AN11876

Starting development with TapLinx SDK

Rev. 2.0 — 1 September 2020 Application note
617120

Document information

Information	Content
Keywords	TapLinx, Android SDK
Abstract	This application note introduces the TapLinx SDK for Android and Java desktop development kit and shows how to integrate it into Android Studio for own application developing. It continues to give a short overview of the MIFARE DESFire and it closes with a collection of typical use-cases with MIFARE products and how they can be solved with TapLinx code snippets.

Revision history		
Rev	**Date**	**Description**
2.0	20200901	Rewritten for a new document editing system, sections for older versions removed
1.6	20190911	New chapter "How to start with TapLinx for Desktop"
1.5.2	20190902	Section "Release notes for Version 1.7" added
1.5.1	20190326	Section "Release notes for Version 1.6" added
1.5	20181016	Section "Release notes for Version 1.5" added
1.4	20180723	Section "Release notes for Version 1.4" added
1.3	20180312	Section "Using TapLinx with an AAR library" added
1.2	20170619	Chapter "Short introduction into the MIFARE DESFire architecture" inserted
1.1	20160921	Some links and references updated
1.0	20160913	Document release 1.0
0.9	20160802	Start of document

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 helps you to start with TapLinx and explains the steps which are required for integrating and using TapLinx with Android Studio. It also explains how to register the library and how to start with your own app. In the last chapter, you find tips and tricks for typical use-cases for MIFARE products.

1.1 Why using TapLinx

If you decide to implement your own Android app for communicating with a MIFARE, NTAG or ICODE products, many obstacles laying on your way usually. Using security features, which is a very common use case, requires on the Android device to implement encryption algorithms. Moreover the secure communication requires additional methods like CRC32, MAC signing etc. All of these algorithms must be implemented before any user-specific method can be used.

NXP provides data sheets and applications notes (AN) for giving hints and pre-calculated examples to make the life easier for the developers. But an own implementation can be a painful and long journey because of the adversities of debugging crypto systems. Here comes TapLinx into the game and shortcut the massive effort for an own low-level implementation! All low-level functions are encapsulated in the TapLinx library, no detailed datasheets are required anymore.

TapLinx offers high-level methods like `readData()` and make all encryption, MACing and what is required for the selected communication mode “under the hood”. The developer can focus to the project requirement and let TapLinx does the rest.

1.2 Where to find the code snippets

As shown in figure Figure 1 the attachments are visible by clicking the paper-clip icon.

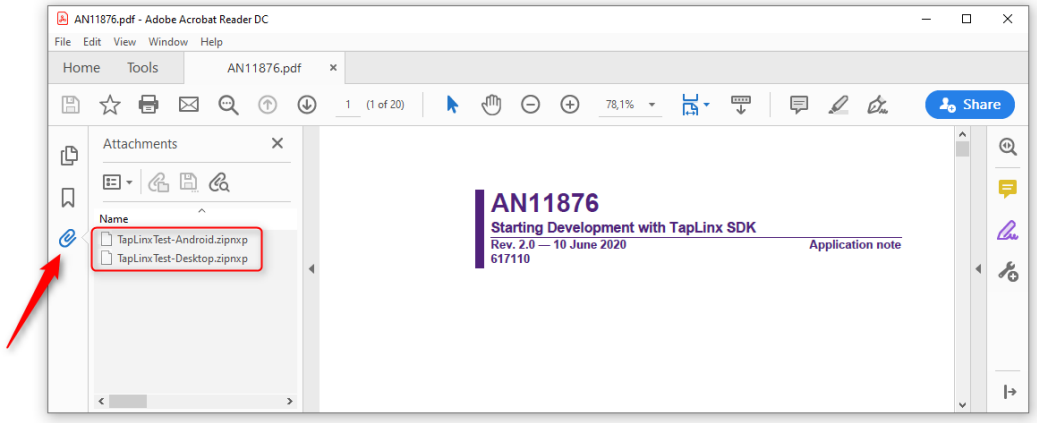

Figure 1. Code snippets of this application note

The attachments are ZIP archives, showing in Table 1:

Table 1. ZIP Archives in the attachment

File Name	Description
TapLinxTest-Android.zipnxp	Contains a “skeleton” *Android Studio* project app with all preparations to detecting a tapped card
TapLinxTest-Desktop.zipnxp	Contains a Java AWT app for *Eclipse IDE* polling a connected USB reader and show the content in the app

Note: The file extension is “zipnxp”. After saving the file from PDF document change the extension to “zip”. The reason for the unusual extension is the ban on no EXE and ZIP attachments in PDF documents.

