



Document XD0201P, Revision D, 15 Dec 2020

# Xsens DOT SDK Programming Guide for Android

| Revision | Date | By | Changes |
|---|---|---|---|
| A | 7 Jan 2020 | XUF, ABO | Initial release |
| B | 25 April 2020 | XUF | Update minimum support to Android OS 8.0<br>Update SDK architecture diagram<br>Added supported platforms<br>Update new measurement modes<br>Add SDK changelogs links<br>Add data conversion example code<br>Delete OTA and MFM section |
| C | 27 August 2020 | XUF, CHW | Add sync class and interfaces<br>Add recording class and interfaces<br>Add sync example code<br>Add recording example code<br>Add 4 new payload modes<br>Change the name of interfaces from "…Cb" to "…Callback" |
| D | 15 Dec 2020 | XUF, CHW, AJI | Add an interface to check new firmware<br>Add the workflow to start synchronization<br>Add the workflow to start and stop real-time streaming<br>Add the workflow for heading reset<br>Add the workflow to start and stop recording<br>Add the workflow to export recording data<br>Add the new output rates for real-time streaming and recording<br>Add new filter profile<br>Add button callback function<br>Support RSSI<br>Add get sync status function<br>Add stop sync function |

# Table of Contents

# List of Tables



# List of Figures

# 1 Introduction

The Xsens 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 *Xsens DOT SDK core documentation* with detailed information. Before getting started with the SDK, it's advised to read *Xsens DOT User Manual* first to understand the basic functions of the sensor.

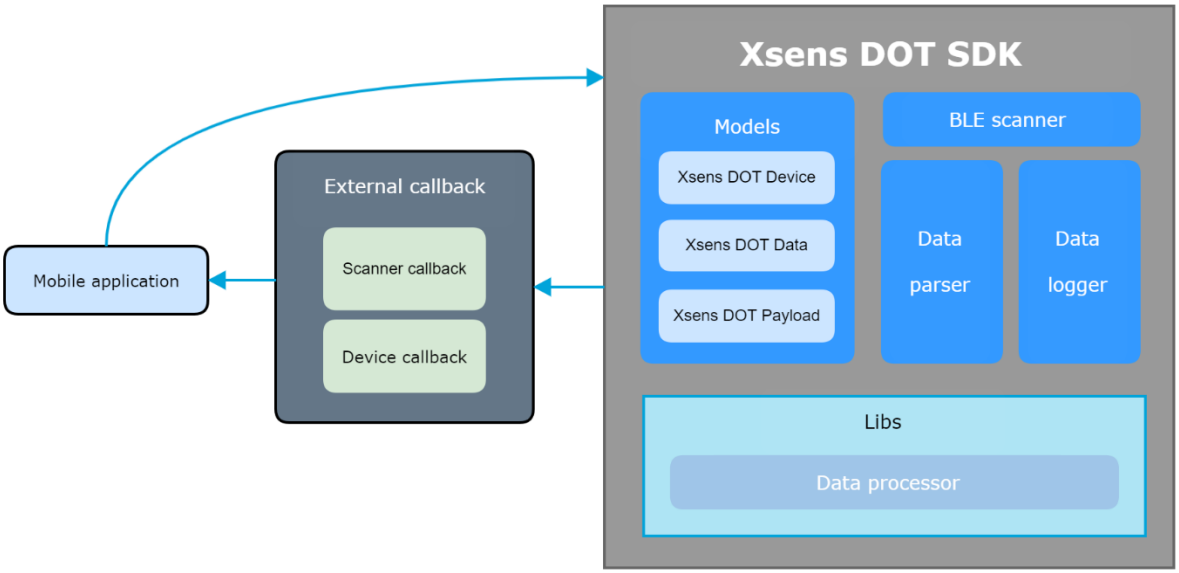


Figure 1: Xsens DOT Mobile SDK Architecture

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 Xsens 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<br>• ARMv8 CPU architecture<br>• Bluetooth<br>  ○ Best performance with BLE 5.0, DLE[2] supported<br>  ○ Compatible with Bluetooth 4.2 |

## 2.2 Example code

Refer to this project on GitHub for the Android example code of Xsens DOT SDK:

- https://github.com/xsens/xsens_dot_example_android

## 2.3 SDK Changelogs

https://base.xsens.com/hc/en-us/articles/360013337940-Xsens-DOT-Release-Notes-and-Change-Logs-

