TapLinx Application Note

Usage & Upgrading from MIFARE SDK to TapLinx

1. Introduction

The TapLinx library allows you to communicate with NFC devices on an Android system easily. The library encapsulates all low-level communication as well as all device proprietary dependencies and offers a homogeneous interface. This application note will help you to start with TapLinx and explains the steps which are required for integrating and using TapLinx with Android Studio. It will also explain how to register the library and how to start with your own app. In the last chapter you will find tips and tricks for typical use-cases for MIFARE products.

1.1 Where to find Code Snippets

All code snippets in this application note can be found in the attachments of this PDF file. Please open the *Attachments* tab and save the complete Java files as shown in Fig 1.

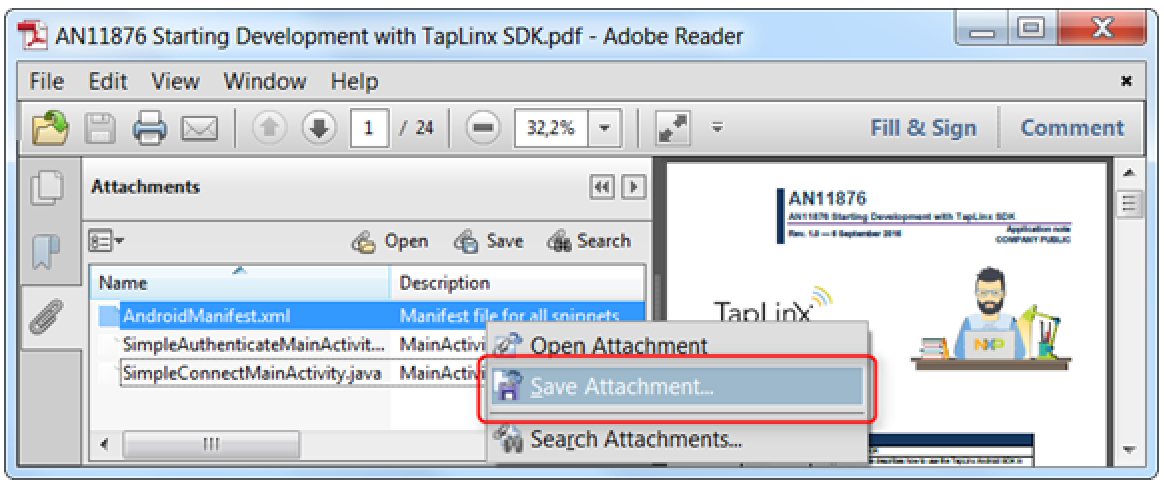

AN11876 Starting Development with TapLinx SDK.pdf - Adobe Reader
File Edit View Window Help
Fill & Sign
Comment
Attachments
Open
Save
Search
Name
Description
AndroidManifest.xml
Manifest file for all snippets
SimpleAuthenticateMainActivit...
MainActivi
SimpleConnectMainActivity.java
MainActivi
Open Attachment
Save Attachment...
Search Attachments...
AN11876
TapLinx

Fig 1. Code snippets of this application note

At the time of writing this application note, an online information portal is being created. You will find samples and code snippets for typical use-cases there.

2. The First Time Online Registration

When you login to the TapLinx server you can enter a package string of your app and you will receive a key string which you have to use in the source code of your software to enable the TapLinx library for unlimited use. The package key must be the same package key as your Manifest file. How to insert the key string is shown in Fig 6.

Register your app in order to enable the TapLinx library for your Android device. This must be done only once and is mandatory. After this registration the library is enabled and you can use the library within your app without any limitation in time or usage.

Please note that you will see a pop-up dialog which reminds you to connect your device to the internet for registering. You can skip this registration nine times, but the tenth reminder will set the TapLinx library in a deactivated state. This state can only be released if you connect your device to the internet and get a successful registration confirmation.

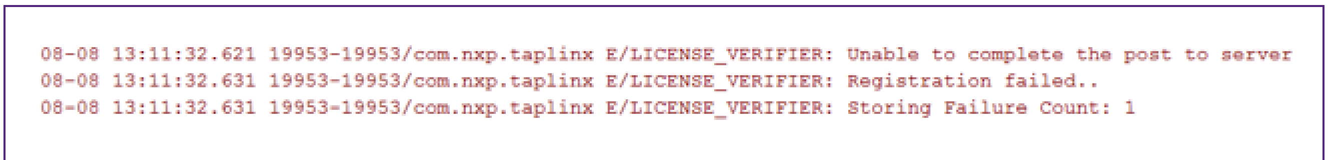
2.1 In Case of an Unsuccessful Registration Confirmation

If the registration was not successful, two things may have gone wrong:

1. The key string was not inserted properly in the source code. This key must be set to string for the second parameter of the method *NfcNxpLib.registerActivity()*. Please use the key string and do not add any other information like a version number etc. to it.

2. The package name you enter at the registration is different from the package name you used in your project. Please use the package key only for the registered project. If you want to start another Android project and you want to use the TapLinx library, please register the package name of this project. You will get a new package key for using with the new project.

You will see a message that notifies you in the LogCat output (see Fig 2). If no network access is available, the failure counter is stalled. But if network access is available, you have nine more attempts to get a successful registering response. The library works as expected in this time. But from the 11th attempt onwards the library will be deactivated and no card communication is possible. An exception message is shown in this case.



```
08-08 13:11:32.621 19953-19953/com.nxp.taplinx E/LICENSE_VERIFIER: Unable to complete the post to server
08-08 13:11:32.631 19953-19953/com.nxp.taplinx E/LICENSE_VERIFIER: Registration failed..
08-08 13:11:32.631 19953-19953/com.nxp.taplinx E/LICENSE_VERIFIER: Storing Failure Count: 1
```

Fig 2. Notification of unregistered library usage

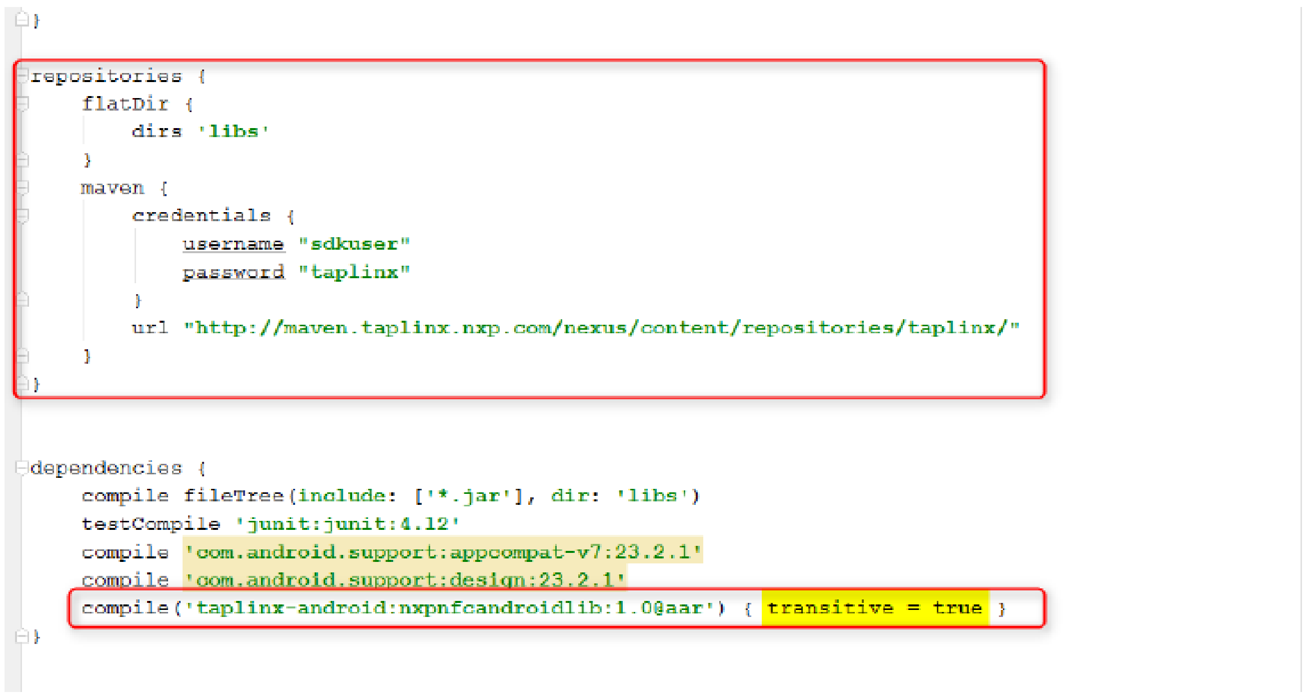
3. Setup TapLinx and the Required Libraries

This chapter explains which preparation have to be done to use the TapLinx library with Android Studio. It shows the dependencies and explains which third-party components have to be included.

