Document XD0201P, Revision A, 7 Jan 2020

Xsens DOT SDK Programming Guide for Android

Revision	Date	By	Changes
A	7 Jan 2020	XUF, ABO	Initial release

Table of Contents

1 Introduction

The **XsensDotSdk** is a software development kit for mobile application (supporting both iOS & Android platforms). This document addresses its usage in Android platform and Android developers can use this SDK in their own application to connect and measure & collect data from Xsens DOT wearable sensor platform.

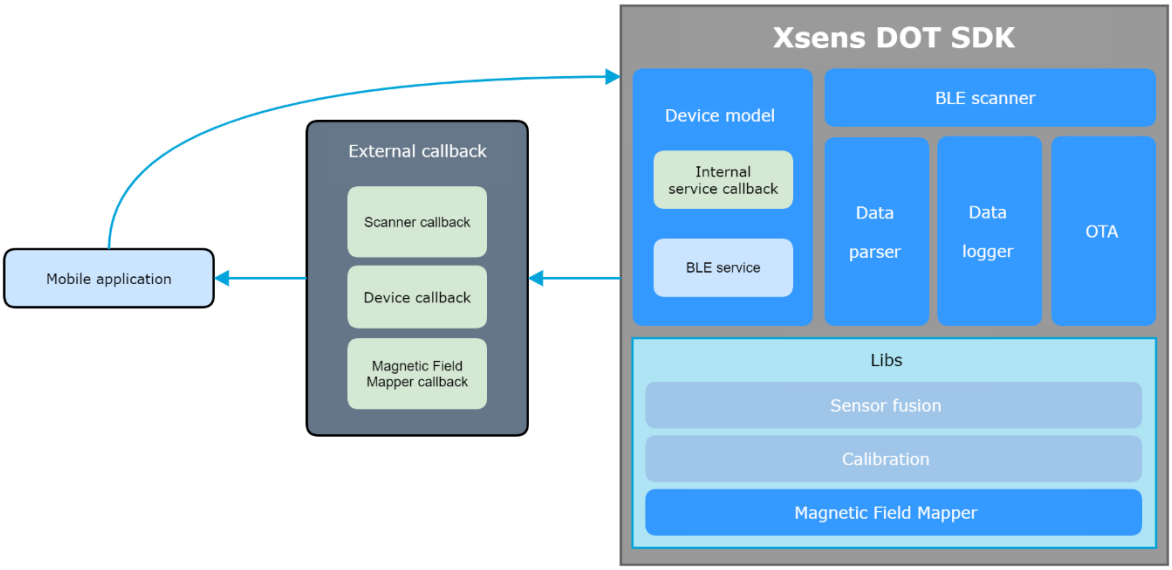

Figure 1: Xsens DOT Mobile SDK Architecture

The SDK also provides some public classes for developers to facilitate easier integration into specific application. The Xsens DOT sensors and the SDK works best with a smart phone that supports Bluetooth 5.0+, with Data Length Extension (DLE). Please refer to *Xsens DOT release notes* for detailed information about supported development platform.

Figure 1 shows the SDK components: it contains two interfaces for notifying the state of device and data output, and the different classes available for usage. Note that not every class can be new or referenced. The sensor fusion and calibration libraries are running on the Xsens DOT firmware.

This document should be used in conjunction with the XsensDotSdk Documentation available as a zipped file and can be download from this page. The following section (Section 2) provides information on getting started with SDK and Section 3 provides usage examples for the classes available.

2 Getting Started with SDK

2.1 Prerequisites

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

1. Make sure the **minSdkVersion** is 22+ (Android 5.1) in the build. Gradle (app level) file
2. Use androidx.* artifacts
3. Dependency workmanager:
   implementation "androidx.work:work-runtime:2.2.0 "

