Permissions

The following permissions are required by the application:

Search for devices:

```
Manifest.permission.ACCESS_FINE_LOCATION;
```

Wifi devices and update

```
Manifest.permission.INTERNET);
Manifest.permission.ACCESS_NETWORK_STATE;
Manifest.permission.ACCESS_WIFI_STATE;
Manifest.permission.CHANGE_WIFI_STATE;
```

Bluetooth Devices

```
Manifest.permission.BLUETOOTH;
Manifest.permission.BLUETOOTH_ADMIN
```

Update

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

Permissions

Permissions can be validated using the following function

```
boolean[] verifyPermissions() {

   boolean[] permissions = {false, false};
   if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {

      if (ContextCompat.checkSelfPermission(this,Manifest.permission.ACCESS_COARSE_LOCATION)==PackageManager.PERMISSION_DENIED{
         ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.ACCESS_COARSE_LOCATION}, 1);
         return permissions;
      }
   if(ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_WIFI_STATE) == PackageManager.PERMISSION_GRANTED
      && ContextCompat.checkSelfPermission(this, Manifest.permission.CHANGE_WIFI_STATE) == PackageManager.PERMISSION_GRANTED){
      permissions[0] = true;
   }

   if(ContextCompat.checkSelfPermission(this, Manifest.permission.BLUETOOTH) == PackageManager.PERMISSION_GRANTED
      && ContextCompat.checkSelfPermission(this, Manifest.permission.BLUETOOTH_ADMIN) == PackageManager.PERMISSION_GRANTED
      && getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE))

         permissions[1] = true;
   }
   else {
      permissions[0] = true;
      permissions[1] = getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE);
   }
   return permissions;
}
```

Initializations

```
LeicaSdk.InitObject initObject = new LeicaSdk.InitObject("commands.json");
LeicaSdk.init(getApplicationContext(), initObject);
```

- init SDK with JSON file(which holds informations about commands and responses)
- the JSON file (here “commands.json”) has to be located in app/src/main/assets/

- Receivers

```
DeviceManager.getInstance(this).registerReceivers(this);
```

- this has to be done once. Register receivers for internally register receivers for wifi and bluetooth adapters
  “this” is the activity

```
void onDestroy() {
   deviceManager.unregisterReceivers();
}
```

IMPORTANT: unregister receivers when the activity is being finished

- Logging:
Specify the minimum logLevel for the sdk

```
LeicaSdk.setLogLevel(Log.VERBOSE);
```

Set Keys for SDK functionality

The following code enable certain functionalities of the code, such as Search for available devices

```
AppLicenses appLicenses = new AppLicenses();
LeicaSdk.setLicenses(appLicenses.keys);
```

Search for devices

```
// "this" is the activity
  DeviceManager deviceManager = DeviceManager.getInstance(this);
  deviceManager.setFoundAvailableDeviceListener(this); //"this" implements DeviceManager.FoundAvailableDeviceLis-
tener
  deviceManager.setErrorListener(this); // "this" implements ErrorListener to receive errors from the deviceMana-
ger

// permission is a boolean[] and represents the permission/ability to use wifi or bluetooth. permission[0] is
// for wifi and permission[1] is for bluetooth
  boolean[] permissions = verifyPermissions();
  deviceManager.findAvailableDevices(permissions);
```

- get object of DeviceManager and trigger search

when an available device is found "**public void onAvailableDeviceFound(Device device)**" will be called.

(for showing results in the UI, it is recommended to use device.getDeviceName())

See: SearchDevicesActivity.png in ...\Documentation\Diagrams

Search for Bluetooth Devices

Devices that require pairing:
- S910
- D810
- X3/X4

When these devices are paired to an android phone, they will be shown as available, regardless if they are reachable or not, if the device is not reacheable the sdk will try and fail to connect. This is done based on Android OS technical restrictions.