3.1 Obtain TapLinx from a Maven Repository

You obtain the TapLinx library from the NXP Maven repository. The TapLinx library depends on Google Analytics. Either you declare a separate dependency to Google Analytics or you set the property “{transitive=true}” at the end of the compile statement in the Gradle build script (marked yellow in Fig 3). The property “transitive=true” let Gradle evaluate the dependencies and load the required libraries.

The required changes in your build script *build.gradle* are red framed in Fig 3. These sections have to be inserted in your build script.



```
}

repositories {
    flatDir {
        dirs 'libs'
    }
    maven {
        credentials {
            username "sdkuser"
            password "taplinx"
        }
        url "http://maven.taplinx.nxp.com/nexus/content/repositories/taplinx/"
    }
}


dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:23.2.1'
    compile 'com.android.support:design:23.2.1'
    compile('taplinx-android:nxpnfcandroidlib:1.0@aar') { transitive = true }
}
```

Fig 3. Gradle script 'build.gradle' for accessing the TapLinx Maven repository

You will find a build script *build.gradle* with the settings from Fig 3 in the attachments.

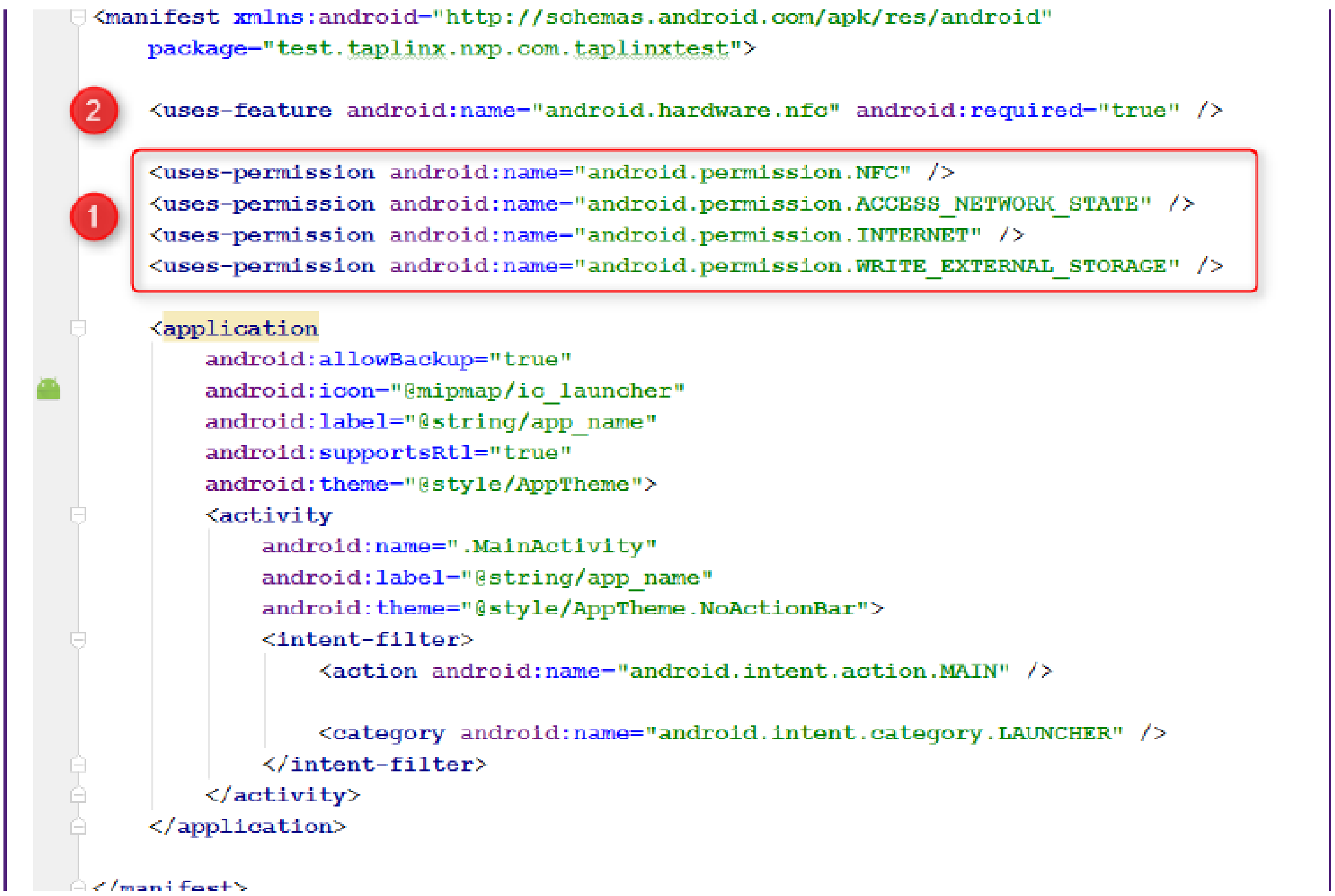
4. How to Start Programming with the TapLinx Library

The project must be prepared for use with the TapLinx library. In this chapter the preparation of the project source files are shown.

The manifest file haves to be prepared first.

1. The custom app must have the permissions of NFC, in order for network state and Internet access. The TapLinx library checks the registration key on a NXP server with the given package name. Therefore the OS will allow network access. The permission for external storage (the device "/sdcard") is used for log messages and dump files. See (1) below.
2. The "use-feature" is not required, but recommended to prevent unintentional user actions on a device which does not supports NFC. See (2) below.

Figure 4 below

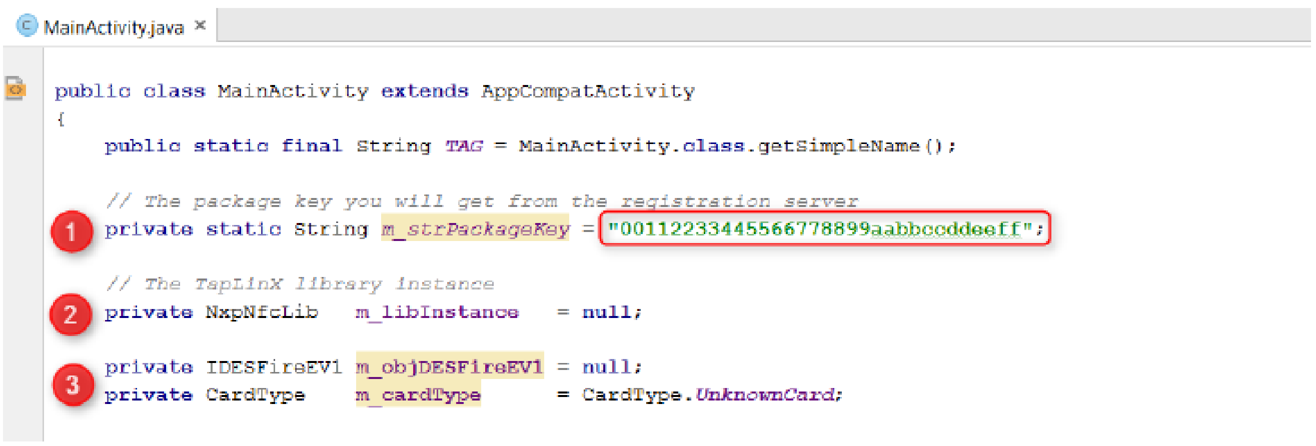

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="test.taplinx.nxp.com.taplinxtest">

    <uses-feature android:name="android.hardware.nfc" android:required="true" />

    <uses-permission android:name="android.permission.NFC" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

You will find the *Manifest.xml* file in the attachments.

