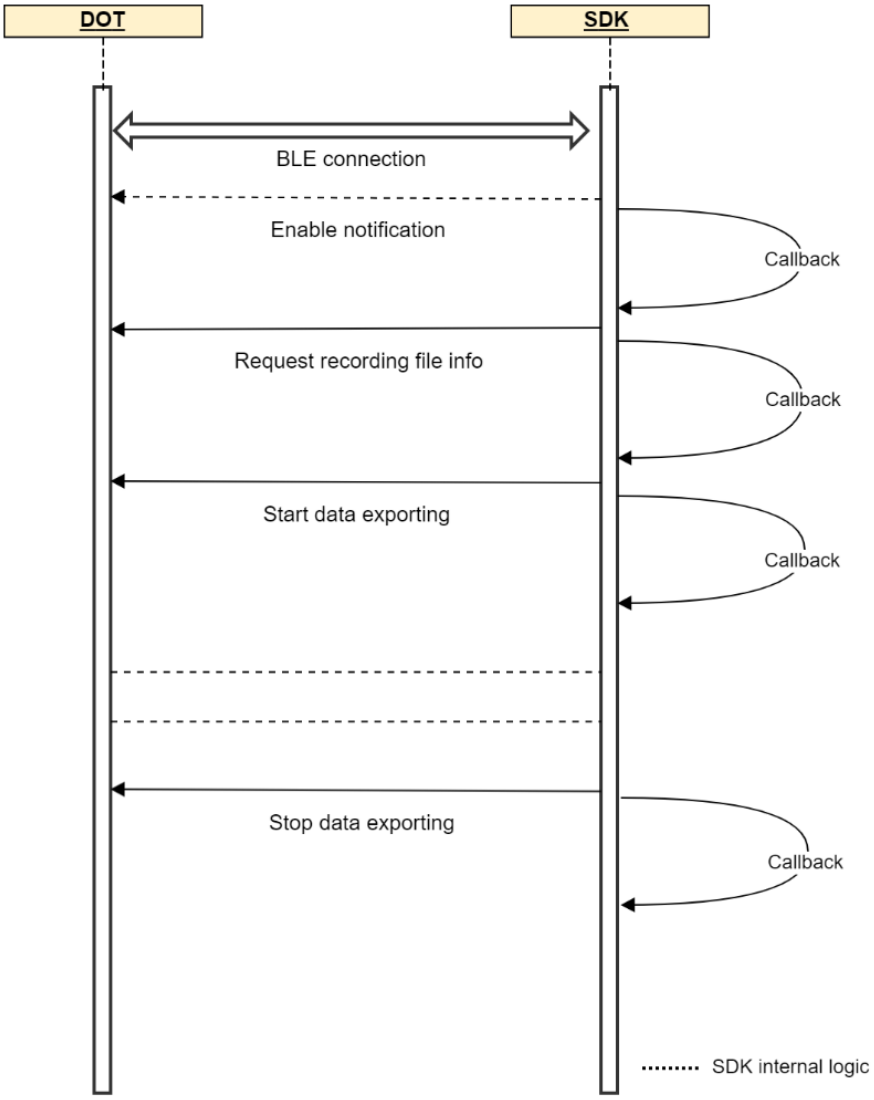

Figure 7: Workflow to export recording data

Before exporting data, ensure that Notification and mManager.requestFlashInfo() are enabled first.

4.13.1 Get export file information

After selecting the sensors to be exported, call mManager.requestFileInfo() to get the list of recording files:

```
public void onDotRequestFileInfoDone(String address,
ArrayList<DotRecordingFileInfo> list, boolean isSuccess) {
    // A list of file information can be obtained, one message
contains: fileId, fileName, dataSize
}
```