1.3 Where to download additional resources

Using the TapLinx AAR file is one of the two options how TapLinx can be integrated in an Android app. Section 3.2 explains both approaches in detail. The AAR file of the current version is always available on the TapLinx main site (after a successful login):

www.mifare.net/taplinx/

Figure 2 shows the TapLinx information panel where material can be obtained. The red frame shows the download for the Android AAR file.

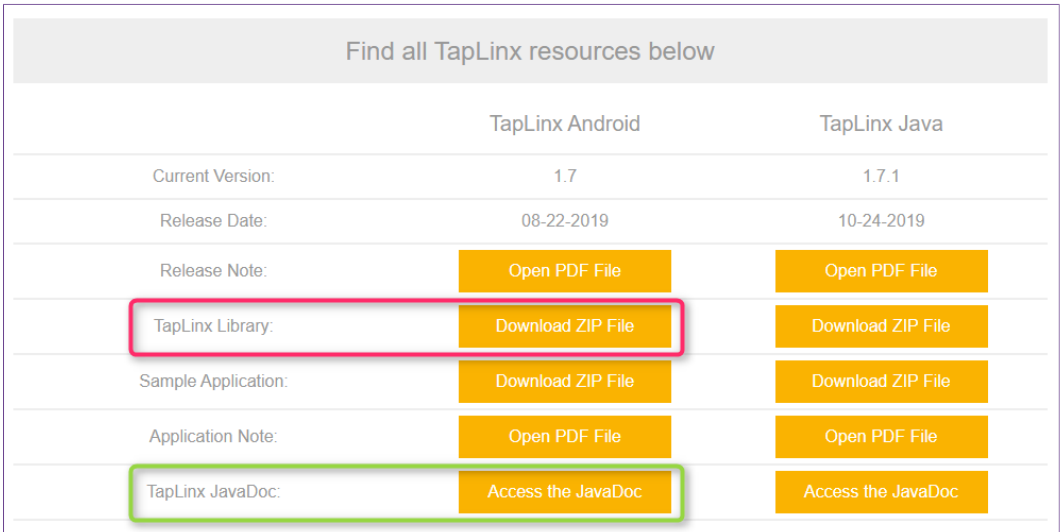

Figure 2. The “button panel” for getting additional TapLinx resources

From the panel also other TapLinx material can be downloaded, the change history, documentation and this AN.

1.4 Resources for help and documentation

The standard Java development tools contain the *JavaDoc* compiler to generate HTML files from the embedded documentation snippets in the source files. The JavaDoc files of TapLinx are available online and as ZIP archive for download.

The TapLinx main site shows (after login) a link to “Java Documentation” and open the complete documentation under this link:

https://www.mifare.net/developer/javadoc/android/

The JavaDoc files explain all product-specific classes and utility classes. The documentation is also available as ZIP archive via download from the “button panel” on main site. The green frame in Figure 2 shows the download of JavaDoc archive.

If still some questions open, every developer can ask in the TapLinx user forum, available over the button “Forum” on the main site. But before you ask a hint: many typical questions have already been asked. Please look into the posts whether you find your question there. Often you find code snippets in the answer to show how to implement the solution with TapLinx. The user forum can be reached from this link:

https://www.mifare.net/support/forum/forum/taplinx-developers/

2 The online registration procedure

TapLinx need to be registered with the package name of the application. The package name of an Android app should reflect the associate company URL and the name of the app itself. Such package name must be unique if it is intended to publish the app on Google Play. Even if it is not intended to publish the app, it is a good idea to follow the rules of a meaningful package name. The package name must be unique on the NXP server for getting a TapLinx license string.

The registration process returns a license key string which must be inserted in the source file where the registration method is called (details of this method will be shown later). When the app starts for the very first time, TapLinx tries to open an Internet connection to the NXP server for registration verification. If the verification succeeds, the registration is finished and not started again for the life time of the app.

If an Internet connection cannot be established, the registration verification is postponed to the next starting of the app. This postponement can occurs a maximum of 10 times before TapLinx becomes inactive. There is no way to recover from this inactive state after 10 postponements. To continue using the device for development, it requires to uninstall the app and install it again.

It is possible to use TapLinx with a standalone app without any Internet connection. In this case, a so called “offline registration” should be used. This kind of registration uses two license strings, an “online string” and an “offline string”. Even in the case the app gets an Internet connection in the meantime, TapLinx continually works with the online string and later with the offline string.