The next step is to prepare the *MainActivity.java* (it is the Java file which contains the MainActivity class) as shown in Figure 5 below.

1. The definition of the package key. This string contains the license key you will receive after the registration. It is used in the method *registerActivity()* of the library instance. Please note that the key string marked in red in Fig 5 “0011…” is not a valid key! It is used here only for demonstration purposes.
2. Defining the instance of the library itself. This instance is used for accessing all methods and interfaces of the TapLinx library.
3. Here also are instances which contains a DESFire EV1 object and a type of card is also defined. This is not required and used here only for demonstration purposes.

The next step the library instance has to be registered within the activity.

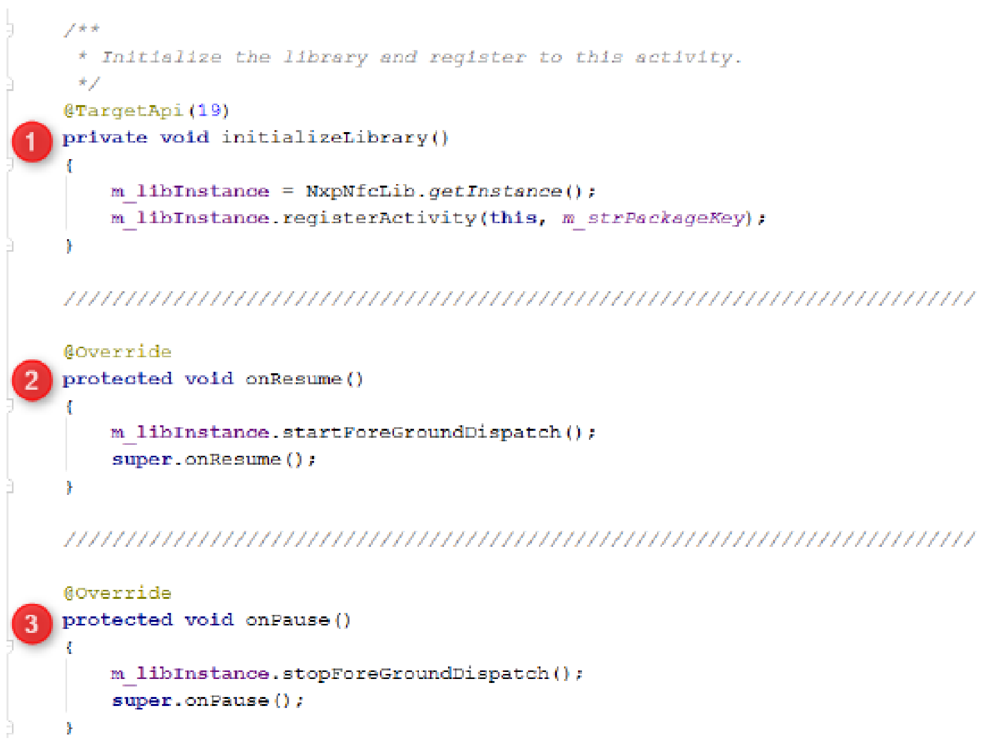


```java
public class MainActivity extends AppCompatActivity
{
    public static final String TAG = MainActivity.class.getSimpleName();

    // The package key you will get from the registration server
    private static String m_strPackageKey = "00112233445566778899aabbccddeeff";

    // The TapLinX library instance
    private NxpNfcLib   m_libInstance   = null;

    private IDESFireEV1 m_objDESFireEV1 = null;
    private CardType    m_cardType      = CardType.UnknownCard;
```

Fig 5. Variable definitions in the MainActivity class source file

After defining the variables the library has to be instantiated. This step is shown in Fig 6.

1. The user defined method *initializeLibrary()* shows how the TapLinx library has to be instantiated. The second parameter of *registerActivity()* must be the package key.
2. The “Android NFC using styles” is recommend to activate NFC intents for your app only, if your app has become the user focus and is visible in the foreground. Therefore dispatching NFC intents are activated in *onResume()*.
3. It is recommended to deactivate NFC intents for your app when another app climbs the foreground. Therefore dispatching NFC intents are deactivated in *onPause()*.



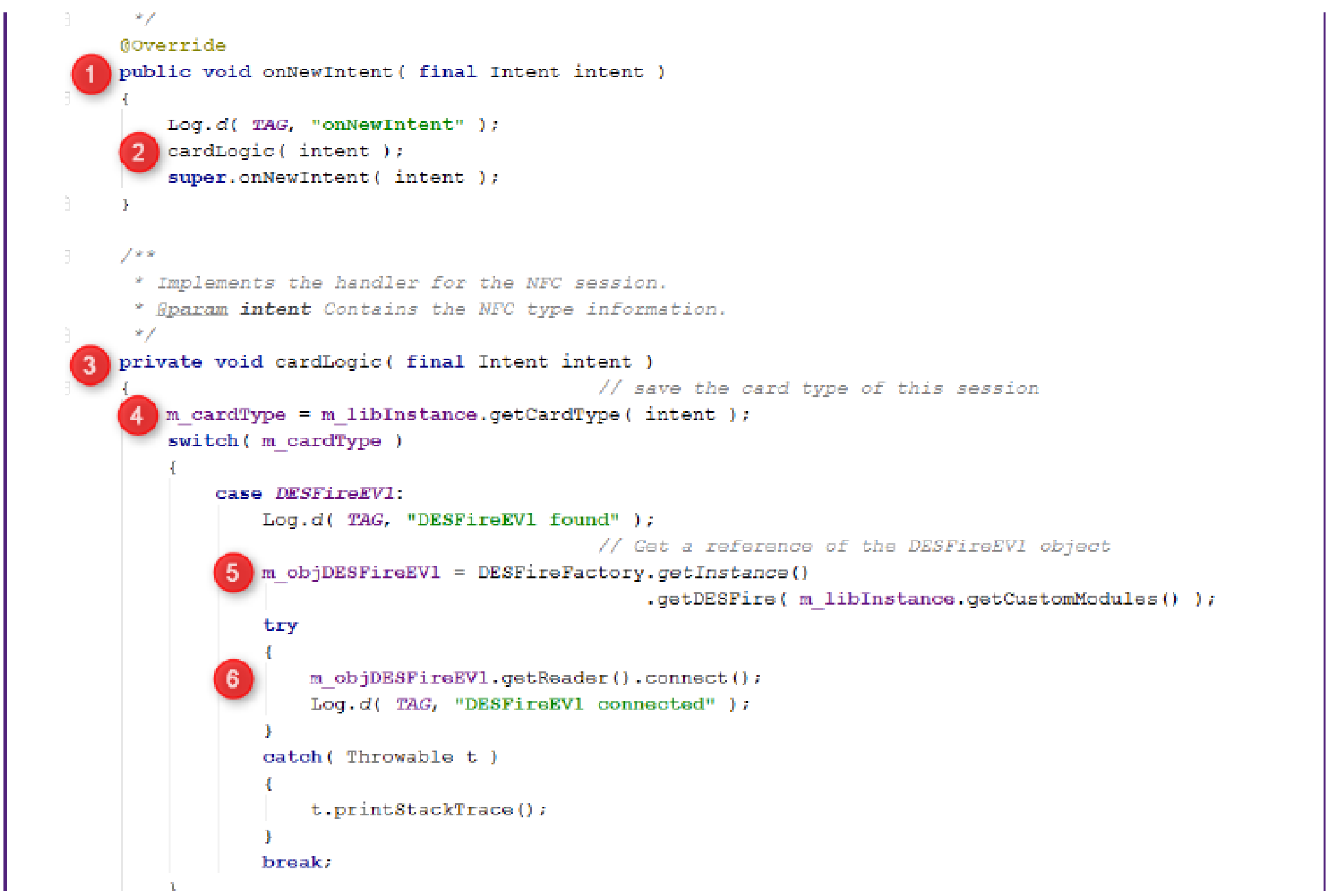
```
/**
 * Initialize the library and register to this activity.
 */
@TargetApi(19)
private void initializeLibrary()
{
    m_libInstance = NxpNfcLib.getInstance();
    m_libInstance.registerActivity(this, m_strPackageKey);
}

////////////////////////////////////////////////////////////////////////////

@Override
protected void onResume()
{
    m_libInstance.startForeGroundDispatch();
    super.onResume();
}

////////////////////////////////////////////////////////////////////////////

@Override
protected void onPause()
{
    m_libInstance.stopForeGroundDispatch();
    super.onPause();
}
```

Fig 6. Make NFC Intents available for the app

The last step is to overwrite the onNewIntent() method of the Activity class. This method is called if your app receives an intent from the system or other sources. Fig 7 shows the implementation.

