**Pointer MultiSense Mobile SDK**

Ver. 1.1

**Table of Contents:**

[**INTRO**](#_gjdgxs) **3**

[**GETTING THE MULTISENSE SDK**](#_30j0zll) **3**

[**ADDING THE SDK TO YOUR PROJECT**](#_1fob9te) **3**

[**BASICS**](#_3znysh7) **3**

[Definitions and terms](#_2et92p0) 3

[**ADDING YOUR FIRST MULTISENSE**](#_tyjcwt) **4**

[**READING GENERAL MULTISENSE DATA**](#_3dy6vkm) **6**

[**SET MULTISENSE CONFIGURATION**](#_1t3h5sf) **7**

[Sensors Activating](#_4d34og8) 7

[Setting Alerts Thresholds](#_2s8eyo1) 8

[Setting Timers](#_17dp8vu) 8

[**READING BLE DATA**](#_3rdcrjn) **10**

[**READING SENSORS DATA**](#_2jxsxqh) **13**

[**Example of scanning for the MultiSense device(s), observe it and read the data from MS**](#_1y810tw) **15**

# **INTRO**

Pointer’s mobile SDK is already integrated with the Pointer MultiSense wrapping with a friendly API. The SDK will let you take an advantage of all features included in the MultiSense (i.e. Temperature, Humidity, Accelerometer, Magnetic and Light sensors). All of them can be easily controlled through API and all specific functions are up to you.

# **GETTING THE MULTISENSE SDK**

You can download your Android SDK by opening and download the package from this link - [Pointer to put a link here]

# **ADDING THE SDK TO YOUR PROJECT**

You can add SDK to your project in a few steps:

1. Add the library package to your project
   1. Open your project in Android Studio.
   2. Click **File > New > New Module.**
   3. Click **Import .AAR Package** then click **Next**.
   4. Enter the location of the SDK AAR file then click **Finish**.
2. Make sure the library is listed at the top of your settings.gradle file, as shown here: include ':app', ':cellotrack\_sdk\_1.0.0-release'
3. Open the app module's build.gradle file and add a new line to the dependencies block as shown in the following snippet:

dependencies {

implementation project(":my-library-module")

}

1. Click **Sync Project with Gradle Files**.

In order for SDK to work, your app must have the [ACCESS\_FINE\_LOCATION](https://developer.android.com/reference/android/Manifest.permission.html#ACCESS_FINE_LOCATION) or [ACCESS\_COARSE\_LOCATION](https://developer.android.com/reference/android/Manifest.permission.html#ACCESS_COARSE_LOCATION) permission.

# **BASICS**

The MultiSense uses [BLE](https://en.wikipedia.org/wiki/Bluetooth_low_energy) to communicate with the mobile device.

## **Definitions and terms**

|  |  |
| --- | --- |
| **MultiSense/TAG/MS** | A Pointer’s MultiSense BLE device |
| **Master** | Device controlling a MultiSense |
| **MAC Address** | A six-byte hex number which represents a physical address of the MultiSense device, two devices cannot share the same MAC Address |
| **RSSI** | Received Signal Strength Indicator, how powerful a signal is, measured in Dbm. Can be converted in Meters/Feets |
| **OTA** | Over The Air (protocol) |
| **Advertise /**  **BLE Advertisement / Advertisement /**  **BLE Ad** | A state that the MultiSense advertises itself when not connected |
| **Device Name** | The name advertised by the MultiSense |
| **Write** | The way to send information to the MultiSense |
| **Read** | The way to get data once (for each read) from the MultiSense |
| **Rx** | Receive data to the MultiSense |
| **Tx** | Transmit data from the MultiSense |
| **Log / Memory / DataLogger** | A built-in a MultiSense a Memory module that tracks and saves all info from sensors while MultiSense is not connected over Bluetooth |

# **ADDING YOUR FIRST MULTISENSE**

MultiSenses are represented over MultiSenseDevice objects in SDK. While working with the SDK you need to use this four objects to access them:

**MultiSenseManager**

The MultiSenseManager is a provider class for MultiSenseScanner and MultiSenseObserver objects.