Search for WIFI devices:

In WIFI there are two options:

- The device itself creates a hotspot(hotspot mode).
- The device and the phone/tablet are connected to the same WIFI.

IMPORTANT:
In hotspot mode it is needed to connect to the device WIFI hotspot of the device first, so that the SDK can find the device.

Stop searching for devices

To stop searching for devices use the method – `deviceManager.stopFindingDevices();`

Example:

```
    void stopFindingDevices() {
      findDevicesRunning = false;
      deviceManager.stopFindingDevices();
    }
```

Refreshing the search process - findDevicesTimer

The search process needs to be refreshed after a period of time. This time can be custom set on app side. The refreshing is needed, because found devices in the past can not longer be available in the present. Devices that are paired on settings will be shown regardless if they are available or not.
- The recommended time interval is 10 seconds.

Example:
To start the findDevicesTimer:

```
// a) because already found devices may be out of reach by now,
// b) the user may changed adapter settings meanwhile
if (findDevicesTimer == null) {
      findDevicesTimer = new Timer();
       findDevicesTimer.scheduleAtFixedRate(new TimerTask() {
                  @Override
             public void run() {
                       findAvailableDevices();
                  }
      }, findAvailableDevicesDelay, findAvailableDevicesPeriod);
}
```

To stop findDevicesTimer

```
void stopFindingDevices() {
      final String METHODTAG = ".stopFindingAvailableDevices";
       Log.i( CLASSTAG, METHODTAG + ": Stop find Devices Task and set BroadcastReceivers to Null" );
      findDevicesRunning = false;
      deviceManager.stopFindingDevices();
}
```

Get Available Commands

Before sending a command to the device, it is needed to know which commands the device supports. To retrieve the supported commands, just call getAvailableCommands method on the device

```
CurrentDevice.getAvailableCommands()
```

Sending commands to device

The SDK handles commands one after another (synchronous). For most of the commands (except for some Bluetooth commands which do not have an answer) a response object is created containing the response data. The data needs to be extracted from that response object.

A DeviceException will be thrown if a command is sent while another command is still in progress.

Sending a command works in three phases.

Phase 1
- calling one of the methods to send a command, for example `device.sendCommand()`, and assign the returned Response object.

Phase 2
- Waiting for data to be received from the device by calling `response.waitForData()` of the Response object.
NOTE: this call will block until either data is received or a timeout hits. The default timeout is 10 seconds.
So, it is recommended to make the call NOT IN THE MAIN THREAD.

Phase 3
- After either data is received or a timeout hit, the Response needs to be parsed.
If a timeout hit or any other error occured the error object of the response class will be not null and it will contain more information about the error.
If successful, the error object will be null and the parsing of the data within the Response object is possible.

Sending commands to device - Example

Example:

```
     // send a measure distance command and wait until data are returned
     ResponseBLEMeasurements response = (ResponseBLEMeasurements) currentDevice.sendCommand(Types.Commands.Distance);

     // wait until the Leica device has processed the commands and transferred the data
     response.waitForData();

    // get the data
    MeasuredValue data = response.getDistanceValue();
    }
```

Most important specific Response Objects:

- ResponsePlain: Handle the custom Commands
- ResponseBLEMeasurement
- ResponseYetiMeasurement
- ResponseWifiMeasurment
- ResponseDeviceInfo
- ResponseTemperature
- ResponseImage

Parsing a Response

Depending on the connection type (bluetooth or WIFI) and the command which is sent, a specific Response class is returned.
The best way to parse a Response is to cast it to the specific Response class and get the data directly from its member variables.

Example

```
if (response.getError() != null) {
   Log.e(CLASSTAG, METHODTAG+": response error: " + response.getError().getErrorMessage());
   return;
}

if (response instanceof ResponseDeviceInfo) {

   edm_temp.setText(String.valueOf(response.getTemperatureDistanceMeasurementSensor_Edm()));
   hz_temp.setText(String.valueOf(response.getTemperatureHorizontalAngleSensor_Hz()));
   v_temp.setText(String.valueOf(response.getTemperatureVerticalAngleSensor_V()));
   ble_temp.setText(String.valueOf(response.getTemperatureBLESensor()));

} else if (…
```

Receiving async data from device

Data can be received from the device asynchronously, if for example distance is measured by pressing the buttons on the disto device itself.