2.2 Import SDK package

This section addresses setup parameters and some practical considerations for proper usage of the XsensDotSdk. 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. 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 this SDK as shown below.

```
private void initXsSdk() {

    String version = XsensDotSdk.getSdkVersion();

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

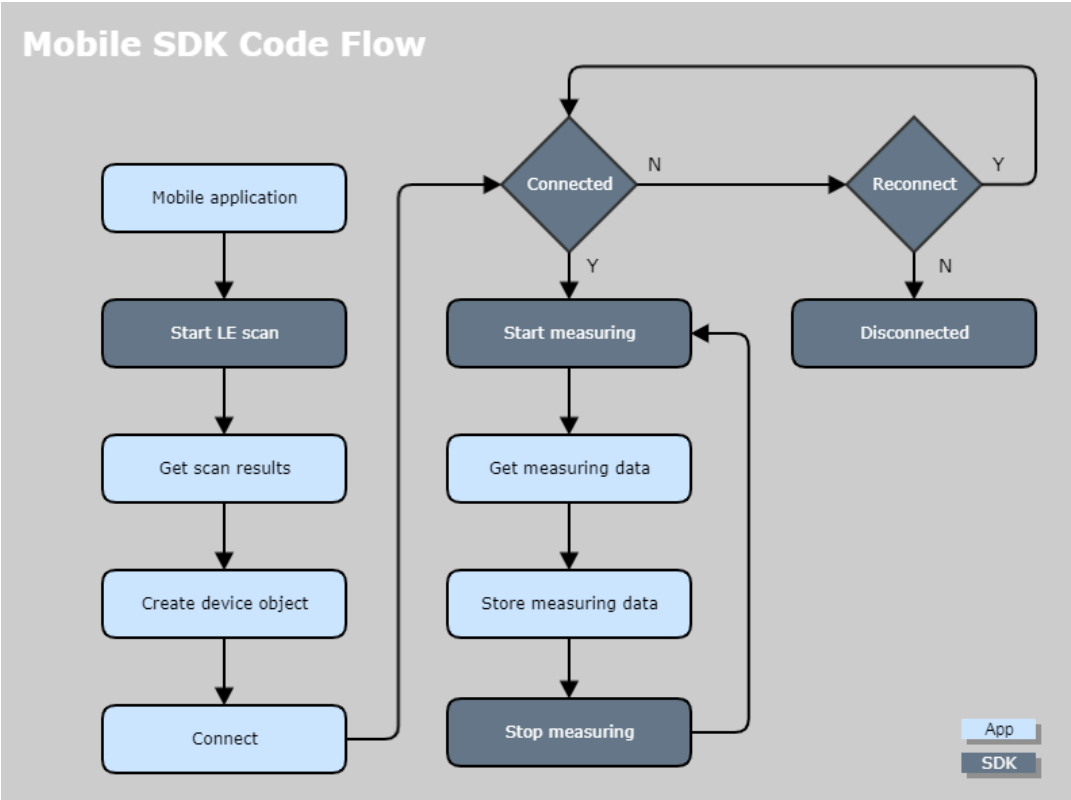

Figure 2: Mobile SDK Code Flow

The mobile SDK code flow is shown in Figure 2. This flow process can be used by Android developers after importing XsensDotSdk library into Android project and creating an SDK object. The first thing is starting BLE scan. Developers can obtain the scan result from a callback function and use this *BluetoothDevice* object to initialize *XsensDotDevice* class. Most of 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, the SDK will check if the reconnection feature is enabled or not. If it's enabled, a reconnection will start automatically.

After the sensor is connected to the mobile phone, developers can call *startMeasuring* function to notify the sensor to enter measurement mode, the measurement data will output from a callback function. There are 3 different measurement modes available for different data lengt – default mode, inertial data mode and orientation data mode, refer to Xsens DOT User Manual for more information. The *XsensDotLogger* class object can be used to collect the measured data and store the data to the mobile device.

2.3 Implement Interface

Developers can implement *XsensDotDeviceCb* and *XsensDotScannerCb* in one activity as shown below.

```
public class MainActivity extends AppCompatActivity
```

```
                    implements XsensDotDeviceCb, XsensDotScannerCb {
    ...
    ...
    ...
}
```

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.

2.4 Classes and Interfaces

The list of classes and available interfaces as part of Xsens DOT SDK is shown in Table 1.

Table 1: Classes in Xsens DOT SDK

Class	Purpose
XsensDotSdk	XsensDotSdk is the SDK main object, used for global settings such as enable debug or reconnect features.
XsensDotDevice	XsensDotDevice represents a Xsens DOT device object, including basic information and operations, measurement, OTA and MFM. Return the data by XsensDotDeviceCb and XsensDotMfmCb.
XsensDotData	XsensDotData contains all the measurement data, including acceleration, angular velocity and mag data, etc.
XsensDotLogger	XsensDotLogger is used to log measurement data. The default format is csv files.
XsensDotParser	XsensDotParser is a class for parsing data from the device via Bluetooth.
XsensDotScanner	XsensDotScanner is a class for scanning Xsens DOT device. Return the scanned device by XsensDotScannerCb.
OtaServerHostUri	For setting different host URI in different server environment such as beta and stable.

2.5 Interfaces

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

Table 2: Interfaces in Xsens DOT SDK

Class	Purpose
XsensDotDeviceCb	An interface for notifying device information, measurement data and OTA status.
XsensDotScannerCb	An interface for notifying LE scan result
XsensDotMfmCb	An interface for notifying MFM status and data of device

2.6 Permissions

The permissions used by this SDK are as listed in Table 3. Make sure these permissions are set and are part of AndroidManifest.xml file in your project.

Table 3: Permissions list

Permission	Purpose
INTERNET	Check the DFU state via server
ACCESS_NETWORK_STATE	Monitor network connection
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

The following sections gives usage examples for the XsensDotSdk.

3.1 Debugging Flag

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

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

This setting is disabled by default.

3.2 Reconnection Setting

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

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

3.3 LE Scan

For using 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 *XsensDotScannerCb* (i.e. an activity that implemented the XsensDotScannerCb 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();
    …
}
```

3.4 Connect

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

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

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

3.5 Data Measurement

The *XsensDotDevice* can report sensor data via *onXsensDotDataChanged* callback function. To use this, notify the device to enter the measuring mode, and 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 information

- getAcc
- getGyr
- getDq
- getDv
- getMag
- 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();
```

3.6 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();
```

3.7 Connect Multiple Devices

To connect to multiple devices, 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.