4.13.2 Set export data format

After getting the file list, select the export data quantities and call the method *mManager.selectExportedData(mSelectExportedDataIds)*. *mSelectExportedDataIds* is an array of data quantities that need to be exported. Please sort from smallest to largest. Check *DotRecordingManager.RECORDING_DATA_ID_** for detailed information. For example:

```
mSelectExportedDataIds = new byte[3];
mSelectExportedDataIds[0] =
DotRecordingManager.RECORDING_DATA_ID_TIMESTAMP;
mSelectExportedDataIds[1] =
DotRecordingManager.RECORDING_DATA_ID_EULER_ANGLES;
mSelectExportedDataIds[2] =
DotRecordingManager.RECORDING_DATA_ID_CALIBRATED_ACC;
```

NOTE:

- Free acceleration is not provided in this firmware. Refer to this BASE article to calculate free acceleration from orientation (quaternion) and acceleration.

4.13.3 Start exporting

Then select the files to be exported and call *mManager.startExporting(exportingFileList)* according to the *exportingFileList* to export.

```
public void onDotDataExported(String address, DotRecordingFileInfo
fileInfo, DotData exportedData) {
    // When the export is in progress, this callback will be called,
returning each exported data DotData, corresponding to the selected
field
    // Data can be stored through and written to the csv file
    // E.g:
    if (xsLogger == null) {
        xsLogger = DotLogger.createRecordingsLogger(ctx,
mSelectExportedDataIds, filename, tag, device.firmwareVersion,
BuildConfig.VERSION_NAME);
    }
    xsLogger.update(data);

}
```

Every time a file is exported, there will be a callback:

```
public void onDotDataExported(String address, DotRecordingFileInfo
fileInfo) {
}
```

All the files have been exported:

```
public void onDotAllDataExported(String address) {
}
```

4.13.4 Stop exporting

During the exporting, you can also stop by calling *stopExporting()*:

```
mManager.stopExporting();
```

And the callback method will be triggered:

```
public void onDotStopExportingData(String address) {
    // Determine whether all devices stop exporting
}
```

4.13.5 Check exporting status

Erasing flash, data recording and data exporting operation, only one operation can be processing at the same time. Therefore, before doing related operations, check the status of the sensor by:

```
mManager.requestRecordingState();
```

And the callback will be triggered:

```
public void onDotRecordingAck(String address, int recordingId, boolean
isSuccess, DotRecordingState recordingState){
    // Check you exporting state here
}
```

Recording states are defined in Table 9.

IMPORTANT!
One sensor corresponds to one manager. You need to clear the previous manager if this manager is on longer used or renew a new manager.

```
mManager.clear();
```

4.14 Firmware update

Continuous firmware releases from are scheduled for new features, improvements, and bug fixes. With Over-the-Air (OTA) firmware update function in Movella Dot, you can easily update the sensors to latest firmware version.

With the OTA SDK, you can do the firmware update in your own application from Movella Dot update server via OTA.

NOTE:

- Sensors can only upgrade or downgrade when **in charging status**.
- Network connection is required for OTA.

4.14.1 Import SDK

In Android SDK, OTA SDK and core SDK are two separate SDK. As an extension of the core SDK, the OTA SDK relies on the core SDK. Integrate the OTA AAR file into your project as described in section 2.5 to use OTA functions.

4.14.2 Permissions

Add the network and storage permissions in AndroidManifest.xml. Since storage permission is a runtime permission, you need to add the runtime permission at the same time.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"
/>
<uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

4.14.3 Instantiate OTA manager

Instantiate an OTA manager after a sensor is connected and initialized. The constructor is:

```
DotOtaManager(@NonNull Context context, @NonNull LifecycleOwner
lifecycleOwner, @NonNull DotDevice device, @NonNull DotOtaCallback
callback);
```

Then you need to implement the callback functions：

```
SomeClass implements DotOtaCallback {
    ...
}
```

And then instantiate an *DotOtaManager* object：

```
DotOtaManager mOtaManager = new DotOtaManager(this, this, mDotDevice,
DotOtaCallback);
```

4.14.4 Check firmware update

Once you have the OTA manager, you can use it to check firmware update or downgrade. Based on the current firmware version and release type, you can check if there is new firmware version available with *checkOtaUpdates*. This function is usually used when you just want to check if there is new firmware available.

```
mOtaManager.checkOtaUpdates();
```

After calling *checkOtaUpdates*, *onOtaUpdates* method will be triggered. If the result = YES and version is not an empty string, it means there is a firmware that can be updated. The result would be "NO" if there is no new firmware available.