2.1 The registration portal for TapLinx

NXP offers an online portal for TapLinx registration. For the registration, a valid email address and a password must be entered. For TapLinx app registration, this URL is available:

https://inspire.nxp.com/mifare/

After signing-in, an overview of all registered apps is shown (see Figure 4). The button “Add New App” allows adding a new package name into the personal package list (see Figure 3).

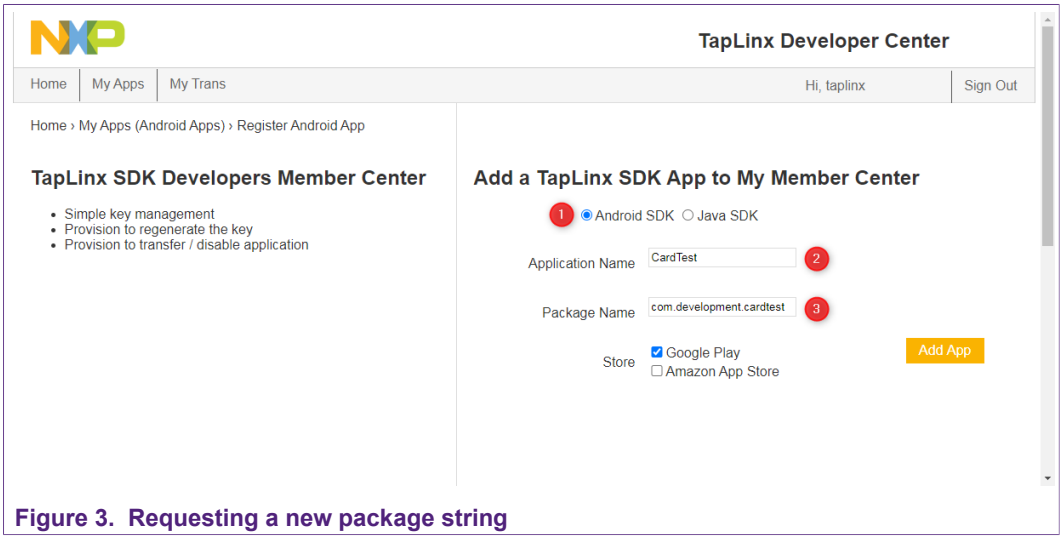

Figure 3. Requesting a new package string

With the radio button (1), you select between the “TapLinx for Android” and “TapLinx for Java Desktop” library. You can give a decorative name (2) and finally a package name (3). As mentioned, the package string must be a valid Android format and unique on the NXP server.

After entering a valid package, the license string is shown on the main page where all registered apps are shown (see Figure 4).