1. *onNewIntent()* have to be overwritten. Please do not forget to call the base class method *onNewIntent()*.
2. All application specific code is handled in *cardLogic()*.We will focus later only to the implementation of this method. No other handlers need to be implemented.
3. The user defined method *cardLogic()* implements the custom code for interact with the tags and smartcards. We will focus on this method with later examples and code snippets in this application note.
4. The type of card is saved in the class variable (see the declaration in Fig 6 (3)).
5. The object instance is saved in the class variable (see the declaration in Fig 6 (3)). This is the card object which is used for the communication with the PICC.
6. Before you can start the communication with the card (we will see it later in the other snippets), you have to call *connect()* to establish the card object properly.

Figure 7 below



```
     */
    @Override
    public void onNewIntent( final Intent intent )
    {
        Log.d( TAG, "onNewIntent" );
        cardLogic( intent );
        super.onNewIntent( intent );
    }

    /**
     * Implements the handler for the NFC session.
     * @param intent Contains the NFC type information.
     */
    private void cardLogic( final Intent intent )
    {                                       // save the card type of this session
        m_cardType = m_libInstance.getCardType( intent );
        switch( m_cardType )
        {
            case DESFireEV1:
                Log.d( TAG, "DESFireEV1 found" );
                                            // Get a reference of the DESFireEV1 object
                m_objDESFireEV1 = DESFireFactory.getInstance()
                                                .getDESFire( m_libInstance.getCustomModules() );
                try
                {
                    m_objDESFireEV1.getReader().connect();
                    Log.d( TAG, "DESFireEV1 connected" );
                }
                catch( Throwable t )
                {
                    t.printStackTrace();
                }
                break;
```

A word of what is happening in this code snippet. You will see the first commands of a typical sequence (we will see in other snippets the same prologue). If you know from the card type enumerator that a DESFire EV1 is detected, you have to retrieve the proper card object from the library—in our case a DESFire EV1 object. Before you can start to refer to this object, you have to call connect() to establish the content properly.

You will find the Java file *SimpleConnectMainActivity.java* which implements the snippet from Fig 7 in the attachments.

4.1 Setup Spongy Castle Libraries for Using in the Sample App

Spongy Castle is used for storing the keys of the sample app. You can use Spongy Castle also for the encryption and decryption of data blocks. This subchapter explains how to prepare your app to use the Spongy Castle libraries.

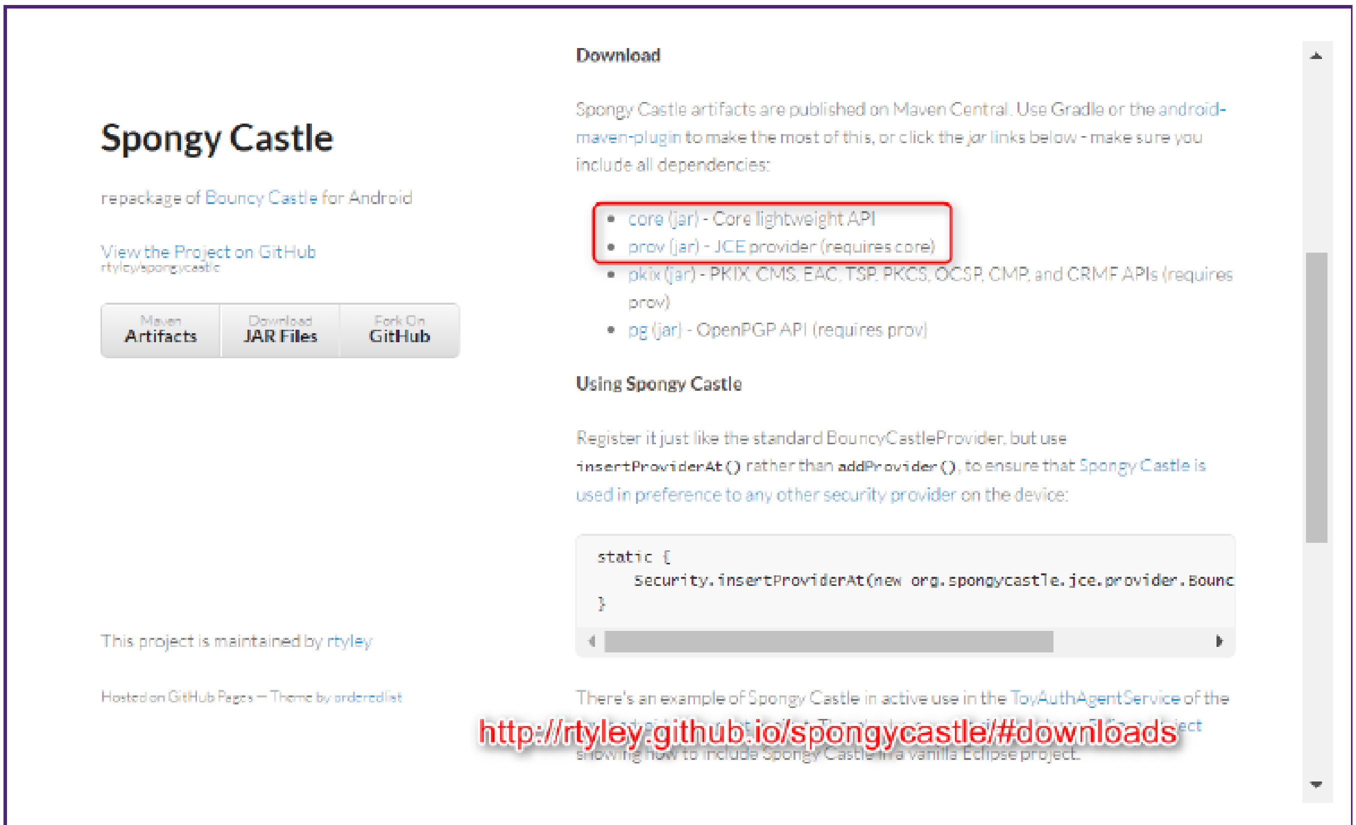
> The Spongy Castle library is not a must! You can use it, but you can decide also to use a different cryptographic provider or even no provider. Section 4.2 shows how to authenticate to a DESFire EV1 without Spongy Castle libraries.

To include Spongy Castle into your project, there are two alternatives. First, download the JAR files directly from the site and include it into your project. The libraries are available from this URL:

https://rtyley.github.io/spongycastle/

For basic cryptography and using only the keystore, the JAR files *core-1.54.0.0,jar* and *prov-1.54.0.0.jar* (see Fig 8) are required.

The second approach is to import the library reference to the Gradle build system and let Gradle download and install the libraries.