*Example:*

MultiSenseManager multiSenseManager = new MultiSenseManager(context);

MultiSenseScanner multiSenseScanner = multiSenseManager.createScanner();

MultiSenseObserver multiSenseObserver = multiSenseManager.createObserver();

**MultiSenseScanner**

MultiSenseScanner scans new devices if you will click on the button on a Tag.

The MultiSense transmits its advertisements in a very short period of time. So in order to receive it efficiently, it is recommended to scan for long periods of time from a master side.

Use this method to perform scanning:

public void scan(MultiSenseDeviceCallback callback);

To stop scanning use:

public void stopScan();

There’s a method indicating whether MultiSenseScanner is scanning at the moment

public boolean isScanning();

**MultiSenseDeviceCallback**

Interface for retrieving results from MultiSenseScanner.

This method will be called when an error encountered:

public void onError(int errorType, String message);

Parameters:

int errorCode: one of MultiSenseManager.ErrorCode

String message: Human-readable error message

This method will be called when the new device was discovered as a result of pressed button:

public void onChange(MultiSenseDevice multiSenseDevice);

**MultiSenseObserver**

MultiSenseObserver monitors the state of the device

void startObserveTags(MultiSenseObserverCallback callback);

To stop monitoring use:

void stopObserveTags();

MultiSenseObserver object’s properties and methods:

* distance limit for observing

public void setDistanceLimit(float distanceMeters);

* a Boolean value indicating whether MulitSenseObserver is observing at the moment

public boolean isObserving();

* add a MultiSense with mac address to observing

public void startObserveTags(MultiSenseDeviceCallback callback);

* remove a MultiSense with mac address from observing

public void removeTag(String mac);

* remove all MultiSense from observing

public void stopObserveTags();

* save MultiSense device configuration to the device on the next communication session

public void saveConfiguration(String mac, MultiSenseConfiguration configuration);

* save MultiSense device settings to the device on the next communication session

public void saveSettings(String mac, MultiSenseSettings settings);

**MultiSenseObserverCallback**

Interface for retrieving results from MultiSenseObserver.

This method will be called when an error encountered:

public void onError(int errorType, String message);

Parameters:

int errorCode: one of MultiSenseManager.ErrorCode

String message: Human-readable error message

This method will be called after successful sensors reading:

public void onChange(MultiSenseDevice multiSenseDevice);

This method for tracking log reading progress:

public void onReadingLoggerStatusChange(String address,

MultiSenseReadingLoggerStatus status);

Parameters:

String address: MAC-address of the device, from which logs are being read

MultiSenseReadingLoggerStatus status: reading state

**MultiSenseReadingLoggerStatus**

Class containing information about current state of reading logger data.

Returns current status. Can be one of MultiSenseReadingLoggerStatus.Status:

LOADING, SUCCESS, ERROR

public Status getStatus()

Returns reading percent

public float getPercent()

# **READING GENERAL MULTISENSE DATA**

**MultiSenseDevice**

Besides sensors and timers the MultiSense has general data which can be retrieved from MultiSenseDevice object’s properties:

* Name of MultiSense device

public String getName()

* MAC of MultiSense device

public String getAddress()

* Advertisement data of MultiSense device (**see** [**below**](#26in1rg))

public MultiSenseAdvertisement getAdvertisement()

* RSSI

public int getRssi()

* Configuration of MultiSense device (**see** [**below**](#44sinio))

public MultiSenseConfiguration getConfiguration()

* Sensors of MultiSense device (**see** [**below**](#z337ya))

public List<MultiSenseSensors> getSensors()

**\*Note:** getSensors() will return at least one MultiSenseSensors object

* Settings of MultiSense device (**see** [**below**](#1ksv4uv))

public MultiSenseSettings getSettings()

* Logger data of MultiSense device

public String getLoggerData()

**\*Note:** This is developer tool, returns raw data in order to put them in a file for debugging

# **SET MULTISENSE CONFIGURATION**

The MultiSense's configuration is the settings which include: which sensors should be activated, alert scheduling, timers settings and power level.