ReceivedDataListener is an asynchronous callback which is triggered if an action is performed on a Leica device (E.g. measurement button is pressed on DISTO). Data is wrapped in different packages and needs to be extracted from the user. For details about how to parse the incoming data, please look into the example project.

Implement ReceivedDataListener:

```
    public void onAsyncDataReceived(ReceivedData receivedData) {}
```

The activity needs to register for the callback:

```
    device.setReceiveDataListener(this); // "this" is the activity
```

Bluetooth asynchronous data - multiple linked packets

Bluetooth devices can send multiple data packets that are in relation to each other.
The command Distance (*g*) its the only bluetooth command retrieving a response: `ResponseBLEMeasurements`
When a synchronous bluetooth command is sent, the response will hold all the values, already converted

Asynchronous data will come in pairs Measurement – Unit, and depending on how the device is configured it will receive
- Distance – Distance Unt
- Inclination – Inclination Unit
- Direction

Example

```
ReceivedBleDataPacket receivedBleDataPacket = (ReceivedBleDataPacket)receivedData.dataPacket;
if(receivedBleDataPacket != null) {
   String id = receivedBleDataPacket.dataId;
   switch (id) {

      case Defines.ID_DS_DISTANCE: {
         if (deviceIsInTrackingMode == false) {
            //clears only UI
            distance.setText(R.string.blank_value);
            distanceUnit.setText(R.string.blank_value);
            inclination.setText(R.string.blank_value);
            inclinationUnit.setText(R.string.blank_value);
            direction.setText(R.string.blank_value);
            directionUnit.setText(R.string.blank_value);
         }
         float data = receivedBleDataPacket.getDistance();    // distance is always float
         distanceValue = new MeasuredValue(data); // save the value, a unit may come in the next packet
         hasDistanceMeasurement = true
      }
      break;
```

Bluetooth asynchronous data - multiple linked packets – Example

```
      //Assign units to the distance measured Value object and do the conversion
         case Defines.ID_DS_DISTANCE_UNIT: {
            if (distanceValue != null) {
               short data = receivedBleDataPacket.getDistanceUnit();    // unit is always short
               distanceValue.setUnit(data);
               distanceValue.convertDistance();
               distance.setText(distanceValue.getConvertedValueStrNoUnit());
               distanceUnit.setText(distanceValue.getUnitStr());
               hasDistanceMeasurement = true;
            }
         }
         break;
```

Receiving errors from device

- implement ErrorListener and set it for the device object

```
    device.setErrorListener(this);   // "this" is the activity
```

Disconnect from a device

If a device gets disconnected `onConnectionStateChanged(Device device, Device.ConnectionState state)` gets called and state equals `Device.ConectionState.disconnected`.
If a device is disconnected, the device object **cannot call connect again** and **cannot send command any more**.

To disconnect manually from a device just call disconnect method on the device class.

Example:

```
currentDevice.disconnect();
```

Reconnect to a device

If a connected device turned off or got out of range, the device object of the SDK will automatically reconnect.
In this case `onConnectionStateChanged` is called with the state “Reconnecting”. If the device is turned on again/comes within range again, the device object of the SDK will automatically establish the connection.

If the connection is successful `onConnectionStateChanged` is called with the state “Connected”.

If `device.disconnect` is called on a connected device, there will not be a auto reconnection. To reconnect to the device, the real device has to be in range and available.

IMPORTANT:
It is NOT recommended to save the device object and call `device.connect` at any time after calling `device.disconnect` .
Rather it is recommended to do a new search `deviceManager.findAvailableDevices(permissions);` and use the device object returned by the new search.