Spongy Castle

repackage of Bouncy Castle for Android

View the Project on GitHub
rtyley/spongycastle

Maven Artifacts | Download JAR Files | Fork On GitHub

This project is maintained by rtyley

Hosted on GitHub Pages — Theme by orderedlist

Download

Spongy Castle artifacts are published on Maven Central. Use Gradle or the android-maven-plugin to make the most of this, or click the jar links below - make sure you include all dependencies:

- core (jar) - Core lightweight API
- prov (jar) - JCE provider (requires core)
- pkix (jar) - PKIX CMS, EAC, TSP, PKCS, OCSP, CMP, and CRMF APIs (requires prov)
- pg (jar) - OpenPGP API (requires prov)

Using Spongy Castle

Register it just like the standard BouncyCastleProvider, but use `insertProviderAt()` rather than `addProvider()`, to ensure that Spongy Castle is used in preference to any other security provider on the device:

```
static {
    Security.insertProviderAt(new org.spongycastle.jce.provider.Bounc
}
```

There's an example of Spongy Castle in active use in the ToyAuthAgentService of the

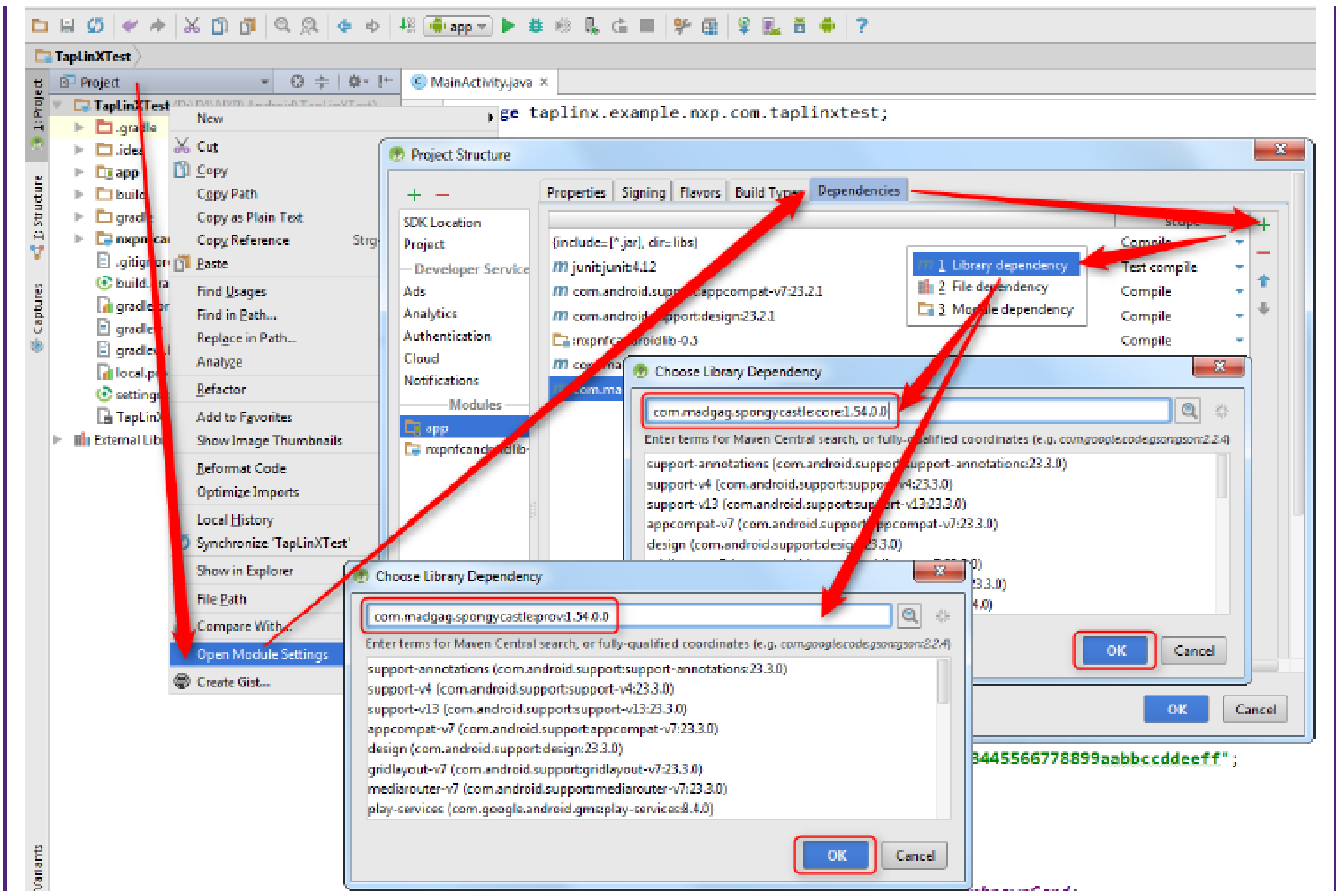
http://rtyley.github.io/spongycastle/#downloads

showing how to include Spongy Castle in a vanilla Eclipse project.

Fig 8. Download of Spongy Castle Libraries

We will show the latter approach to include the library reference into the Gradle build system of Android Studio. To manage this, make a right click onto the *Project* and select *Open Module Settings*. Select the *Dependencies* tab and click "+" and *Library dependencies* two times (see Fig 9 below).

Figure 9 below



```
ge taplinx.example.nxp.com.taplinxtest;
```

	Scope
{include=[*.jar], dir=libs}	Compile
junit:junit:4.12	Test compile
com.android.support:appcompat-v7:23.2.1	Compile
com.android.support:design:23.2.1	Compile
:nxpnfcandroidlib-0.3	Compile

Choose Library Dependency: com.madgag.spongycastle:core:1.54.0.0

Choose Library Dependency: com.madgag.spongycastle:prov:1.54.0.0

Enter terms for Maven Central search, or fully-qualified coordinates (e.g. com.google.code.gson:gson:2.2.4)

```
support-annotations (com.android.support:support-annotations:23.3.0)
support-v4 (com.android.support:support-v4:23.3.0)
support-v13 (com.android.support:support-v13:23.3.0)
appcompat-v7 (com.android.support:appcompat-v7:23.3.0)
design (com.android.support:design:23.3.0)
gridlayout-v7 (com.android.support:gridlayout-v7:23.3.0)
mediarouter-v7 (com.android.support:mediarouter-v7:23.3.0)
play-services (com.google.android.gms:play-services:8.4.0)
```

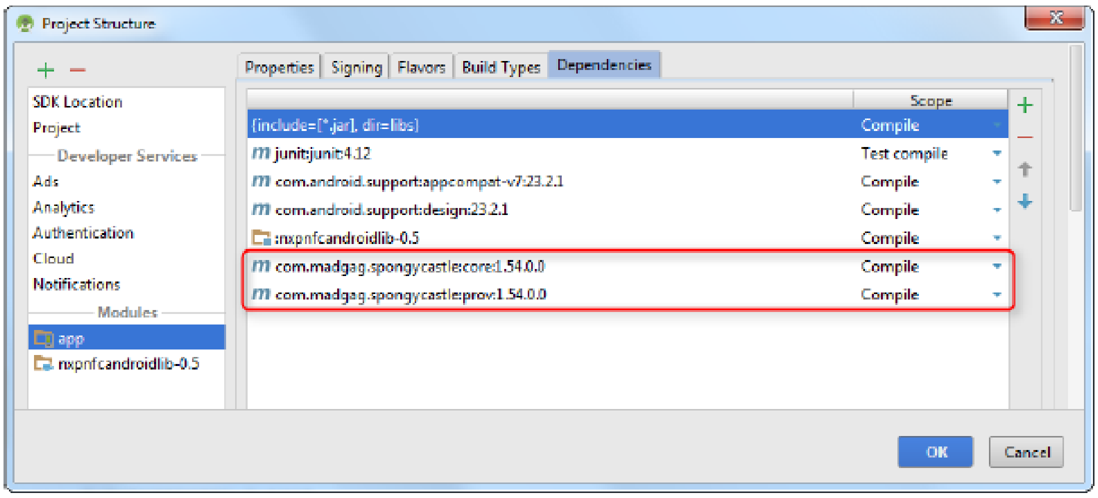
```
3445566778899aabbccddeeff";
```

Add the library dependencies of both libraries. At the time of writing this application note, version 1.54.0.0 is the latest release.

Table 1: Spongy Castle Maven Repository

Library Type	Maven Repository URL
Core	com.madgag.spongycastle:core:1.54.0.0
Provider	com.madgag.spongycastle:prov:1.54.0.0

At the end you should see the both libraries appended to the dependency list of your project (see Fig 10).



