NUR JAVA API: GETTING STARTED WITH ANDROID STUDIO 2

SIMPLE STRUCTURAL EXAMPLE TO SET UP AND TEST USB COMMUNICATION

VERSION C

SCOPE

The intent of this document is to instruct how a very basic communication setup is done with USB connection on a device that supports USB OTG and thus can communicate with a Nordic ID UHF reader devices including the evaluation board with USB connection and the STIX and Sampo readers.

This document does not describe basic Android application development issues such as creating controls and so on – importing and making use of the NUR-based readers using the NUR Java API is in the focus.

REQUIREMENTS

Things required here are

- Android Studio (version this instructional manual is based on is 2.1.2)
- Basic knowledge of Android application development
- Access to the Nordic ID software distribution for needed Java libraries and sources
- An OTG capable Android device, prefereably running "reasonably new" version of Android (4.2 or higher recommended)
- NUR module based reader that communicates using USB (note that in GitHub there are examples of BLE usage also)

NOTE ON READERS

Some Android devices, when acting as a USB host over OTG, may not be able to provide enough power to the USB device. This is not the case generally with Sampo readers or NUR evaluation kits as both of them have an external power source connected to them.

However, a STIX reader draws all of its power through the USB connection. To try out the device's capability to provide enough current, use the Nordic ID provided Android sample.

If the device is not able to provide enough current to the reader:

- the reader may go to "on-off-on-off..." loop i.e. the Android device may constantly cut off the power due to current surge and immediately turn the power on again
- the reader may appear OK, but when e.g. an inventory is started, the reader then draws enough power to cause the Android device's USB over current protection to trigger and thus cut off the power

There is no general rule how to determine whether the device will work OK with the STIX reader. Therefore always try to acquire the detailed technical specification of the Android device you are using in order to determine whether the USB host can provide enough current: 500mA is a recommended minimum and more than 500mA is better.

CREATING THE COMMUNICATION TEST

Start a new Android project. The base directory that is used here is

`"C:\AndroidTestProjects\"`

Project created there is "NURCommunicationTest" so the path will be

`"C:\AndroidTestProjects\NURCommunicationTest"`.

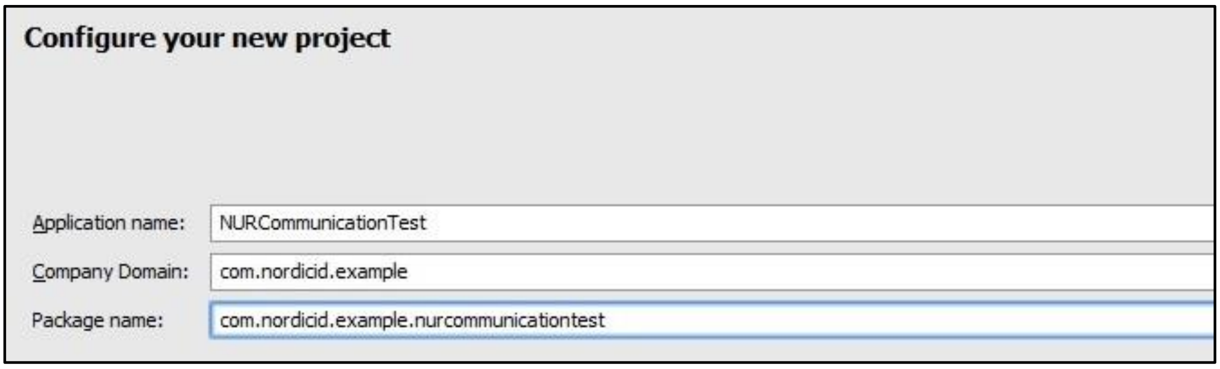
Change the company domain to reflect your own if needed. Here he domain is

`com.nordicid.example`

and the package name will be

`com.nordicid.example.nurcommunicationtest`

The view in the dialog:



Configure your new project

Application name:	NURCommunicationTest
Company Domain:	com.nordicid.example
Package name:	com.nordicid.example.nurcommunicationtest

In the next phase set the minimum SDK version as required. In this example the SDK version 4.2 (Jelly Bean) is selected; it might be older as well, but this is used for this example.

The target SDK will be set later so that the test application will actually run in versions older than 5.0 or 6.0 as well.

Next select the type of activity. Here it is simply an empty activity as we are only testing the communication interfaces and no special features are needed.

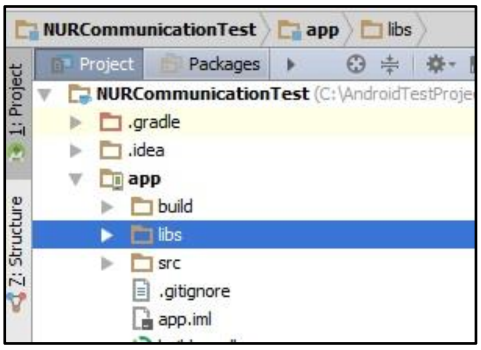
The activity's name and its layout name are left as is, for testing the functionality they are not relevant.