## 2.4 Prerequisites for Android Studio Project

This section addresses setup parameters for proper usage of the Xsens 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:2.4.0"
4. Dependency lifecycle: implementation "androidx.lifecycle:lifecycle-runtime:2.2.0"

## 2.5 Import SDK Package

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

1. Open your Android Studio project and select **`File /New/New Module`**…, select **`Import .JAR/.AAR Package`**.

[2] Data Length Extension

2. Select the AAR file of the XsensDotSdk, click **Finish**.
3. Select **File/Project Structure…/Dependencies**, choose a main module (it's **app** in normal case) and click the **Add Dependency/Module Dependency**.
4. Select **XsensDotSdk** module then press **OK** to close this dialog.
5. After **Build finished**, you can do some basic settings of the SDK as shown below.

```
private void initXsSdk() {

    String version = XsensDotSdk.getSdkVersion();

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

## 2.6 Implement Interface

Developers can implement *XsensDotDeviceCallback* and *XsensDotScannerCallback* in one activity as shown below.

```
public class MainActivity extends AppCompatActivity
                    implements XsensDotDeviceCallback,
XsensDotScannerCallback {
  ...
  ...
  ...
}
```

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 needs to be implemented automatically.

## 2.7 Classes

The list of classes as part of Xsens DOT SDK is shown in Table 2.

Table 2: Classes in Xsens DOT SDK

| Class | Description |
|---|---|
| XsensDotSdk | The SDK main object used for global settings such as enable debug or reconnect features. |
| XsensDotDevice | Represents a Xsens DOT device object, including basic information and operations. Return data by XsensDotDeviceCallback. |
| XsensDotData | Contains all the measurement data, including acceleration, angular velocity, and mag data, etc. |
| XsensDotLogger | A class for data logging. |
| XsensDotParser | A class for parsing data from the device via Bluetooth. |
| XsensDotScanner | A class for scanning Xsens DOT device. Return the scanned device by XsensDotScannerCallback. |
| XsPayload | For setting different payload types for measurement. |

| | |
|---|---|
| XsensDotRecordingManager | Data recording manager, including data recording, and exporting methods. |
| XsensDotSyncManager | Synchronization manager for sensors' syncing |

## 2.8 Interfaces

The list of available interfaces as part of Xsens DOT SDK is shown in Table 3.

Table 3: Interfaces in Xsens DOT SDK

| Class | Description |
|---|---|
| XsensDotDeviceCallback | An interface for notifying device information, measurement data. |
| XsensDotScannerCallback | An interface for notifying LE scan result |
| XsensDotMeasurementCallback | An interface for notifying measurement status and data. |
| XsensDotCiCallback | An interface for notifying firmware crash information result. |
| XsensDotRecordingCallback | An interface for notifying recording status and data of device. |
| XsensDotSyncCallback | An interface for synchronization result. |
| XsensDotOtaSimpleCallback | An interface for checking new firmware version. |

## 2.9 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 |

# 3 SDK Usage with Examples

## *3.1 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.

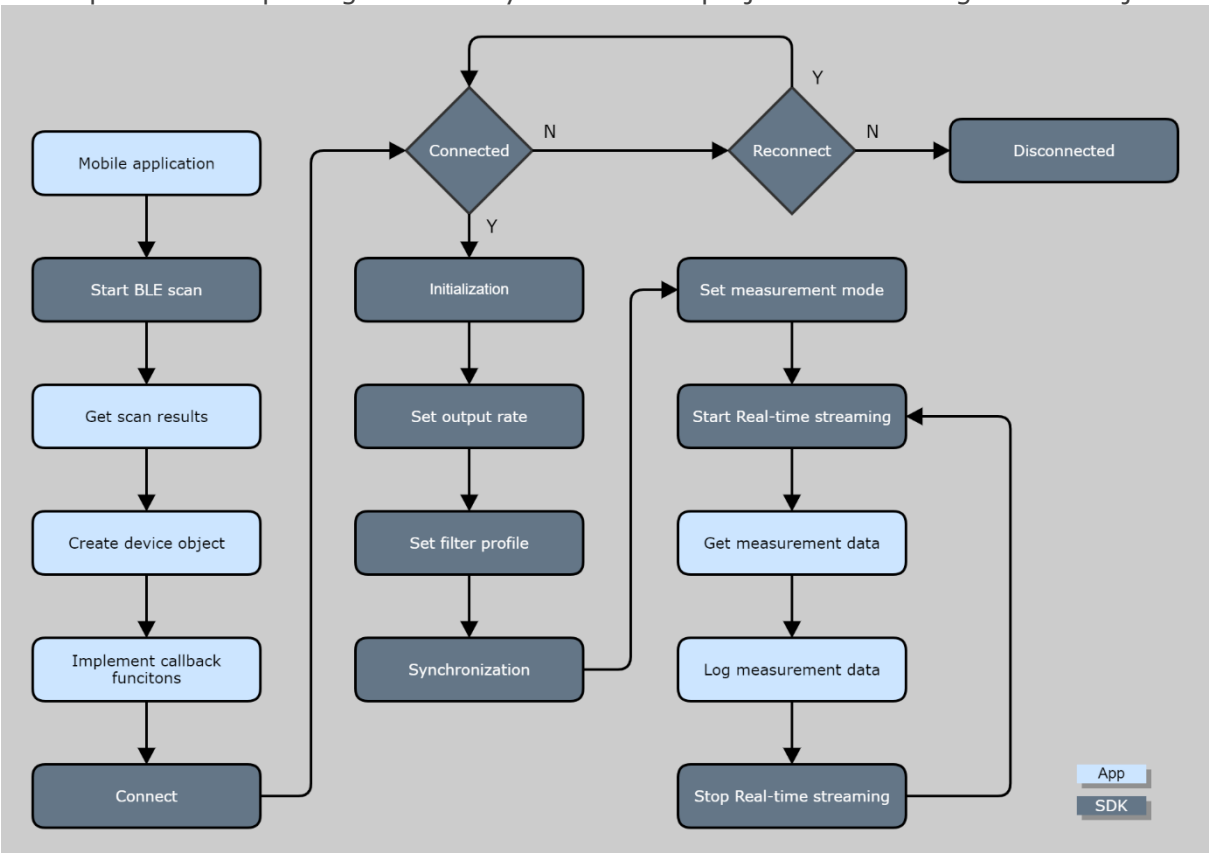


Figure 2: Xsens DOT Android SDK Workflow

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 *XsensDotDevice* class. Most of the operations can be done by making use of this class.

Developers can call the connect function in *XsensDotDevice* 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.

## 3.2 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 – XsensDotSdk.