	Scope
{include=[*.jar], dir=libs}	Compile
junit:junit:4.12	Test compile
com.android.support:appcompat-v7:23.2.1	Compile
com.android.support:design:23.2.1	Compile
:nxpnfcandroidlib-0.5	Compile
com.madgag.spongycastle:core:1.54.0.0	Compile
com.madgag.spongycastle:prov:1.54.0.0	Compile

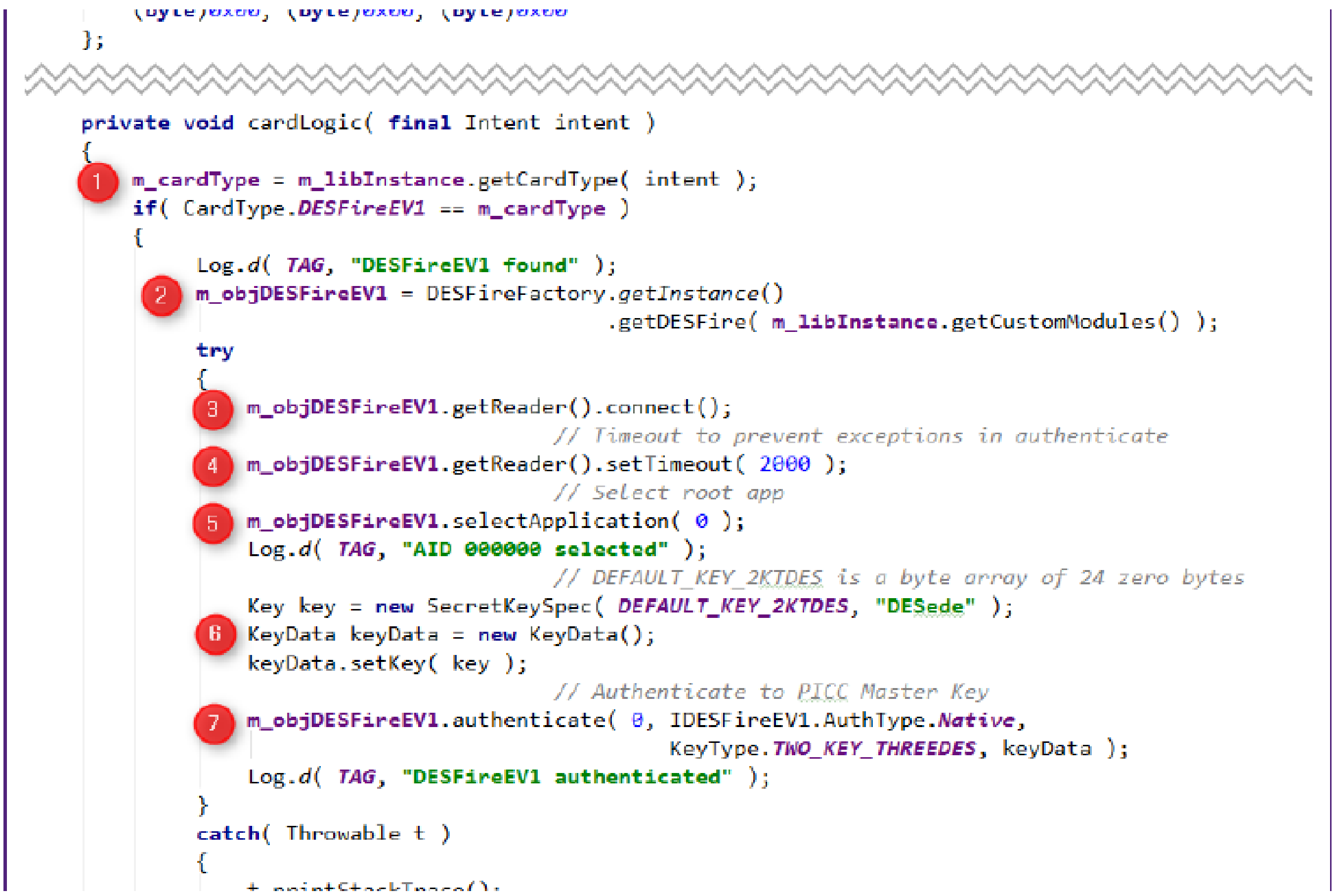
Fig 10. Added Spongy Castle Library Dependencies

4.2 Authentication to a DESFire EV1 without the Spongy Castle Keystore

This section shows an authentication to a DESFire EV1 without using the Spongy Castle keystore. For instance, if the high security aspect in an app is less important, an authentication can be achieved with the approach from code snippet Fig 11.

The code snippet is verified with a blank DESFire EV1. The default setting is the cipher “Two Key Triple DES” (2K3DES) and a default key with all zeros (00…00). The default key has to be defined as byte array of 24 zero bytes (not visible in the code snippet). The snippet implements the same basic coding scheme as used in the introduction in Fig 8.

Figure 11 below



```
    (byte)0x00, (byte)0x00, (byte)0x00
};

private void cardLogic( final Intent intent )
{
1   m_cardType = m_libInstance.getCardType( intent );
    if( CardType.DESFireEV1 == m_cardType )
    {
        Log.d( TAG, "DESFireEV1 found" );
2       m_objDESFireEV1 = DESFireFactory.getInstance()
                                        .getDESFire( m_libInstance.getCustomModules() );
        try
        {
3           m_objDESFireEV1.getReader().connect();
                                // Timeout to prevent exceptions in authenticate
4           m_objDESFireEV1.getReader().setTimeout( 2000 );
                                // Select root app
5           m_objDESFireEV1.selectApplication( 0 );
            Log.d( TAG, "AID 000000 selected" );
                                // DEFAULT_KEY_2KTDES is a byte array of 24 zero bytes
            Key key = new SecretKeySpec( DEFAULT_KEY_2KTDES, "DESede" );
6           KeyData keyData = new KeyData();
            keyData.setKey( key );
                                // Authenticate to PICC Master Key
7           m_objDESFireEV1.authenticate( 0, IDESFireEV1.AuthType.Native,
                                          KeyType.TWO_KEY_THREEDES, keyData );
            Log.d( TAG, "DESFireEV1 authenticated" );
        }
        catch( Throwable t )
        {
            t.printStackTrace();
```

1. Check the card type and save it.
2. Retrieve a DESFire EV1 object from the TapLinx library. You will use it for all card related methods.
3. Start the communication with the card object with *connect()*.
4. It might be required to add a timeout to prevent a failure while the authentication takes place. This depends on the phone type and Android version. You can omit this call.
5. Select the application AID = 000000 to authenticate to the PICC Master Key. Any AID other than 000000 will redirect the authentication to the Application Master Key.
6. Create a temporary Java Key object and initialize it with the key value from the byte array. The key settings defines a 2K3DES key which is the default for the DESFire EV1.
7. The *authenticate()* method uses the temporary key object, card key 0 and the authentication type "Native".

If no exception occurs, the card is now authenticated to the TapLinx library.

A word of what is happening in this code snippet. The MIFARE DESFire EV1 is a security product and you usually have to authenticate for changing the content. This is required if a blank card is personalized for the first time. Of course, a DESFire EV1 can be also configured to allow reading and writing without any authentication before, but this is not the default use-case for a security product.

The first card command is to select the application with the AID = 0 (application identifier). To select this AID, the follow authenticate command uses the PICC Master Key. The AID = 0 is always available, even on a blank card. But you cannot create files inside of this application. It is used only for card management.

The code snippet creates a Java key object on the fly and initialize it with a byte array DEFAULT_KEY_2KTDES defined in the MainActivity class. The byte array contains 24 zero bytes.

The *authenticate()* method verify that the Android device have the permission to manage the card content. We will use this command many times in the follow code snippets.