Use *checkOtaUpdatesAndDownload* method if you want to download the firmware file (.mfw) after checking and start the OTA process.

```
mOtaManager.checkOtaUpdatesAndDownload();
```

After calling *checkOtaUpdatesAndDownload* method, there will be several callbacks. One is the *onOtaUpdates*, which is the same as calling *checkOtaUpdates*. Another one is *onOtaDownload*. If the available firmware has been downloaded, this method will be triggered. If the new firmware file does not match the current sensor, the *onOtaFileMismatch* will be triggered and the OTA progress will be ended.

```
SomeClass implements DotOtaCallback {
    public void onOtaUpdates(String address, boolean result, String
version, String releaseNotes) {
        ...
    }

    public void void onOtaFileMismatch(String address) {
            ...
    }

    public void onOtaDownload(String address, boolean result) {
        ...
    }
        ...

}
```

4.14.5 Check firmware downgrade

Firmware downgrade function is provided to downgrade the beta firmware versions to the last stable version. Stable firmware versions cannot downgrade to any previous versions.

Moreover, if a new stable version is available, beta versions cannot rollback to previous stable versions. You can only update the beta versions to the new stable version under this circumstance.

Similar to checking update, you can use *checkOtaRollback* method to check if the sensor can rollback.

```
mOtaManager.checkOtaRollback();
```

After calling *checkOtaRollback*, *onOtaRollback* method will be triggered. If the result = YES and version is not an empty string, it means that there is a firmware that can be rollbacked. The result would be "NO" if there is no firmware available.
Use *checkOtaRollbackAndDownload* method if you want to download the firmware file (.mfw) after checking and start the OTA process.

```
mOtaManager.checkOtaRollbackAndDownload();
```

After calling *checkOtaRollbackAndDownload* method, there will be two callbacks. One is the *onOtaRollback*, which is the same as calling *checkOtaRollback*. The other is *onOtaDownload*. If the available firmware has been downloaded, this method will be triggered:

```
SomeClass implements DotOtaCallback {
        ...
    public void onOtaRollback(String address, boolean result, String
version, String releaseNotes) {
        ...
    }

    public void onOtaDownload(String address, boolean result) {
        ...
    }
        ...
}
```

4.14.6 Start OTA

You can start the OTA process once the target firmware file has been downloaded. Start the OTA by calling *startOta* method:

```
SomeClass implements DotOtaCallback {
        ...
    public void onOtaDownload(String address, boolean result) {
        ...
        If (result) {
            mOtaManager.startOta();
    }
        ...

}
```

During the OTA process, the firmware file will be transmitted to the sensor and updated. You can get the OTA status from these callback methods:

1. *onOtaStart* – The OTA has started successfully.
2. *onOtaProgress* – The OTA is still in progress.
3. *onOtaEnd* – The OTA has ended successfully and the update or downgrade is done.

```
SomeClass implements DotOtaCallback {
        ...
```

```
    public void onOtaStart(String address, boolean result, int
errorCode)
    {
        ...
    }

public void onOtaProgress(String address, float progress, int
errorCode)
    {
        ...
    }

    public void onOtaEnd(String address, boolean result, int errorCode)
    {
        ...
    }
        ...
}
```

The OTA will fail if any of the above stages fails. There are some common reasons for OTA failure:

1. Failed to send 'start OTA' and 'stop OTA' commands.
2. OTA file is not sent completely and is always retransmitting. This is usually due to the insufficient Bluetooth performance of the mobile device.
3. Sensor is out of charging status during OTA.
4. Sensor disconnects during OTA.