```
XsensDotSdk.setDebugEnabled(true);
```

This setting is disabled by default.

## 3.3 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.

```
XsensDotSdk.setReconnectEnabled(true);
```

You can cancel the reconnecting by:

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

## 3.4 BLE scan

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

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

```
private XsensDotScanner mXsScanner;

private void initXsScanner() {

    mXsScanner = new XsensDotScanner(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 *onXsensDotScanned* callback function. Note that only Xsens DOT device is reported.

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

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

## 3.5 Connect

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

```
XsensDotDevice xsDevice =
new XsensDotDevice(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 *onXsensDotConnectionChanged* callback function will be triggered. If the state equals to *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 *onXsensDotServicesDiscovered* callback function.

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

    if (state == XsensDotDevice.CONN_STATE_DISCONNECTED) {

        // Update UI
    }
}

@Override
public void onXsensDotServicesDiscovered(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

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

- getLogState
- getTag
- getCurrentOutputRate
- getFilterProfileInfoList
- isSynced
-
- …

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

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

## 3.6 Connect multiple sensors

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

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

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

```
XsensDotDevice xsDevice = new XsensDotDevice(
                                    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 *XsensDotDevice* will be removed from the list after the device is disconnected. In a similar way, put *XsensDotLogger* object into a list to manage data collecting and logging for multiple devices.

## 3.7 Initialization

After the sensor connection, an initialization process is introduced to enable BLE notifications and obtain basic information, such as firmware version, tag name, synchronization status, etc. *onXsensDotInitDone()* is the callback after the initialization is successful.

NOTES:

- Any read or write operation should be called after a successful.

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

## *3.8 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.

### 3.8.1 Get current filter profile

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

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

You can also get the current filter profile from callback:

```
SomeClass implements XsensDotDeviceCallback {

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

### 3.8.2 Get all the filter profiles

Get all the supported filter profiles through *getFilterProfileInfoList*:

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

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

```
SomeClass implements XsensDotDeviceCallback {

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

### 3.8.3 Set a new filter profile

Before set 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.

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

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

## 3.9 Output rate

After the initialization is done, you can get or set the output rate for the measurement by *XsensDotDevice* class. Table 5 shows the available output rates during measurements.