3.8 Over-the-Air firmware update

To start an OTA, implement the *XsensDotDeviceCb* callback and initiate an OTA request with *XsensDotDevice.checkOtaUpdatesAndDownload*.

```
class OtaActivity extends BaseActivity implements XsensDotDeviceCb {}

mXsDevice.checkOtaUpdatesAndDownload(OtaActivity.this,
OtaActivity.this);
```

Before performing the OTA upgrade, make sure that sensor is plugged in with the USB cable, otherwise the callback will inform *onXsensDotOtaDischarge* and terminate the OTA upgrade process.

```
public void onXsensDotOtaDischarge(String address) {
// do something, maybe update the UI
}
```

Call *onXsensDotOtaChecked()* to check if there is a new upgrade file on the server. SDK will automatically download the file and inform the result via o*nXsensDotOtaDownloaded*.

```
public void onXsensDotOtaChecked(String address, boolean result, String
version) {
  if (result && (version == null || version.isEmpty())) {
     // Your Xsens DOT firmware is up to date
  }
    // do something
  }
```

After a successful download, an OTA start command will be sent to the sensor if the upgrade file matches, otherwise *onXsensDotOtaFileMismatch* will be triggered;

```
public void onXsensDotOtaFileMismatch(String address) {
// Do something
}
```

The upgrade file will be transmitted to the sensor after the start command and each time a packet is sent, there will be a callback from *onXsensDotOtaUpdated*.

```
public void onXsensDotOtaDownloaded(String address, final int result) {
if (result == OTA_SUCCESS) {
        mXsDevice.setTrackerCharging(true);
        boolean isSuccess = mXsDevice.startOta();
        // do something
    }
}
```

If some packets are lost during the transmission, the sensor can notify the App to resend by using *XsensDotDevice.resendOtaContent*. When upgrade file transmission completes, send an end command. The sensor will verify the upgrade file and completes the OTA upgrade. A reboot is required after the OTA.

```
@Override
```

```
public void onXsensDotOtaUpdated(final String address, final int type,
final int result, final int total, final int current, final int
errorCode) {
      // do something
      switch (type) {
      case OTA_TYPE_START:
          // do something
          break;
      case OTA_TYPE_CONTENT:
          if (result != OTA_SUCCESS) {
              mXsDevice.resendOtaContent(current);
          }
          // do something
          break;
      case OTA_TYPE_END:
          if (result == OTA_SUCCESS) {
              progressDialog = ProgressDialog.show(OtaActivity.this,
"", "Rebooting. Please wait...", true);
              // do something
          }
          break;
}
}
```

3.9 Magnetic Field Mapper

To use Magnectic Field Mapper (MFM), *XsensDotMfmCb* should be initially implemented. Then set *XsensDotDevice.setXsensDotMfmCallback*.

```
public class MfmActivity extends BaseActivity implements
XsensDotDeviceCb, XsensDotMfmCb {}
```

After connecting to the sensor, you can start MFM by calling this function.

```
XsensDotDevice.startMfm();
```

During the MFM, sensor needs to rotate around all three axes and multiple directions to collect data. The following function call can be used to stop the MFM data collection.

```
XsensDotDevice.stopMfm();
```

After the stop, the lib will analyze the collected magnetic data during the rotation and return the result by *onXsensDotMtbDataChanged* callback

```
public void onXsensDotMtbDataChanged(String address, final int result,
final byte[] mtbData){
    if (result == MfmResult.ACCEPTABLE) {
        // do something
    } else if (result == MfmResult.GOOD) {
        // do something
    } else if (result == MfmResult.NOT_AVAILABLE) {
        // do something
    } else if (result == MfmResult.BAD) {
        // do something
    } else {
        // do something
}
}
```

The parameter of mtbData is the data after a successful MFM and can be written to sensor using *XsensDotDevice.writeMfmResultToDevice*.

```
XsensDotDevice.writeMfmResultToDevice(mtbData)
```

Unfinished tasks need to be clear in life cycle function onDestroy().

```
XsensDotDevice.cancelAsyncWorks(this);
```

3.10 Others

When the reconnection feature is enabled and the connection of device is lost, SDK will start reconnecting automatically. You can cancel the reconnecting by

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

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();
```