The *onOtaUncharged* callback will be called if the sensor is not in charging and the OTA process will end.

```
SomeClass implements DotOtaCallback {
        ...
    public void onOtaUncharged(String address)
    {
        ...
    }
}
```

4.14.7 Stop OTA

You can stop the OTA when it is still in progress:

```
mOtaManager.stopOta();
```

After stopOta method called, this callback will be triggered:

```
SomeClass implements DotOtaCallback {
public void onOtaEnd(String address, boolean result, int errorCode)
    {
        ...
    }
}
```

4.14.8 Clear cache

The downloaded firmware files will be saved in Android app internal data storage. For now we only have OTA related cache files in the data cache folder. You can delete them by using *clearCache* method.

```
DotSdk.clearCache(applicationContext);
```

NOTE:
One sensor corresponds to one manager. You need to clear the previous manager if this manager is no longer used or renew a new manager.

```
mOtaManager.clear();
```

4.15 MFM

When Movella DOT sensor is mounted to an object that contains ferromagnetic materials, the measured magnetic field can become distorted, causing errors in measured orientation. To correct for known magnetic disturbances, Magnetic Field Mapper function has been developed to allow users to remap the magnetic field of the sensor.

The MFM can be executed in a few minutes and yields a new set of calibration values that can be written to the Movella DOT's non-volatile memory, which means it will not be erased by powering off or firmware updates.

Refer to this BASE article for more information.

With the SDK, you can start/stop MFM, start data processing, get mtb output data and write it to sensor.

Figure 6 shows the recommended workflow to start and stop MFM with Android SDK.

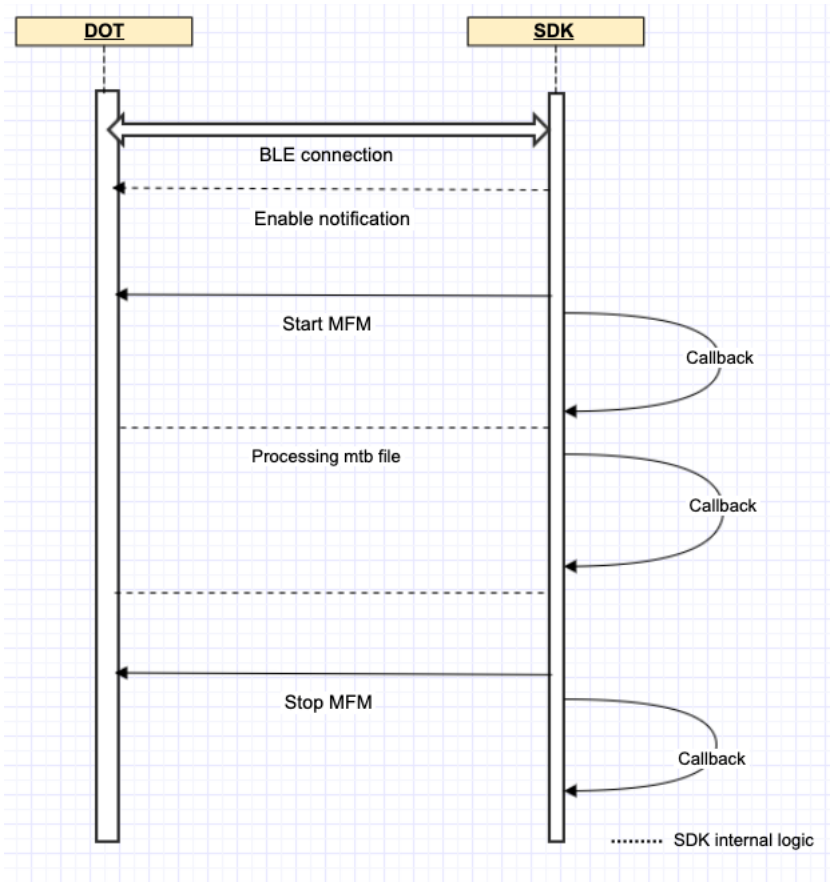

Figure 8: Workflow to start and stop MFM

To perform the MFM function, an *DotMfmManager* needs to be instantiated. In most cases, one *DotDevice* uses one *DotMfmManager*.