## **Sensors Activating**

The MultiSense's sensors won't start to read and send sensor data until they are configured. The alerts from each sensor won't work until the specific sensor is enabled. To enable specific sensor you need to save MulitiSenseConfiguration object with related MultiSenseEnabledSensors object fields set to true. You can use existing MulitiSenseConfiguration object to make incremental changes.

*Example:*

MultiSenseConfiguration deviceConfiguration =

multiSenseDevice.getConfiguration();

MultiSenseEnabledSensors enabledSensors = new MultiSenseEnabledSensors

.Builder()

.setLight(true)

.setHumidity(true)

.create();

MultiSenseConfiguration newMultiSenseConfiguration =

new MultiSenseConfiguration

.Builder(deviceConfiguration)

.setSensorMask(enabledSensors)

.create();

multiSenseObserver.saveConfiguration(deviceConfiguration);

***Accelerometer sensor***

public Builder setAccelerometer(boolean enabled)

***Hall-effect (magnetic) sensor***

public Builder setHallEffect(boolean enabled)

***Ambient Light sensor***

public Builder setLight(boolean enabled)

***Temperature sensor enable***

public Builder setTemperature(boolean enabled)

***Humidity sensor enable***

public Builder setHumidity(boolean enabled)

***Set prevent pushbutton power down***

Normally a long button press (>4 sec) will turn the MultiSense OFF (inactive mode). To prevent power down by the button set enabled parameter to true:

public Builder setPowerDown(boolean enabled)

***Set ‘Tx on violations only’ mode***

If "TX on violations only" mode is enabled, the Relaxed timer shall be used for determine sensors sampling rate only (without transmission), but the Violation and Proximity timers shall be working normally.

public Builder setTxReason(boolean enabled)

## **Setting Alerts Thresholds**

Each sensor other than the magnetic sensor, has a threshold that will determine if an alert will happen or not. Each sensor can be configured separately through MultiSenseConfiguration builder object :

***Light threshold***

public Builder setLightThreshold(int lightThreshold)

lightThreshold range: all even numbers in [0:510]

***Temperature threshold***

public Builder setTempUpper(int tempUpper)

if the temperature falls below this an alert will be sent. Range: [-40:80]°C

public Builder setTempLower(int tempLower)

if the temperature rises higher than this an alert will be sent. Range: [-40:80]°C

***Humidity threshold***

public Builder setHumidityUpper(int humidityUpper)

if the humidity percentage falls below this an alert will be sent. Range: [0:100]%

public Builder setHumidityLower(int humidityLower)

if the humidity percentage rises higher than this an alert will be sent. Range: [0:100]%

## **Setting Timers**

MultiSense has four configurable timers, each timer will determine the MultiSense's advertise time in different states. The first three timers affect the Temperature and Humidity sampling.

Besides that, there are few asynchronous events that will cause a single transmission (TX-Reason):

– Pressing the button.

– Impact or free-fall event generated by the accelerometer (crossing a preconfigured threshold).

– Sensing a change in the magnetic field (opening/closing of a door or window that the permanent magnet is installed on).

– Starting from FW version 4.44 (4V44) and up, also crossing the light threshold to either direction (darkness <--> light).

***Proximity timer***

This timer is used all the time **independently** from the other timers.

To configure the proximity timer, you need to set the time in seconds between two transmissions caused by this timer.

Range: all the even numbers in [2:86400] sec (up to 24h).

public Builder setProximityTimer(int proximityTimer)

***Relaxed timer***

The Relaxed Timer is used when the temperature and humidity are within their limits. Meaning that if, for example, the temperature limits were configure to [0:20]°C. as long as the **measured** temperature (when the MultiSense wakes for any reason and samples the sensors) is within this range, the next temperature sampling will be after the number of seconds configured in the relaxed timer has passed.