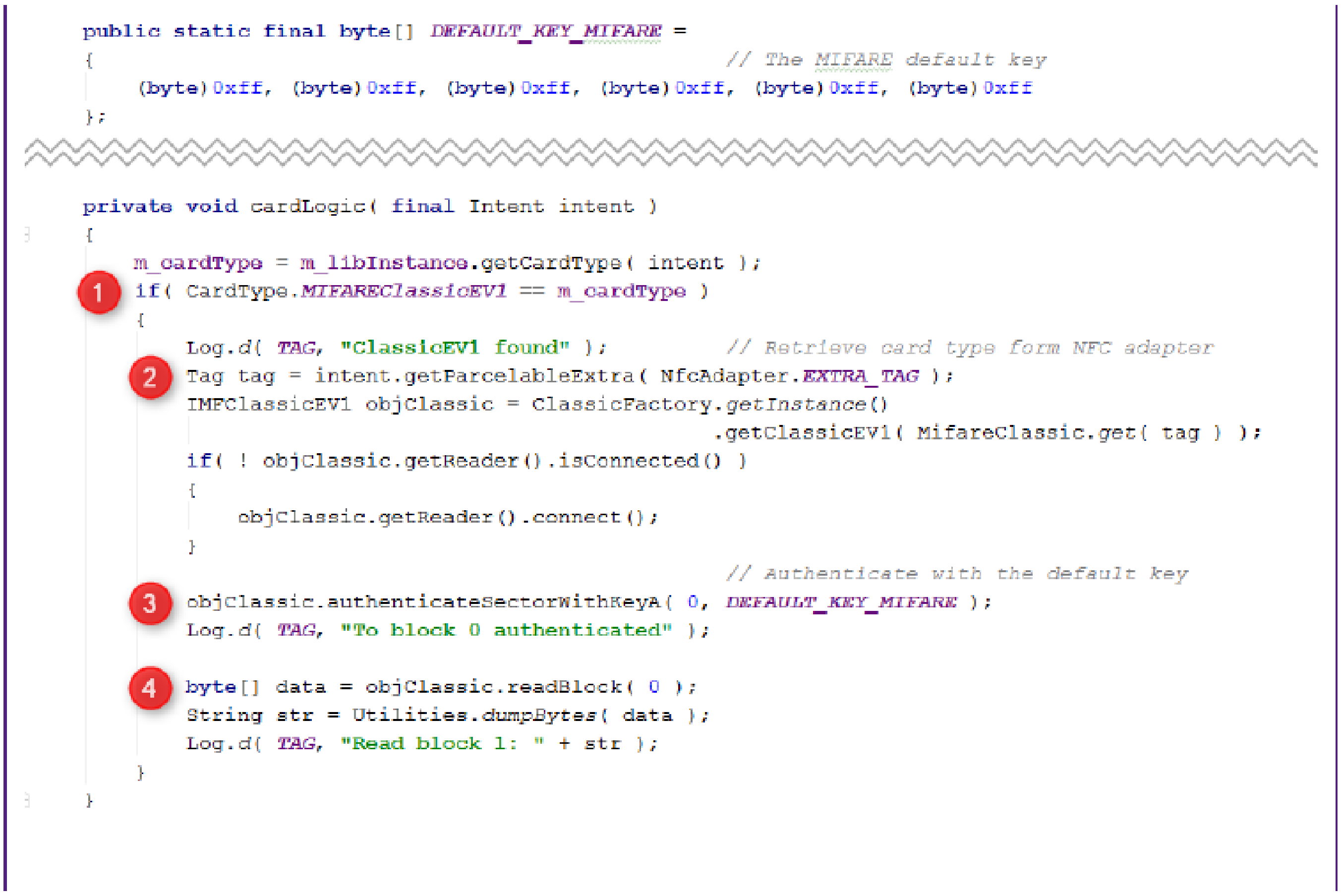
You will find the Java file *DESFireAuthenticateMainActivity.java* in the attachments.

4.3 Authentication to a Classic EV1 with the Android NFC API

The MIFARE Classic and MIFARE Plus uses a proprietary protocol and cipher. This is available as part of the Android NFC API. The code snippet in this chapter shows a simple authentication and a reading from the first block of a MIFARE Classic EV1.

Only the Android NFC API is used, so no references to other libraries are required. The code snippet in Fig 12 shows only the handler method *cardLogic()*, the other parts are not changed.

Figure 12 below



```
public static final byte[] DEFAULT_KEY_MIFARE =
{                                                       // The MIFARE default key
    (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff
};

private void cardLogic( final Intent intent )
{
    m_cardType = m_libInstance.getCardType( intent );
    if( CardType.MIFAREClassicEV1 == m_cardType )
    {
        Log.d( TAG, "ClassicEV1 found" );               // Retrieve card type form NFC adapter
        Tag tag = intent.getParcelableExtra( NfcAdapter.EXTRA_TAG );
        IMFClassicEV1 objClassic = ClassicFactory.getInstance()
                                        .getClassicEV1( MifareClassic.get( tag ) );
        if( ! objClassic.getReader().isConnected() )
        {
            objClassic.getReader().connect();
        }
                                                        // Authenticate with the default key
        objClassic.authenticateSectorWithKeyA( 0, DEFAULT_KEY_MIFARE );
        Log.d( TAG, "To block 0 authenticated" );

        byte[] data = objClassic.readBlock( 0 );
        String str = Utilities.dumpBytes( data );
        Log.d( TAG, "Read block 1: " + str );
    }
}
```

The key for the authentication is defined as byte array inside of the MainActivity class (DEFAULT_KEY_MIFARE). In this case it is the default key FFFFFF for a blank card.

1. First, the card type is saved and checked. In this example a Classic EV1 is used as black card for verifying.
2. For using the Android NFC API, the tag type has to be saved, because it is needed as parameter for the library method *getInstance()*. This differs from the TapLinx method shown in Fig 11.
3. Before you can access any sector of a block, you have to authenticate to the destination block with the key A or B. For the blank card the default key A is used.
4. To verify the access to the first block, the sector is read and printed out to the LogCat console.

Please take in mind that the authentication is required for every block you want to read or write.

You will find the Java file *ClassicAuthenticateMainActivity.java* in the attachments.

5. Coming from MIFARE SDK (Advanced/Lite)

If you have already experience with the MIFARE SDK (Advanced or Lite), you can upgrade to TapLinx, but you have to change your source files. The changes are moderate, most of the classes keep their names and the meaning. We will show the main differences in this chapter.

5.1 No built-in keystore in TapLinx

The Mifare SDK Advanced provides an own keystore implementation which allows also to use a MIFARE SAM AV2 as hardware keystore. The SDK Lite uses a byte array (byte[]) as parameter type for the plain key values in all methods which needs a key. From the security point of view, using plain values as keys in an app is a potential risk.

TapLinx uses the Java key classes to contain key values and allows it to use any available third-party keystore in your app. The sample app uses Bouncy Castle as keystore, but any other or even no keystore can be used with TapLinx. The section 4.1 shows how to integrate Spongy Castle into your project and 4.2 shows how to authenticate with an on-the-fly generated key.

5.2 Modifications in the Android Intent Calling Mechanism

In the MIFARE SDK you have to define the callback handler inside of the *MainActivity* class. A library filter method is called and the callback handler is used to detect and handle card requests. This callback handler is now implemented as part of the TapLinx library and only a method must be provided to get notified with NFC intents as shown in Fig 8.

Coming from the Advanced or Lite SDK, the user implementation in *MainActivity* class is tidier and better maintainable now. Only one overwritten method and one user defined method have to be used for the own implementation.

6. Adding the Library Documentation Files to Android Studio

Android Studio allows it to add documentation files into your project. As shown in Fig 13, the documentation is added by selecting the “Project” (1), selecting the depending library (2) and with the right mouse button click on (2) in the popup menu the item “Library Properties…” (3).