ADDING THE NUR API LIBRARY TO THE PROJECT

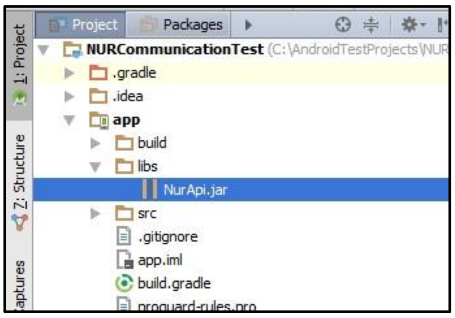
The NUR API is the protocol and communication handler for the application. The API is created within the application and it is then given a transport. The transport may be USB, TCP/IP, Bluetooth or something that the developer implemented. In this basic example we just create a simple USB autoconnection transport.

As the sources for the transport layer that actually moves the data between the API and the reader (or module), it is entirely possible to implement some other transport as well in a situation where the "other end's" device is something of an own design. Such a device might for example be a NUR module that is hooked into a small microcontroller handling Bluetooth communications.

In the left pane, select the view to be "Project". Under the "NURCommunicationTest", navigate to "app" -> "libs", right-click it and select "Show in Explorer".



In the explorer window open the "libs" folder and then open another explorer window and navigate to where the "NurApi.jar" is located (e.g. NUR distributions Java/common –folder) . Copy the JAR-file to the "libs" folder and go back to the Android Studio. Now you can see that the NurApi.jar appears in the "libs"folder:



Right click the JAR-file and select "Add As Library". In the following dialog just select OK. Now the actual API is imported into the project. Now add

```
import com.nordicid.nurapi.*;
```

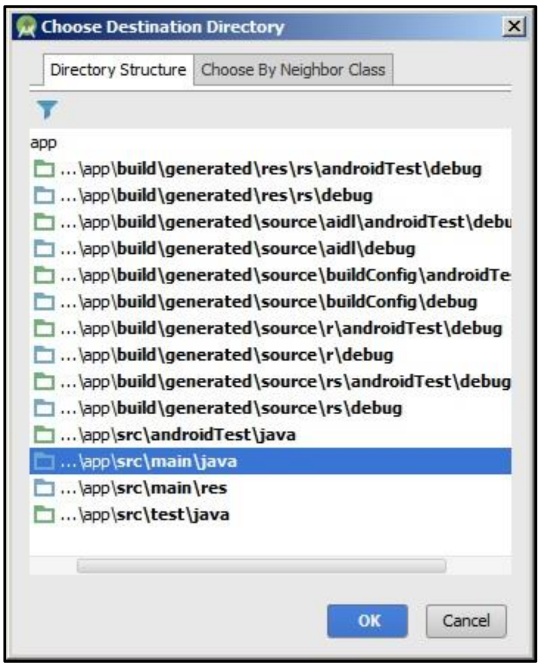
after other imports in the “MainActivity.java” and the API is ready to be used in the application.

ADDING THE TRANSPORT PACKAGE AND SOURCES

The transport layer is the part that actually carries the data to and from the module.

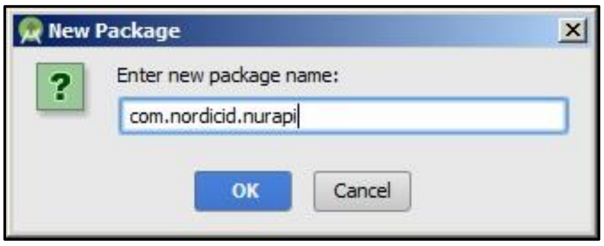
In this example we are about to import only the USB transport part as it works with older API versions as well. The Bluetooth and BLE extension are imported the same way, but they require a bit newer API version (4.3 for BLE itself and 4.4 for some parts of the implementation).

Change the view in the left pane to "Packages". Again right-click the "app" and navigate to New -> Package. In the directory structure select ..\app\src\main\java:


Choose Destination Directory

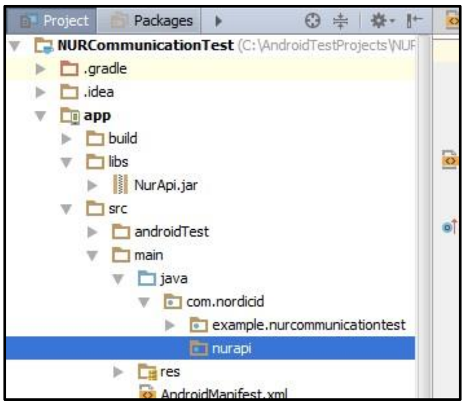
```
app
...\app\build\generated\res\rs\androidTest\debug
...\app\build\generated\res\rs\debug
...\app\build\generated\source\aidl\androidTest\debu
...\app\build\generated\source\aidl\debug
...\app\build\generated\source\buildConfig\androidTe
...\app\build\generated\source\buildConfig\debug
...\app\build\generated\source\r\androidTest\debug
...\app\build\generated\source\r\debug
...\app\build\generated\source\rs\androidTest\debug
...\app\build\generated\source\rs\debug
...\app\src\androidTest\java
...\app\src\main\java
...\app\src\main\res
...\app\src\test\java
```