Table 5: 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 = XsensDotDevice.getCurrentOutputRate();
```

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

```
SomeClass implements XsensDotDeviceCallback {

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

Set a new output rate for the measurement:

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

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

## 3.10 Synchronization

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

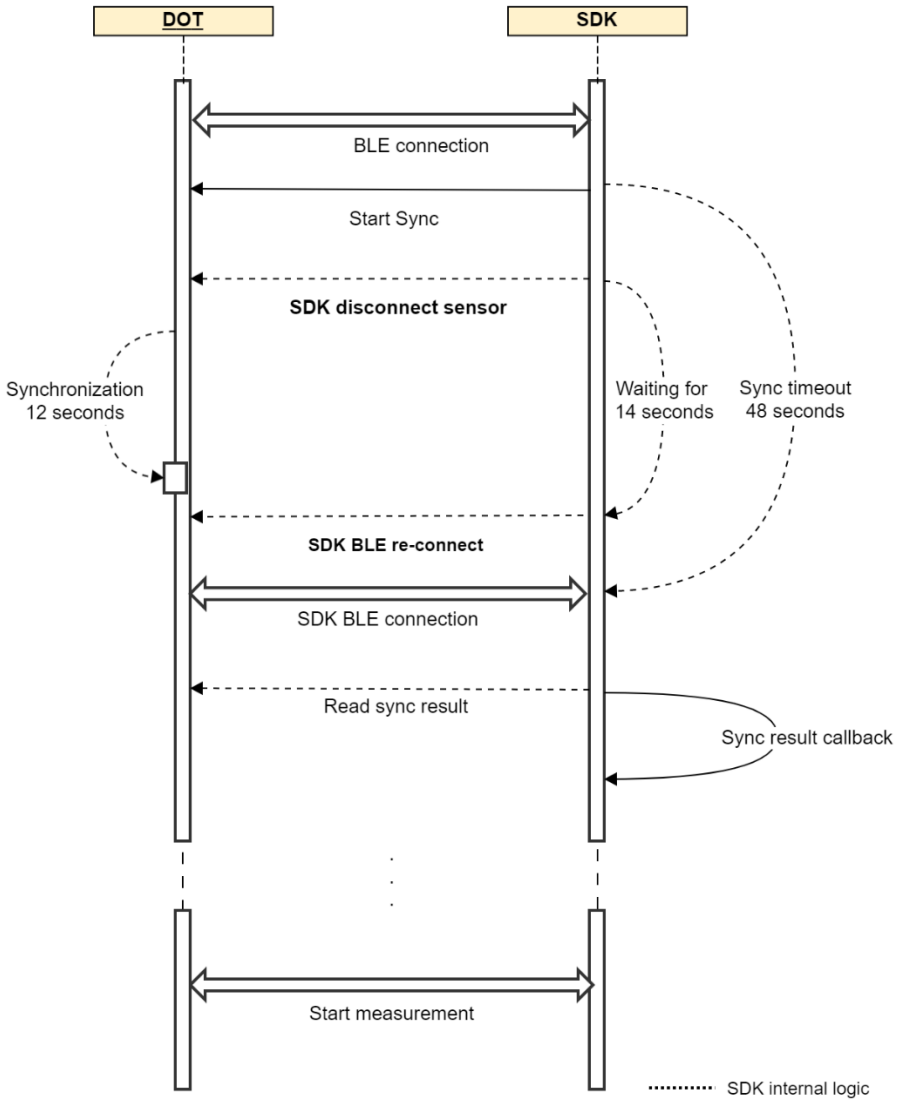


Figure 3: Workflow to start synchronization with SDK

First implement the *XsensDotSyncCallback* interface.

```
public class RecordingFragment implements XsensDotSyncCallback {

    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) {
    }
}
```

### 3.10.1 Get sync status

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

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

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

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

### 3.10.2 Start sync

One of the sensors must be set as the root sensor before starting synchronization:

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

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

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

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

NOTES:

- Do not interrupt during the synchronization process.

### 3.10.3 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.

If synchronization fails on any sensor, it's advised to stop the synchronization for the successful sensors and re-do the synchronization again.

Start real-time streaming and recording after all the sensors are synchronized.