First you need to implement the *DotMfmCallback* interface:

```
public class EasyMfmFragment implements DotMfmCallback {

    public void onDotMfmProgressChanged (String address, int percentage)
{

    }

    public void onDotMfmCompleted (String address, int result, byte[]
mtbData) {

    }
}
```

Then instantiate an *DotMfmManager*, for example:

```
private DotMfmManager mMfmManager;
mMfmManager = DotMfmManager(context, DotDevice, EasyMfmFragment.this);
```

4.15.1 Start/stop MFM

You can call *mMfmManager.startMfm()* to start MFM, it starts to collect real-time streaming data in SDK.
Call *mMfmManager.stopMfm(false)* method to stop MFM, it stops to collect real-time streaming data:

- Parameter *isWaitingForResult*: Indicates whether to wait for processing results. Normally, it is set to false.

After startMfm method called, this callback method will be triggered, you can update the percentage in UI view:

```
public void onDotMfmProgressChanged (String address, int percentage) {
    ...
    // You can update the percentage in UI view
    mAdapter.updatePercentage(address, percentage);
    ...
}
```

4.15.2 Start processing

Instantiate an *DotMfmProcessor* for processing data, for example:

```
private DotMfmProcessor mMfmProcessor;
mMfmProcessor = DotMfmProcessor(context);
mMfmProcessor.clear(); // If you want to reuse this, please clear it.
```

After the percentage of data collection reaches to 100, it means that the collected data can be used to process MFM, you can stop MFM and call mMfmProcessor.addMtbFile(address, path) and *mMfmProcessor.startProcess()* to process data. It will be processed for a period of time, depending on the performance of the phone.

```
public void onDotMfmProgressChanged (String address, int percentage) {
    ...
    if (percentage == 100) {
```

```
        activity?.runOnUiThread {

            // Stop MFM first
            mMfmManager.stopMfm(false);

            // Need to run on UI thread.
            // Processing collected data.
            String path = mMfmManager.getMtbFilePath();

            if (path.isNotEmpty()) {
                mMfmProcessor.addMtbFile(address, path);
                mMfmProcessor.startProcess();
            }
        }
    }
    ...
}
```

4.15.3 Completed MFM

After data processing is done, this callback method will be triggered, If the result is *DotMfmResult.ACCEPTABLE* or *DotMfmResult.GOOD*, you can decide whether to write to the sensor by calling *mMfmManager.writeMfmResultToDevice(mtbData)*:

```
    public void onDotMfmCompleted (String address, int result, byte[]
mtbData) {
    ...
    if (result == DotMfmResult.ACCEPTABLE
       || result == DotMfmResult. GOOD) {
        boolean isSuccess = mMfmManager.writeMfmResultToDevice(mtbData);
        // check write status, re-write, or update the UI
        if (isSuccess) {
            ...
        } else {
            ...
        }
    }
    ...
}
```

NOTE:
One sensor corresponds to one manager. You need to clear the previous manager if this manager is no longer used or renew a new manager.

```
mMfmManager.clear();
```

4.16 Factory Reset

4.16.1 Initialization

After a device is connected via Bluetooth, the device can be reset to factory defaults. To initiate factory reset, first implement 'SettingsCallback' and override its methods.

```
class FactoryResetActivity : AppCompatActivity(), SettingsCallback {
        ...

}
```

Create an instance of `DotSettingsManager`

```
var dotSettingsManager =
DotSettingsManager.getInstance(this@FactoryResetActivity)
```

4.16.2 Check for feature support

Check whether the device supports being restored to factory defaults. This check is performed while resetting the device as well.

```
var doesSupport = device.doesSupportFactoryReset()
```

4.16.3 Factory Reset

After a device is connected via Bluetooth, the device can be reset to factory defaults. To initiate factory reset, first implement 'SettingsCallback'

```
class FactoryResetActivity : AppCompatActivity(), SettingsCallback,
DotDeviceCallback {
        ...

}
```