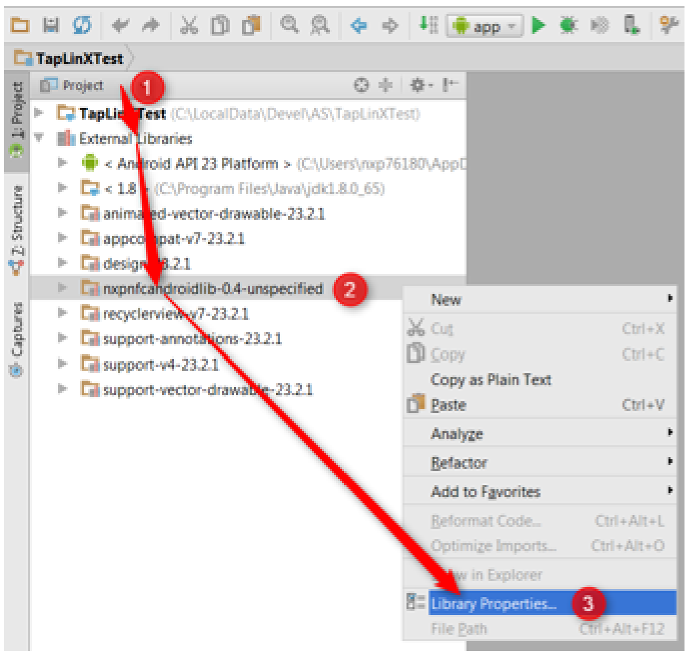

app
TapLinXTest
Project
1
TapLinXTest (C:\LocalData\Devel\AS\TapLinXTest)
External Libraries
< Android API 23 Platform > (C:\Users\nxp76180\AppD
< 1.8 > (C:\Program Files\Java\jdk1.8.0_65)
animated-vector-drawable-23.2.1
appcompat-v7-23.2.1
design-23.2.1
nxpnfcandroidlib-0.4-unspecified
2
recyclerview-v7-23.2.1
support-annotations-23.2.1
support-v4-23.2.1
support-vector-drawable-23.2.1
1: Project
2: Structure
Captures
New
Cut Ctrl+X
Copy Ctrl+C
Copy as Plain Text
Paste Ctrl+V
Analyze
Refactor
Add to Favorites
Reformat Code... Ctrl+Alt+L
Optimize Imports... Ctrl+Alt+O
Show in Explorer
Library Properties...
3
File Path Ctrl+Alt+F12

Fig 13. Adding documentation to the project

The next step depends on using local stored JavaDoc files or using a link to a global web server which contains the JavaDoc.

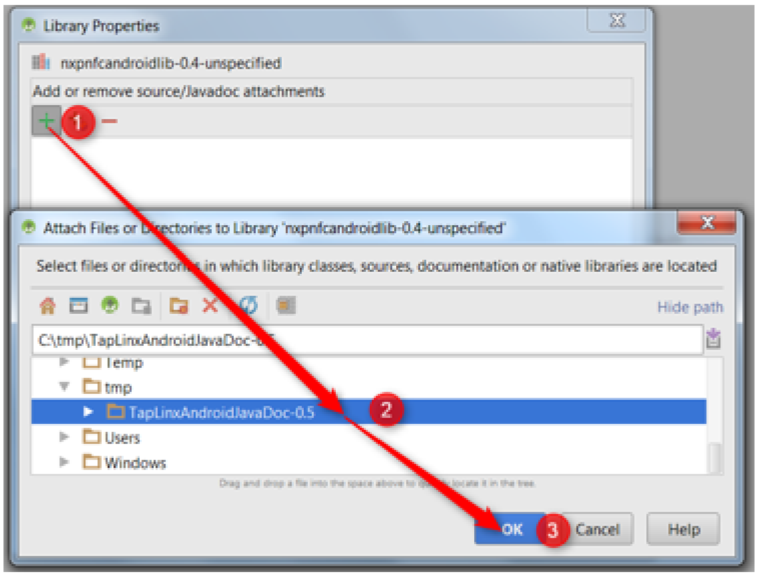
6.1 Add local JacDoc files into your project

In case of local JavaDoc files you have to unpack the ZIP archive and save the folder with the files at any accessible location on your PC.

The Library Properties Dialog, as shown in the upper screenshot of Fig 14, allows to include a local resource with the leftmost “plus” button (1) and selecting the directory location in the following dialog (2) and confirming it with “OK” (3).

It is required to confirm the type of resource added. In case of JavaDoc resources select “JavaDoc” (1) and confirm with “OK” (2) as shown in the lower screenshot of Fig 14.

Figure 14 below

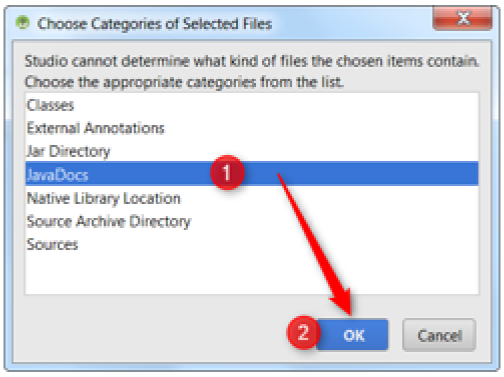

Library Properties
nxpnfcandroidlib-0.4-unspecified
1
Attach Files or Directories to Library 'nxpnfcandroidlib-0.4-unspecified'
Hide path
Temp
tmp
TapLinxAndroidJavaDoc-0.5
2
Users
Windows
OK
3
Cancel
Help

Add or remove source/Javadoc attachments

Select files or directories in which library classes, sources, documentation or native libraries are located

C:\tmp\TapLinxAndroidJavaDoc-0.5

Drag and drop a file into the space above to quickly locate it in the tree.



Studio cannot determine what kind of files the chosen items contain.
Choose the appropriate categories from the list.

- Classes
- External Annotations
- Jar Directory
- JavaDocs
- Native Library Location
- Source Archive Directory
- Sources

6.2 Add an external documentation resource to your project.

In the case of a global documentation resource a different button has to be clicked in the Library Properties Dialog. Select this type of resource in the Library Properties Dialog, as shown in the upper screenshot of Fig 15, click the middle “plus” button (1) and enter the URL of the external resource (2) and confirm with “OK” (3). As shown in the lower screenshot of Fig 15, the external resource is added to your project.

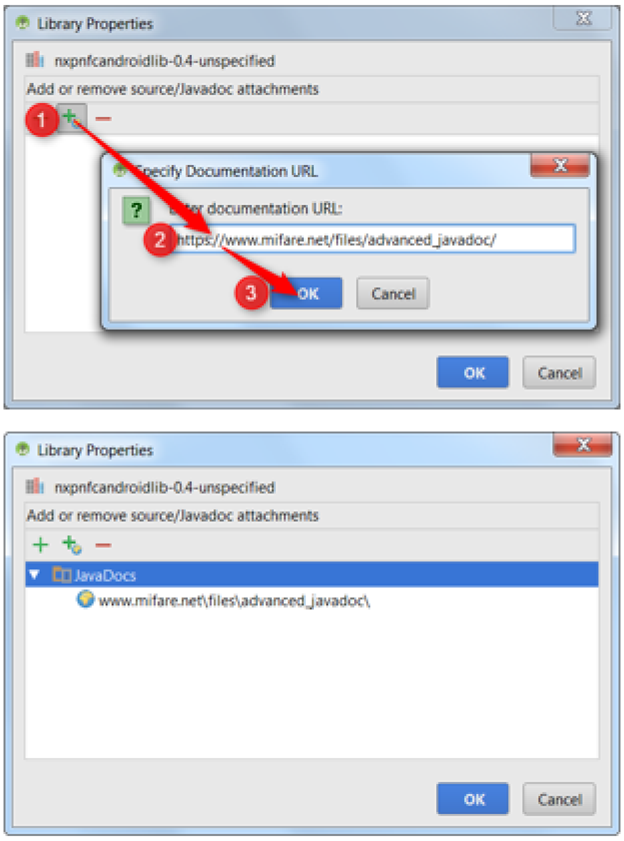

Library Properties
nxpnfcandroidlib-0.4-unspecified
Add or remove source/Javadoc attachments
1
Specify Documentation URL
Enter documentation URL:
2
https://www.mifare.net/files/advanced_javadoc/
3
OK
Cancel
OK
Cancel
Library Properties
nxpnfcandroidlib-0.4-unspecified
Add or remove source/Javadoc attachments
JavaDocs
www.mifare.net\files\advanced_javadoc\
OK
Cancel

Fig 15. Select an external repository as documentation resource

7. FAQ / Tips and Tricks

7.1 An exception occurs while accessing to the PICC

This may has several causes. But if you find in the LogCat output the following string:
Incomplete response received from PICC
then the current method takes too long time before a response can be verified. You can prevent such exception with a new timeout value. The code snippet in Fig 11 shows the usage of setting a new timeout value.

Attachments