### 3.10.4 Stop sync

You can stop the synchronization for all the synced sensors:

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

Or stop the synchronization for specific sensors.

```
XsensDotSyncManager.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) {
        ...
    }
```

## *3.11 Real-time streaming*

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

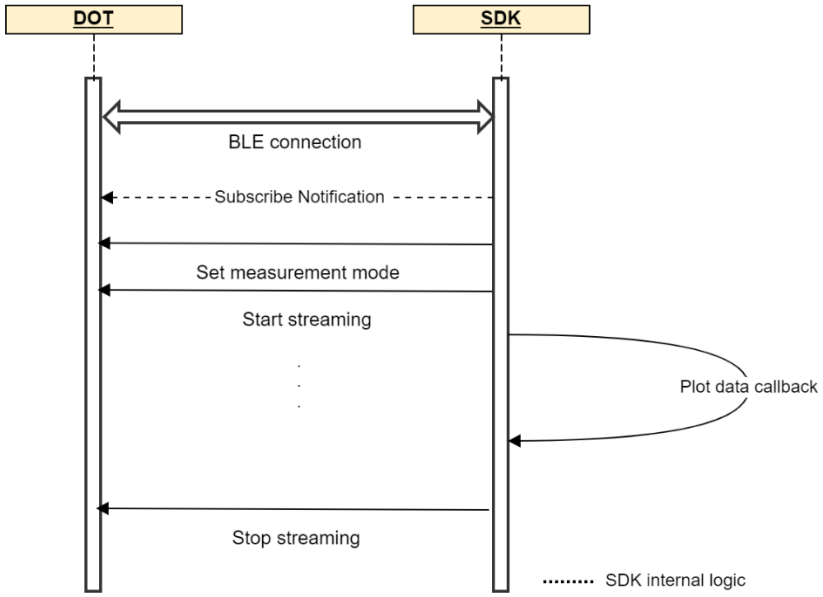


Figure 4: Workflow to start and stop real-time streaming

The *XsensDotDevice* can report sensor data via *onXsensDotDataChanged* callback function.

To use this, you need to notify the sensor to enter the measuring mode first. There are 16 measurement modes with different payload modes. Refer to Appendix for data outputs of different modes. Section 4.2 in *Xsens DOT User Manual* also gives detailed explanation about output values.

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

Then call this function to start measuring.

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

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

```
@Override
public void onXsensDotDataChanged(String address,
                                  XsensDotData xsensDotData) {
}
```

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

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

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

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

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

### 3.11.1 Data logging

The *XsensDotLogger* 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.

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

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

```
public void update(XsensDotData 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();
```

### 3.11.2 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 2 high fidelity modes in Android SDK to provide inertial data:

- PAYLOAD_TYPE_HIGH_FIDELITY_WITH_MAG
- PAYLOAD_TYPE_HIGH_FIDELITY_NO_MAG

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

```
XsensDotDevice xsensDotDevice = ...;
//set measurement mode
xsensDotDevice.setMeasurementMode(XsPayload.PAYLOAD_TYPE_HIGH_FIDELITY
_WITH_MAG);

//start measurement
xsensDotDevice.startMeasuring();

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

### 3.11.3 Data conversions

Data conversion functions are provided in Xsens 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 onXsensDotDataChanged 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:

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

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

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

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

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

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

### Convert quaternion to Euler angles

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

```
final float[] quats = xsensDotData.getQuat();
final double[] eulerAngles = XsensDotParser.quaternion2Euler(quats);
```

## 3.12 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.

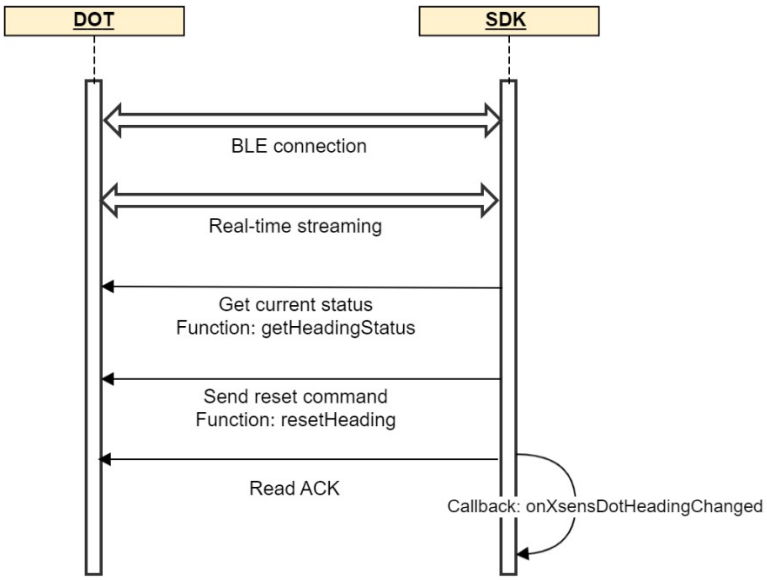


Figure 5: Workflow for heading reset

Refer to Table 6 for the heading reset status of the sensor.

Table 6: 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, then use following functions to perform heading reset.