Reconnect to a device

For automatic reconnection there exists a helper class “ReconnectionHelper”

```
ReconnectionHelper reconnectionHelper = new ReconnectionHelper(device, getApplicationContext());
 // device is the device which should be reconnected to
 // "this" is the activity implements ReconnectionHelper.ReconnectListener
    reconnectionHelper.setReconnectListener(this);
    reconnectionHelper.setErrorListener(this);
```

- to setup ReconnectionHelper:

```
reconnectionHelper.startReconnecting();
```

- trigger reconnection:

If a device is reconnected "public void onReconnect(Device device)" of ReconnectionHelper.ReconnectListener will be called.
"device" will be a new object of class Device.

Retrieving connected devices

The app can take care of saving all connected devices. But the `DeviceManager` instance also does it. By calling `deviceManager.getConnectedDevices()` it will return an array with all connected devices.

Example:
DeviceManagerLeica deviceManager = [DeviceManagerLeica sharedInstance:self errorDelegate: self];
NSArray* connectedDevices = [deviceManager getConnectedDevices]

Bluetooth HID Mode

The HID mode basically means to get notified for bluetooth characteristics or not.
To turn HID mode ON, call [pauseBleConnection
] on the device. This equals notifications about characteristics ARE NOT received by SDK.
To turn HID mode OFF, call [startBleConnection] on the device. This equals notifications about characteristics ARE received by SDK.

IMPORTANT
The device actually will need about 1.5 seconds to actually turn HID mode on/off
Methods `startBTConnection` and `pauseBTConnection` return callback `Device.BTConnectionCallback()`

Example:

```
    if(currentDevice != null) {
        currentDevice.startBTConnection(new Device.BTConnectionCallback() {
            @Override
            public void onFinished() {
                Log.d(METHODTAG, "NOW YOU CAN SEND COMMANDS TO THE DEVICE");
            }
        });
    }
```

Update a bluetooth Yeti device

The FirmwareUpdate object contains information about the containing firmware binaries like version, brand, etc...

1. Get Available Firmware:

Get the new Binaries for updating:
Returns null if the process failed or if there are no new Binaries

```
public FirmwareUpdate getAvailableFirmwareUpdate() throws DeviceException
```

With this FirmwareUpdate object the update can be performed

2. Do update

Update YetiDevice with the information from firmwareUpdate obtained in step1

@firmwareUpdate*: FirmwareUpdate*: contains the binaries to be updated – for DistoDevice (APP, EDM,...) and Components: Disco, fta, ….
@listener : UpdateDeviceListener, receives feedback from update process.

```
public ResponseUpdate updateDeviceFirmwareWithFirmwareUpdate(final FirmwareUpdate firmwareUpdate, final
UpdateDeviceListener listener)throws DeviceException
```

Reinstall a bluetooth Yeti device

The FirmwareUpdate object contains information about the containing firmware binaries like version, brand, etc...

1. Get Available Firmware:

Get the current Binaries for updating:
Returns null if the process failed or if there are no new Binaries

```
public FirmwareUpdate getReinstallFirmware() throws DeviceException
```

With this FirmwareUpdate object the update can be performed

2. Do Reinstall:

```
public ResponseUpdate updateDeviceFirmwareWithFirmwareUpdate(final FirmwareUpdate firmwareUpdate, final
UpdateDeviceListener listener)throws DeviceException
```

ResponseUpdate will hold the result of the update process, if the process was not successfull then:
ResponseUpdate.get error != null

Update a Bluetooth Yeti device

IMPORTANT: It can happen that the JSON file contains information about the component but the component is not available, i.e. Disco device is not connected to Yeti. So the app needs to validate that the components are available for update.