Initiate factory reset:

```
val deviceLIst: ArrayList<DotDevice> = ArrayList()
//Add DotDevices to list
deviceList.add(device1)
deviceList.add(device2)
deviceList.add(device3)
. . .

//Restore to factory settings
dotSettingsManager.restoreFactoryDefaults(deviceLIst)
```

4.16.4 Get Result

Override the 'onFactoryResetResult(String deviceAddress, bool isReset)' method:

```
override fun onFactoryResetResult(deviceAddress: String?,result:
DotResetResults) {
Log.d("AFTER RESET","Device : $deviceAddress | isReset : ${
result.name}")
    }
```

4.17 Other functions

4.17.1 Read RSSI

While scanning sensors, you can get RSSI from scanner callback:

```
SomeClass implements DotScannerCallback {
    public void onDotScanned(BluetoothDevice device, int rssi) {
        ...
    }
}
```

You can also read RSSI when sensor is connected:

```
DotDevice.readRssi();
```

It will trigger the callback:

```
SomeClass implements DotDeviceCallback {
    public void onReadRemoteRssi(String address, int rssi) {
        ...
    }
}
```

4.17.2 Identify

To identify or find your device, you can call the following function. The device will fast blink 8 times and then a short pause when you call this function.

```
xsDevice.identifyDevice();
```

4.17.3 Power saving

In power-saving mode, sensors will turn off the signal pipeline and BLE connection, put the MCU in a sleep state to ensure minimum power consumption. The default time threshold to enter power saving mode is set to 10 min in advertisement mode and 30 min in connection mode. These values are saved in the non-volatile memory and can be adjusted in Movella Dot app or SDK.

There is an example to set power saving time in advertisement and connection mode both to 30 minutes.

```
DotDevice.setPowerSaveTimeout(timeoutXMinutes, timeoutXSeconds,
timeoutYMinutes, timeoutYSeconds);
```

4.17.4 Button callback

If there is a single click on the power button during connection, a notification will be sent with a timestamp when this single click is released. This function is called as "Button callback".

When the pressing time is 10~800ms, it is judged as a valid single click. The timestamp is from sensor's local clock and independent of synchronization.

```
SomeClass implements DotDeviceCallback {

    public void onDotButtonClicked(String address, long timestamp) {
        ...
    }
}
```

4.17.5 Power on options

This feature is to allow user to configure the Movella Dot v2 sensor to be powered on by USB plugin or not. This setting is only available in v2 sensor.

By default, power on by USB is disabled. So, the sensor will be in charging status if connected with USB cable. You can call *enableUsbPowerOn()* to set enable this feature.

```
xsDevice.enableUsbPowerOn(boolean isEnabled);
```

By enabling USB power on, the sensor will power on immediately after the USB plugin. With this feature, you can power on multiple sensors with the USB plugin at once.

5 Appendix

5.1 Real-time streaming modes

NOTE:
You can get other data quantities from the available data in each measurement mode. Refer to section 4.10.6**Error! Reference source not found.** for the conversions that can be used.

5.1.1 Extended (Quaternion)

Table 10: Extended (Quaternion)

Mode name	Payload	Available data
PAYLOAD_TYPE_EXTENDED_QUATERNION	36 bytes	• SampleTimeFine • Orientation (Quaternions) • Free acceleration • Status

5.1.2 Complete (Quaternion)

Table 11: Complete (Quaternion)

Mode name	Payload	Available data
PAYLOAD_TYPE_COMPLETE_QUATERNION	32 bytes	• SampleTimeFine • Orientation (Quaternions) • Free acceleration

5.1.3 Orientation (Quaternion)

Table 12: Orientation (Quaternion)

Mode name	Payload	Available data
PAYLOAD_TYPE_ORIENTATION_QUATERNION	20 bytes	• SampleTimeFine • Orientation (Quaternions)