**Important Note:** remember that the MultiSense sleeps most of the time. even if the temperature of the MultiSense exceeds the limits defined, as long as it didn't happen when it was awake and sampling, **it didn't happen (!)** and the MultiSense will continue sleeping. Same goes for humidity.

public Builder setRelaxedTimer(int relaxedTimer)

***Violation timer***

This timer is used when the temperature and humidity are out of the configured limits. This is the counterpart for the relaxed timer and the same applies to it. see Relaxed Timer above.

public Builder setViolationTimer(int violationTimer)

***Alert timer***

This timer defines the timeout in seconds before alert generation after a limit is crossed.

public Builder setAlertTimer(int alertTimer)

***Setting Memory***

Memory can be set through MultiSenseSettings builder object:

*Example:*

MultiSenseSettings multiSenseSettings = new MultiSenseSettings

.Builder()

.setLoggerEnabled(true)

.setTransmissionPowerLevel(3)

.create();

multiSenseObserver.saveSettings(mac, multiSenseSettings);

Set power level:

public Builder setTransmissionPowerLevel(int transmissionPowerLevel)

Enabling/disabling logger:

public Builder setLoggerEnabled(boolean loggerEnabled)

# **READING BLE DATA**

When MultiSense is in active state it sends BLE Advertisements. To read BLE Ad’s data use the following properties of MultiSenseAdvertisement object:

* Company ID (reserved)

public int getCompanyID()

* OTA version, integer value

public int getOtaVersion()

* OTA version, human-readable string

public int getOtaVersionName()

* Battery level

public int getBatteryLevel()

* Sensors mask (**see** [**below**](#lnxbz9))

public MultiSenseEnabledSensors getSensorMask()

* BOM mask (**see** [**below**](#lnxbz9))

public MultiSenseEnabledSensors getBomMask()

* TX Reason (**see** [**below**](#35nkun2))

public int getTxReason()

* Human-readable firmware version string

public String getFirmwareVersionName()

* Major firmware version

public int getMajorFirmwareVersion()

* Minor firmware version

public int getMinorFirmwareVersion()

* Connection password scrambled

public boolean isConnectPasswordScrambled()

* Sensor data stream scrambled

public boolean isSensorDataStreamScrambled()

"BOM mask" represents which sensors are assembled on the MultiSense HW board. It is hardcoded and determined during the production phase.

"Sensors mask" represent which sensors are enabled and active (configurable).

MultiSenseEnableSensors (Sensors/BOM mask):

MultiSenseEnableSensors object’s properties contain MultiSense’s sensors settings. If sensor is enabled and active the property returns true.

* Prevent pushbutton power down

public boolean isPowerDown()

* TX Reason

public boolean isTxReason()

* Accelerometer

public boolean isAccelerometer()

* Light

public boolean isLight()

* Hall effect (magnetic)

public boolean isHallEffect()

* Humidity (for MultiSense-TH only)

public boolean isHumidity()

* Temperature

public boolean isTemperature()

* Logger enabled

public boolean isLoggerEnabled()

MultiSenseSensors.TxReason

According to MultiSense transmitting policy (see [**Setting Timers**](#w6y6glvoopvb)**)** the MS can be configured to periodically transmit the Ads. Transmitting can be caused by the timers or a few asynchronous events that will cause a single transmission. To obtain an Event causing transmission read MultiSenseAdvertisement.getTxReason(), it will be one of:

* Reserved

int TXREASON\_RESERVED = 0x00

* Relaxed timer

int TXREASON\_RELAXED = 0x01

* Violation timer

int TXREASON\_VIOLATION = 0x02

* Power up

int TXREASON\_POWER\_UP = 0x03

* Power down (by button)

int TXREASON\_POWER\_DOWN = 0x04

* Proximity

int TXREASON\_PROXIMITY = 0x05

* Push button

int TXREASON\_BUTTON = 0x06

* Hall-effect (magnetic sensor) changed state

int TXREASON\_HALL = 0x07

* ACC Impact

int TXREASON\_ACC\_IMPACT = 0x08

* ACC Free-fall (Acc. threshold MSB == 1)

int TXREASON\_FREE\_FALL = 0x09

* Package state changed

int TXREASON\_PACKAGE\_STATE\_CHANGE = 0x0A

* Shut down due to dead-bat

int TXREASON\_SHUTDOWN\_BATTERY = 0x0B

MultiSenseSettings

Data transmission (Data Logger Functionality) general settings

* Transmission power level, as it reported by MultiSense

public int getTransmissionPowerLevel()