For the package name (here) enter "com.nordicid.nurapi":


New Package

Enter new package name: com.nordicid.nurapi

Now in the left pane's project view navigate to the package location, right-click and open again in the explorer (Show in Explorer):



Open the "nurapi" and into this folder, copy the required transport sources for this project. The sources are found in the NUR distribution if you already didn't copy them into a separate folder for easier fetching.

The files needed for this project are:

- "NurApiAutoConnectTransport.java" : interface representing the automatic connection
- "NurApiUsbAutoConnect.java" : this implements one of the automatic connection interfaces (USB, here this is physically OTG)
- "NurApiUsbTransport.java" : this implements the actual "data carrier" for the USB transport layer.

At this point the project can be built by pressing Ctrl+F9 to see that there are no problems.

NUR API LISTENER INTERFACE’S METHODS

The NUR API's event listeners implement a mechanism that allows the application to receive notifications from the API asynchronously. Such events include for "connected", "disconnected", "inventory stream event", "tag trace event" and similar.

Describing these events are not in the scope of this documentation. To get familiar with them it is recommended to use the PC samples (using Eclipse + WindowBuilder) as a starting point. In those samples the various events are demonstrated in more detail.

As the NUR API is already imported, now we simply implement the NurApiListener interface directly to the MainActivity class by adding the "implements" like this:

`public class MainActivity extends AppCompatActivity` ***`implements NurApiListener`***

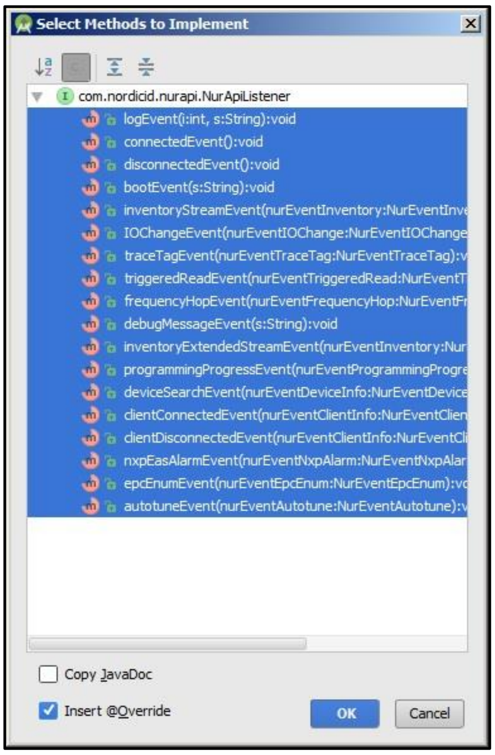
After this the Android Studio is not happy with the MainActivity class anymore; the NurApiListener interface's methods are missing.

Now you either

a) select the "NurApiListener" after the "implements" and press **`Ctrl+I`** (capitol ”i”) so that the Android Studio imlpements the missing method stubs
OR

b) go to the NUR distributions Java-folder, open the "NurApi_event_handler_stubs.java" and copy the method stub implementations from there.

Here we go with the option (a). The Android Studio suggests the interface's method names, accept with OK:



Select Methods to Implement

```
com.nordicid.nurapi.NurApiListener
    logEvent(i:int, s:String):void
    connectedEvent():void
    disconnectedEvent():void
    bootEvent(s:String):void
    inventoryStreamEvent(nurEventInventory:NurEventInve
    IOChangeEvent(nurEventIOChange:NurEventIOChange
    traceTagEvent(nurEventTraceTag:NurEventTraceTag):v
    triggeredReadEvent(nurEventTriggeredRead:NurEventT
    frequencyHopEvent(nurEventFrequencyHop:NurEventFr
    debugMessageEvent(s:String):void
    inventoryExtendedStreamEvent(nurEventInventory:Nur
    programmingProgressEvent(nurEventProgrammingProgre
    deviceSearchEvent(nurEventDeviceInfo:NurEventDevice
    clientConnectedEvent(nurEventClientInfo:NurEventClien
    clientDisconnectedEvent(nurEventClientInfo:NurEventCli
    nxpEasAlarmEvent(nurEventNxpAlarm:NurEventNxpAlar
    epcEnumEvent(nurEventEpcEnum:NurEventEpcEnum):vo
    autotuneEvent(nurEventAutotune:NurEventAutotune):v
```

☐ Copy JavaDoc
☑ Insert @Override

Now the Android Studio is happy again for the time being.