```
XsensDotDevice xsensDotDevice = ...;
xsensDotDevice.setXsensDotMeasurementCallback(this);
//set to one sensor funsion mode
//start measurement
```

```
xsensDotDevice.resetHeading();
```

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

```
public class MeasurementFragment implements XsensDotMeasurementCallback {

      ...

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

      }

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

      }
}
```

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

Then revert heading to original value by calling this function.

```
xsensDotDevice.revertHeading();
```

## 3.13 Recording

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

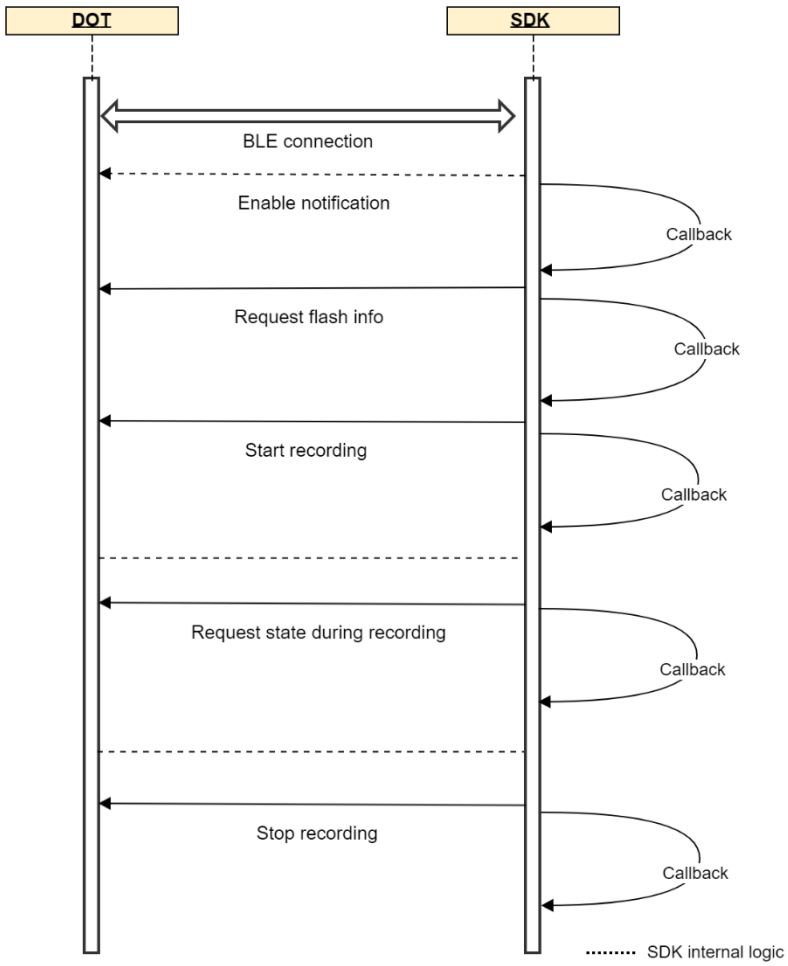


Figure 6: Workflow to start and stop recording

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

First you need to implement the *XsensDotRecordingCallback* interface:

```
public class RecordingFragment implements XsensDotRecordingCallback {

    public void onXsensDotRecordingNotification(String address, boolean
isEnabled) {

    }

    public void onXsensDotEraseDone(String address, boolean isSuccess) {

    }
```

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

    }

    public void onXsensDotRecordingAck(String address, int recordingId,
boolean isSuccess, XsensDotRecordingState recordingState) {

    }

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

    }

    public void onXsensDotRequestFileInfoDone(String address,
ArrayList<XsensDotRecordingFileInfo> list, boolean isSuccess) {

    }

    public void onXsensDotDataExported(String address,
XsensDotRecordingFileInfo fileInfo, XsensDotData exportedData) {

    }

    public void onXsensDotDataExported(String address,
XsensDotRecordingFileInfo fileInfo) {

    }

    public void onXsensDotAllDataExported(String address) {

    }

    public void onXsensDotStopExportingData(String address) {

    }

}
```