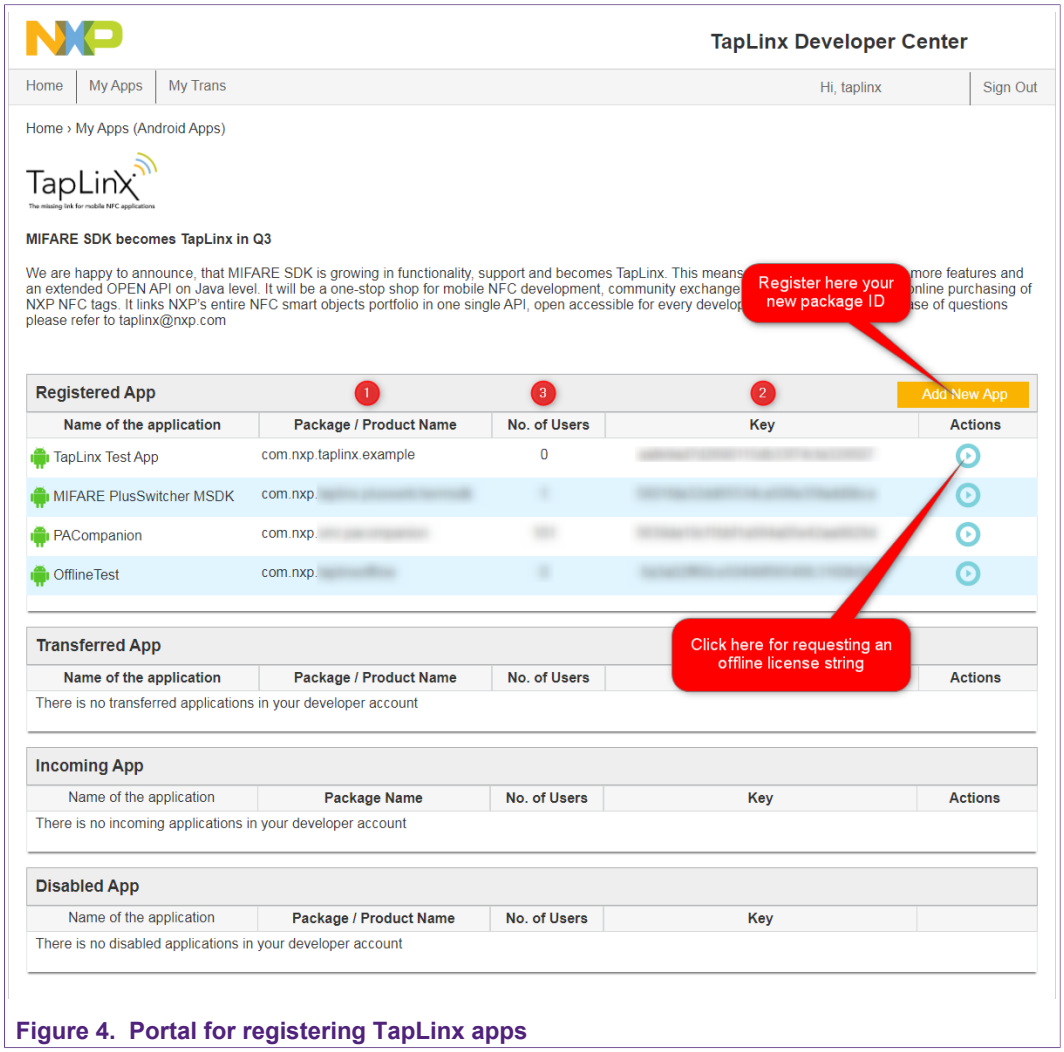

Figure 4. Portal for registering TapLinx apps

The main page shows the package name (1), number of activations (2) and the license key itself (3). Note, the real license strings are obfuscated for this screenshot. Every new activation of your distributed app is registered and increments the “No. of Users” (2).

With the “Action button”, an offline license string can be requested (see Figure 5). With this button, a second license string is requested for inserting it in the source code.

There are also buttons for “Regen Key”, “Transfer” and “Disable”. Do not click theses buttons. In particular, do not click “Disable”, the package name is set as “disabled” and it cannot be recovered again!

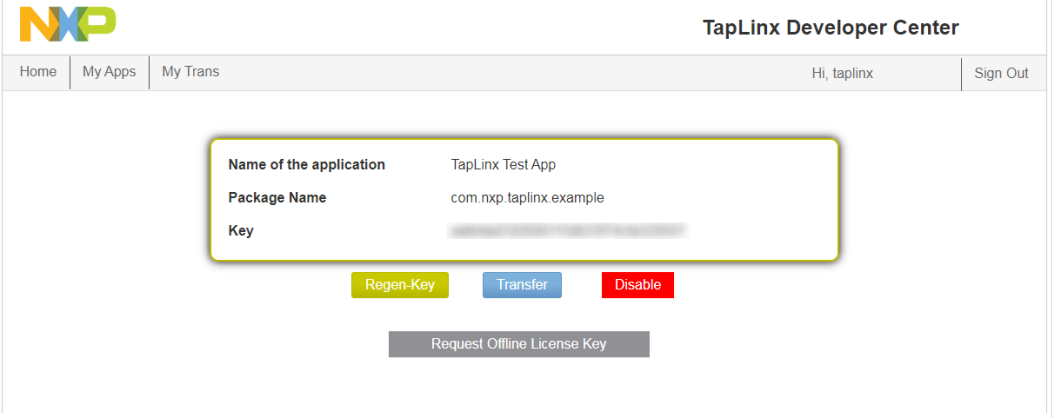

Figure 5. Request for an offline license string

For the impatient reader, the code snippet where the license string should be inserted is shown below. The method `registerActivity()` has two flavors. One with two parameters were the last is the online license key and a second one with three parameters, where the last both are the online and the offline license strings. Section 3.1.2 shows this in detail.

```
// Reference to TapLinx library
private NxpNfcLib m_libInstance = null;

// ...

m_libInstance = NxpNfcLib.getInstance();

// Register with a "online" license string (xxx)
m_libInstance.registerActivity( this, "xxx" );

// Register with a "offline" license string (yyy)
m_libInstance.registerActivity( this, "xxx", "yyy" );
```

One word at the end. You should check the log for the key word “LICENSE_VERIFER” about the success of the registration. If you find in the log something shown in Figure 6, then the registration fails.

```
26946-26946/com.nxp.taplinx.taplinxtest I/LICENSE_VERIFIER: Attempting Local License Verification
26946-26946/com.nxp.taplinx.taplinxtest E/LICENSE_VERIFIER: Local License Verification failed
26946-26946/com.nxp.taplinx.taplinxtest E/LICENSE_VERIFIER: Storing Failure Count: 3
```

Figure 6. Log message if registration verification fails

The failure counter cannot be reset! To get rid of the error state, the app must be uninstalled and installed again.