5.1.4 Extended (Euler)

Table 13: Extended (Euler)

Mode name	Payload	Available data
PAYLOAD_TYPE_EXTENDED_EULER	32 bytes	• SampleTimeFine • Orientation (Euler Angles) • Free acceleration • Status

5.1.5 Complete (Euler)

Table 14: Complete (Euler)

Mode name	Payload	Available data
PAYLOAD_TYPE_COMPLETE_EULER	28 bytes	• SampleTimeFine • Orientation (Euler Angles) • Free acceleration

5.1.6 Orientation (Euler)

Table 15: Orientation (Euler)

Mode name	Payload	Available data
PAYLOAD_TYPE_ORIENTATION_EULER	16 bytes	• SampleTimeFine • Orientation (Euler Angles)

5.1.7 Free acceleration

Table 16: Free acceleration

Mode name	Payload	Available data
PAYLOAD_TYPE_FREE_ACCELERATION	16 bytes	• SampleTimeFine • Free acceleration

5.1.8 High fidelity (with mag)

Table 17: High fidelity (with mag)

Mode name	Payload	Available data
PAYLOAD_TYPE_HIGH_FIDELITY_WITH_MAG	35 bytes	• SampleTimeFine • dq • dv • Angular velocity • Acceleration • Magnetic field • Status

5.1.9 High fidelity

Table 18: High fidelity

Mode name	Payload	Available data
PAYLOAD_TYPE_HIGH_FIDELITY_NO_MAG	29 bytes	• SampleTimeFine • dq • dv • Angular velocity • Acceleration • Status

5.1.10 Delta quantities (with mag)

Table 19: Delta quantities (with mag)

Mode name	Payload	Available data
PAYLOAD_TYPE_DELTA_QUANTITIES_WITH_MAG	38 bytes	• SampleTimeFine • dq • dv • Magnetic field

5.1.11 Delta quantities

Table 20: Delta quantities

Mode name	Payload	Available data
PAYLOAD_TYPE_DELTA_QUANTITIES_NO_MAG	32 bytes	• SampleTimeFine • dq • dv

5.1.12 Rate quantities (with mag)

Table 21: Rate quantities (with mag)

Mode name	Payload	Available data
PAYLOAD_TYPE_RATE_QUANTITIES_WITH_MAG	34 bytes	• SampleTimeFine • Angular velocity • Acceleration • Magnetic Field

5.1.13 Rate quantities

Table 22: Rate quantities

Mode name	Payload	Available data
PAYLOAD_TYPE_RATE_QUANTITIES_NO_MAG	28 bytes	• SampleTimeFine • Angular velocity • Acceleration

5.1.14 Custom mode 1

Table 23: Custom mode 1

Mode name	Payload	Available data
PAYLOAD_TYPE_CUSTOM_MODE_1	40 bytes	• SampleTimeFine • Orientation (Euler Angles) • Free acceleration • Angular velocity

5.1.15 Custom mode 2

Table 24: Custom mode 2

Mode name	Payload	Available data
PAYLOAD_TYPE_CUSTOM_MODE_2	34 bytes	• SampleTimeFine • Orientation (Euler Angles) • Free acceleration • Magnetic field

5.1.16 Custom mode 3

Table 25: Custom mode 3

Mode name	Payload	Available data
PAYLOAD_TYPE_CUSTOM_MODE_3	32 bytes	• SampleTimeFine • Orientation (Quaternions) • Angular velocity

5.1.17 Custom mode 4

Table 26: Custom mode 4

Mode name	Payload	Available data
PAYLOAD_TYPE_CUSTOM_MODE_4	51	• SampleTimeFine • Orientation (Quaternions) • dq • dv • Angular velocity • Acceleration • Magnetic field • Status

5.1.18 Custom mode 5

Table 27: Custom mode 5

Mode name	Payload	Available data

PAYLOAD_TYPE_CUSTOM_MODE_5	44 bytes	• SampleTimeFine • Orientation (Quaternions) • Acceleration • Angular velocity