Then instantiate an *XsensDotRecordingManager*, for example:

```
private XsensDotRecordingManager mManager;
mManager = XsensDotRecordingManager(context, XsensDotDevice,
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.

### 3.13.1 Get flash information

We have a total of 16 MB flash for recording. 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 onXsensDotRecordingNotification(String address, boolean
isEnabled) {

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

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

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

### 3.13.2 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 onXsensDotRecordingAck(String address, int recordingId,
boolean isSuccess, XsensDotRecordingState recordingState) {

    if (recordingId ==
XsensDotRecordingManager.RECORDING_ID_START_RECORDING) {
        // start recording result, check recordingState, it should be
success or fail.
```

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

### 3.13.3 Get recording status

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

```
public void onXsensDotRecordingAck(String address, int recordingId,
boolean isSuccess, XsensDotRecordingState recordingState) {

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

Table 7: 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 |

### 3.13.4 Get recording time

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

```
public void onXsensDotGetRecordingTime(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
}
```

## 3.14 Recording data export

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

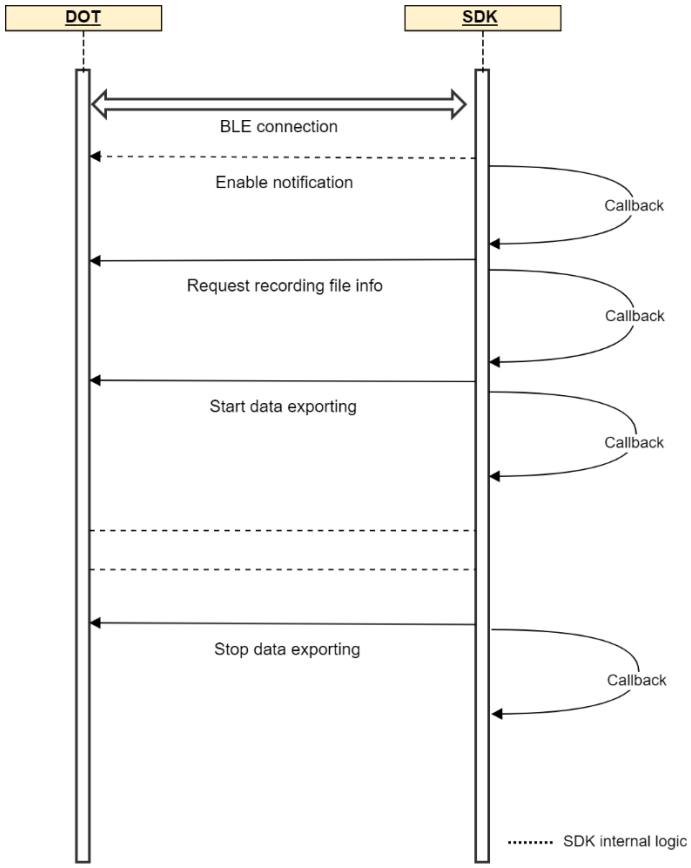


Figure 7: Workflow to export recording data

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

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

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

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 *XsensDotRecordingManager.RECORDING_DATA_ID_** for detailed information. For example:

```
mSelectExportedDataIds = new byte[3];
```

```
mSelectExportedDataIds[0] =
XsensDotRecordingManager.RECORDING_DATA_ID_TIMESTAMP;
mSelectExportedDataIds[1] =
XsensDotRecordingManager.RECORDING_DATA_ID_EULER_ANGLES;
mSelectExportedDataIds[2] =
XsensDotRecordingManager.RECORDING_DATA_ID_CALIBRATED_ACC;
```

NOTES:

- Free acceleration is not provided in this firmware. Refer to this base article to calculate free acceleration from quaternion and dv.

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

```
public void onXsensDotDataExported(String address,
XsensDotRecordingFileInfo fileInfo, XsensDotData exportedData) {
    // When the export is in progress, this callback will be called,
returning each exported data XsensDotData, corresponding to the
selected field
    // Data can be stored through and written to the csv file
    // E.g:
    if (xsLogger == null) {
        xsLogger = XsensDotLogger.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 onXsensDotDataExported(String address,
XsensDotRecordingFileInfo fileInfo) {
}
```

All the files have been exported:

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

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

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

**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();
```

## *3.15 Other functions*

### 3.15.1 Read RSSI

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

```
SomeClass implements XsensDotScannerCallback {
    public void onXsensDotScanned(BluetoothDevice device, int rssi) {
        ...
    }
}
```