Example:

```
ResponsePlain responsePlain = yetiDevice.getUpdate().getValue(serialCommand, yetiDevice);

if(responsePlain.getError() != null
      && responsePlain.getError().getErrorCode() != ErrorDefinitions.DEVICE_FTA_IN_SWD_MODE_CODE ){
                        Log.i("isComponentConnected", "Component is not connected");
                        return false;
}
```

If the component is not available then you should remove the components binaries from the firmware update object

```
List<FirmwareComponent> fwComponents = fwUpdateToUse.getComponents();
Iterator<FirmwareComponent> iterator = fwComponents.iterator();
while (iterator.hasNext()) {
      FirmwareComponent fwComponent = iterator.next(); // must be called before you can call i.remove()     isCom-
ponentConnected = updateFirmwareDeviceHelper.
                                          isComponentConnected(
                                                currentDevice,
                                                fwComponent.getSerialCommand(),
                                                fwComponent.getVersionCommand()
                                          );
      if(isComponentConnected == false){
            iterator.remove();
      }
}
```

Offline Update a Yeti Bluetooth device

The SDK only needs a valid FirmwareUpdate object to do the update. If the FirmwareUpdate object is existing, there is no further online/internet connection needed. So to perform the update offline (without internet connection), the App needs to provide a valid FirmwareUpdate object.
For example, after retrieving the FirmwareUpdate object, the App can save the data contained within the FirmwareUpdate in a database or file. From there the App can load the data and simply recreates the FirmwareUpdate object with all its members and value

The android sample project provides a helper class - `OfflineFirmwareUpdateHelper`

Example:

To save FirmwareUpdate objet into disk

```
    String jsonResult = offlineFirmwareUpdateHelper.saveNextFirmwareUpdate(
                                        fwUpdate,
                                        currentDevice.getDeviceID(),
                                        currentSwVersion,
                                        yetiInformationActivity.getApplicationContext()
                            );
```

To retrieve FirmwareUpdate Object from disk

```
    OfflineFirmwareUpdateHelper offlineFirmwareUpdateHelper = new OfflineFirmwareUpdateHelper();
    final FirmwareUpdate fwUpdateOnDisk =
          offlineFirmwareUpdateHelper.getNextFirmwareUpdate(
                currentDevice.getDeviceID(),
                currentSwVersion,
                yetiInformationActivity.getApplicationContext()
          );
```

Selectable update

If the device is connected to one of its components like the FTA(Disco), there may be an update for that too. It is possible to either update both the device's and the connected components' firmware or only one of them.

Example:
Assuming the FirmwareUpdate object is containing firmware update data for the device and for one(or more) components.
Update both device and components:

```
    if (updateRegion == UpdateRegion.device) {
          fwUpdateToUse.components = null;
    }else if (updateRegion == UpdateRegion.components) {
          fwUpdateToUse.binaries = null;
    }
          launchUpdate(
                currentDevice,
                fwUpdateToUse,
                yetiInformationActivity
          );
```

If either one of the options is selected then you can easily remove the binares from the FirmwareUpdate object

Live Image

Only Leica 3D devices are able to transfer a Live Image.

```
    currentDevice.connectLiveChannel(Device.LiveImageSpeed.FAST);
```

- Starts Live Image Transfer
- To retrieve the image data the app must listen on the ReceivedDataListener and extract the image data from the retrieved packages. For further example code, please look into the example app.

```
      // send any string to device
    currentDevice.disconnectLiveChannel();
```

- Stops Live Image Transfer

Further Documentation

- In Commands.Response package:
  Each Response class has the mapping to the corresponding commands it can handle.

- In Devices package:
  Each device class has the function getAvailableCommands – where you can find a list of supported commands by the Device.

- In Utilities Package, Defines.java:
  You can find the description of all the ID used through the application.

- In ErrorHandling package, ErrorObject.java:
  You can find a definition of the errors reported by the sdk, including the errors sent back by the DISTO Device

- In Types.java
  You can find all the available commands, parameters

- In -response classes
  The corresponding mapping between commands-Responses