APPLICATION VARIABLES

Add these declarations to the MainActivity –class:

```
private NurApi mApi;
private NurApiAutoConnectTransport mAutoConnectTransport = null;
private boolean mListening = false;
private Button mControlBtn;
private TextView mStatus;
private TextView mInformation;
```

Also in the design side add the needed button and two text views and set their identifiers to

"`controlBtn`", "`statusText`" and "`infoText`"

respectively.

API PREPARATION METHOD

The last operation in the "onCreate" –method should be the "prepareAPI()" –method. The implementation is:

```
private void prepareAPI() {
  mApi = new NurApi();
  // Needed for UI access.
  mApi.setUiThreadRunner(new NurApiUiThreadRunner() {
    @Override
    public void runOnUiThread(Runnable runnable) {
      MainActivity.this.runOnUiThread(runnable);
    }
  });
  // "this" implements the listener:
  mApi.setListener(this);
}
```

THE APPLICATION START: ON CREATE

The “onCreate” –method’s contents in whole should look similar to this:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  mControlBtn = (Button)findViewById(R.id.controlBtn);
  mStatus = (TextView)findViewById(R.id.statusText);
  mInformation = (TextView)findViewById(R.id.infoText);

  mStatus.setText("Idle.");
  mInformation.setText("Reader information: N/ A");

  mControlBtn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
      mListening = !mListening;
      if (mListening) {
        startConnectionListening();
        mStatus.setText("Waiting for USB connection...");
        mControlBtn.setText("Stop listening");
      }
      else {
        stopConnectionListening();
        mStatus.setText("Idle");
        mControlBtn.setText("Start listening");
      }
    }
  });
  prepareAPI();
}
```

THE START AND STOP METHODS

As you can see, there were two methods that are called when the control button is pressed: starting and stopping the connection listening.

The start method contents:

```
private void startConnectionListening() {
  if (mAutoConnectTransport == null) {
    mAutoConnectTransport = new NurApiUsbAutoConnect(this, mApi);
    mAutoConnectTransport.setAddress("USB");
  }
}
```

The stop method contents:

```
private void stopConnectionListening() {
  if (mAutoConnectTransport != null)
  {
    mAutoConnectTransport.dispose();
    mAutoConnectTransport = null;
  }
}
```

THE API'S CONNECTION EVENT

The API's connection event here is quite simple and it the method that actually test that the communication is OK. Navigate to the "connectedEvent" –method and the implementation is:

```
@Override
public void connectedEvent() {
  mStatus.setText("Connected.");
  tryReaderInformation();
}
```

And the "tryReaderInformation" –method's imlpementation is:

```
private void tryReaderInformation() {
  try {
    NurRespReaderInfo ri;
    String strInfo = "";
    ri = mApi.getReaderInfo();
    strInfo = "Reader: " + ri.name + "\n" + "FW: " + ri.swVersion;
    mInformation.setText(strInfo);
  }
  catch (Exception e)
  {
    mInformation.setText("Reader information error: " +
       e.getMessage());
  }
}
```

THE API'S DISCONNECT EVENT

Navigate to the "disconnectedEvent" –method and implement it like this:

```
@Override
public void disconnectedEvent() {
  if (mListening)
    mStatus.setText("Waiting for USB connection...");
  else
    mStatus.setText("Idle.");
  mInformation.setText("Reader information: N/ A");
}
```

OTHER REQUIRED METHODS

The application needs these methods' implemented as the "NurApiAutoConnectTransport" –inteface's corresponding methods need to be called:

- onPause
- onResume
- onStop
- onDestroy

The pause, resume and stop methods are all alike:

```
@Override
protected void onPause() {
  super.onPause();
  if (mAutoConnectTransport != null)
    mAutoConnectTransport.onPause();
}

@Override
protected void onResume() {
  super.onResume();
  if (mAutoConnectTransport != null)
    mAutoConnectTransport.onResume();
}

@Override
protected void onStop() {
  super.onStop();
  if (mAutoConnectTransport != null)
    mAutoConnectTransport.onStop();
}
```

However, the "onDestroy" –method needs a bit more processing:

```
@Override
protected void onDestroy()
{
  super.onDestroy();
  if (mAutoConnectTransport != null)
    mAutoConnectTransport.onDestroy();
  if (mApi != null)
    mApi.dispose();
}
```

TRYING IT OUT

Best way to try out the test application would be to debug it over Wi-Fi. However, just starting the application from the Android Studio via USB and then simply removing the USB cable and replacing it with the OTG cable having the reader on the other end works as well.