You can also read RSSI when sensor is connected:

```
XsensDotDevice.readRssi();
```

It will trigger the callback:

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

### 3.15.2 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();
```

### 3.15.3 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 Xsens DOT app or SDK.

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

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

### 3.15.4 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 XsensDotDeviceCallback {

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

### 3.15.5 Firmware update notification

We can check if there is a new firmware version. Please make sure that network permission is set.

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


XsensDotOtaSimpleManager.checkOtaUpdates(this, this, mXsDevice,
    new XsensDotOtaSimpleCallback() {
    @Override
    public void onNewFirmwareVersion(String address, boolean has, int
requestCode) {
        ...
    }
}, 1);
```

# 4 Appendix

## *4.1 Real-time streaming modes*

### 4.1.1 Extended (Quaternion)

Table 8: Extended (Quaternion)

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_EXTENDED_QUATERNION | 36 bytes | • SampleTimeFine<br>• Quaternion<br>• Free acceleration<br>• Status |

### 4.1.2 Complete (Quaternion)

Table 9: Complete (Quaternion)

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_COMPLETE_QUATERNION | 32 bytes | • SampleTimeFine<br>• Quaternion<br>• Free acceleration |

### 4.1.3 Orientation (Quaternion)

Table 10: Orientation (Quaternion)

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_ORIENTATION_QUATERNION | 20 bytes | • SampleTimeFine<br>• Quaternion |

### 4.1.4 Extended (Euler)

Table 11: Extended (Euler)

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_EXTENDED_EULER | 32 bytes | • SampleTimeFine<br>• Euler<br>• Free acceleration<br>• Status |

### 4.1.5 Complete (Euler)

Table 12: Complete (Euler)

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_COMPLETE_EULER | 28 bytes | • SampleTimeFine<br>• Euler<br>• Free acceleration |

### 4.1.6 Orientation (Euler)

Table 13: Orientation (Euler)

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_ORIENTATION_EULER | 16 bytes | • SampleTimeFine<br>• Euler |

### 4.1.7 Free acceleration

Table 14: Free acceleration

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_FREE_ACCELERATION | 16 bytes | • SampleTimeFine<br>• Free acceleration |

### 4.1.8 High fidelity (with mag)

Table 15: High fidelity (with mag)

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_HIGH_FIDELITY_WITH_MAG | 35 bytes | • SampleTimeFine<br>• dq<br>• dv<br>• Angular velocity<br>• Acceleration<br>• Magnetic field<br>• Status |

### 4.1.9 High fidelity

Table 16: High fidelity

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_HIGH_FIDELITY_NO_MAG | 29 bytes | • SampleTimeFine<br>• dq<br>• dv<br>• Angular velocity<br>• Acceleration<br>• Status |

### 4.1.10 Delta quantities (with mag)

Table 17: Delta quantities (with mag)

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_DELTA_QUANTITIES_WITH_MAG | 38 bytes | • SampleTimeFine<br>• dq<br>• dv<br>• Magnetic field |

### 4.1.11 Delta quantities

Table 18: Delta quantities

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_DELTA_QUANTITIES_NO_MAG | 32 bytes | • SampleTimeFine<br>• dq<br>• dv |

### 4.1.12 Rate quantities (with mag)

Table 19: Rate quantities (with mag)

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_RATE_QUANTITIES_WITH_MAG | 34 bytes | • SampleTimeFine<br>• Angular velocity<br>• Acceleration |

### 4.1.13 Rate quantities

Table 20: Rate quantities

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_RATE_QUANTITIES_NO_MAG | 28 bytes | • SampleTimeFine<br>• Angular velocity<br>• Acceleration |

### 4.1.14 Custom mode 1

Table 21: Custom mode 1

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_CUSTOM_MODE_1 | 40 bytes | • SampleTimeFine<br>• Euler angle<br>• Free acceleration<br>• Angular velocity |

### 4.1.15 Custom mode 2

Table 22: Custom mode 2

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_CUSTOM_MODE_2 | 34 bytes | • SampleTimeFine<br>• Euler angle<br>• Free acceleration<br>• Magnetic field |

### 4.1.16 Custom mode 3

Table 23: Custom mode 3

| Mode name | Payload | Available data |
|---|---|---|
| PAYLOAD_TYPE_CUSTOM_MODE_3 | 32 bytes | • SampleTimeFine<br>• Quaternion<br>• Angular velocity |