* Transmission power level, in decibel

public int getTransmissionPowerLevelDb()

MultiSenceConfiguration

MultiSenseConfiguration object contains components settings that can cause transmission - timers, temp and humidity, accelerometer and ALS (See [**Set MultiSense Configuration**](#oumkju8z0ymi))

* Proximity timer

public int getProximityTimer()

* Relaxed timer

public int getRelaxedTimer()

* Violation timer

public int getViolationTimer()

* Alert timer

public int getAlertTimer()

* Sensors enabled

public MultiSenseEnabledSensors getSensorMask()

* Temperature lower

public int getTempLower()

* Temperature upper

public int getTempUpper()

* Humidity lower

public int getHumidityLower()

* Humidity upper

public int getHumidityUpper()

* Impact threshold

public float getImpactThreshold()

* Light threshold

public int getLightThreshold()

# **READING SENSORS DATA**

MultiSenseSensors

To obtain MultiSense’s sensors data at the moment use MultiSenseSensors object’s properties:

* Temperature

public Float getTemperature()

* Humidity

public Float getHumidity()

* Light

public Float getLight()

* Magnet is present

public Boolean isMagnetPresent()

* Package is open

public Boolean isOpenPackage()

* Accel. X

public Float getAccelerometerX()

* Accel. Y

public Float getAccelerometerY()

* Accel. Z

public Float getAccelerometerZ()

* Battery level

public Integer getBatteryLevel()

* TX Reason

public Integer getTxReason()

* TH States

public MultiSenseThStates getThStates()

* Create Date

public Date getCreateDate()

MultiSenseThStates

Besides TX Reason Events the MultiSense can provide Temp & Humidity sensors state as well as Magnetic sensor presence. MultiSense can also show package state (open/close) based on ALS (light sensor) data. To get these states use MultiSenseThStates object’s properties:

* Temperature is upper

public boolean isTempUpper()

* Temperature is violation

public boolean isTempViolation()

* Temperature alert

public boolean isTempAlert()

* Magnet is present

public boolean isMagnetPresent()

* Humidity is upper

public boolean isHumidityUpper()

* Humidity is violation

public boolean isHumidityViolation()

* Humidity alert

public boolean isHumidityAlert()

* Package is open

public boolean isOpenPackage()

# 

# **Example of scanning for the MultiSense device(s), observe it and read the data from MS**

// Create manager

MultiSenseManager multiSenseManager = new MultiSenseManager(this);

// Create scanner

MultiSenseScanner multiSenseScanner = multiSenseManager.createScanner();

// Create observer

MultiSenseObserver multiSenseObserver = multiSenseManager.createObserver();

// Begin scanning

multiSenseScanner.scan(new MultiSenseDeviceCallback() {

@Override

public void onError(int errorType, String message) {

Log.e(TAG, message)

}

@Override

public void onChange(MultiSenseDevice multiSenseDevice) {

// Adding found device by mac address to observer

multiSenseObserver.addTag(multiSenseDevice.getAddress());

}

});

multiSenseScanner.stopScan();

// Begin observing

multiSenseObserver.startObserveTags(new MultiSenseDeviceCallback() {

@Override

public void onError(int errorType, String message) {

Log.e(TAG, message)

}

@Override

public void onChange(MultiSenseDevice multiSenseDevice) {

// Reading device general info

String mac = multiSenseDevice.getAddress();

String name = multiSenseDevice.getName();

int rssi = multiSenseDevice.getRssi();

String firmwareRevision =

multiSenseDevice.getAdvertisement().getFirmwareVersionName();

String hardwareRevision =

multiSenseDevice.getAdvertisement().getOtaVersionName();

// Reading sensors data

List<Float> temperatureSamples = new ArrayList<>();  
 List<Float> humiditySamples = new ArrayList<>();  
 for(MultiSenseSensors sensorSample : multiSenseDevice.getSensors()) {  
 if (sensorSample.getTemperature() != null) {  
 temperatureSamples.add(sensorSample.getTemperature());  
 }  
 if (sensorSample.getHumidity() != null) {  
 humiditySamples.add(sensorSample.getHumidity());  
 }  
 }

}

});