3 How to start with TapLinx for Android

TapLinx Android is one of the two supported platforms described in this AN. The other platform, *TapLinx for Desktop (Java)*, is described in Section 4. The library can be obtained in two ways: first bind it as Maven repository link to your Android Studio project. The library is downloaded automatically in the build process. The other way is to download the library as AAR file from our server manually and bind it to your project as static library. Both approaches have their pros and cons. The Maven repo makes it easy to update from one version to another, the static library allows continue working if you have network issues and the repo server is not available. Both approaches are introduced in this AN.

3.1 Modifications in the source files

For using TapLinx, some modifications must be done in the project files. Some of them are required for using NFC in general, and some of them are specific required for using of TapLinx. All modifications are shown in the following paragraphs and are also available in a “skeleton” project, attached as ZIP archive to this AN. Refer to Section 1.2 for details how to get the sample project files.

3.1.1 Modifications in the Manifest

The Manifest file in an Android project lists all permissions which the app needs from the OS. Usually, the permissions are shown before an app is being installed. A user can accept or decline a permission. How the app responds in the case of declination depends on the app software. Usually, the installation is aborted if one of the permissions cannot be granted.

For TapLinx, the permissions from Figure 7 are required.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.nxp.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:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

Figure 7. Modifications in the Manifest

1. The NFC permissions must be requested, naturally. TapLinx also want to use Internet connection and therefore also this permission must be requested also, even if the user app does not use Internet.
2. The “uses-feature” is not essential, but it helps to pre-select devices, connected to Google Play. If a device has no built-in support for NFC, it is not listed as “installable app”.

The Manifest file can be found in this location: `TapLinxTest/app/src/main/AndroidManifest.xml`.

3.1.2 Modifications in the Java files (main activity class)

The main activity class of an Android project is the “body” of the app. This class fetches all callbacks and here also all system service tasks are handled. For using NFC and using TapLinx, some supplements must be implemented in this file.

```
public class MainActivity extends AppCompatActivity
{
   private String    TAG = MainActivity.class.getSimpleName();
   private TextView  m_textView = null;
                                                    // The licence key (not valid)
1  private String    m_strKey = "00112233445566778899aabbccddeeff";

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

Figure 8. Modifications in the main activity, part 1

The first modification, shown in Figure 8, is the definition of the package license string (1) and the class reference to the TapLinx library (2). Note, the shown license string is only an example, it is not a valid package string!

```
1  private void initializeLibrary()
   {                                                // Initialize the TapLinx library
      m_libInstance = NxpNfcLib.getInstance();
      m_libInstance.registerActivity( this, m_strKey );
   }

   @Override
   protected void onCreate( Bundle savedInstanceState )
   {
      super.onCreate( savedInstanceState );
      setContentView( R.layout.activity_main );

      m_textView = (TextView)findViewById( R.id.mainTextView );

2     initializeLibrary();                          // Initialize library
   }

   @Override
   protected void onResume()
   {                                                // Called if app becomes active
3     m_libInstance.startForeGroundDispatch();
      super.onResume();
   }

   @Override
   protected void onPause()
   {                                                // Called if app becomes inactive
4     m_libInstance.stopForeGroundDispatch();
      super.onPause();
   }
```

Figure 9. Modifications in the main activity, part 2

Figure 9 shows the second modifications in the main activity. Every app must in `onCreate()` initialize its GUI widgets and it is recommended initializing TapLinx also in this method. The TapLinx initialization in Figure 9 is implemented in `initializeLibrary()` (1) and called in `onCreate()` (2). In a case of using an offline key, the method `registerActivity()` has an additional parameter.

Generally, an app in the foreground can always move into the background if the user starts a new app or if system events trigger this. If an app uses NFC communication, then the current communication is affected by this movement. If an app goes into the background, it should also release its current NFC channel and regain it again if it is moved back into the foreground. The methods `startForeGroundDispatch()` (3) and `stopForeGroundDispatch()` (4) control this behavior and should be implemented as shown.

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

2 private void cardLogic( final Intent intent )
{
   if( DESFireEV1 == m_libInstance.getCardType( intent ) )
   {
      IDESFireEV1 objDESFireEV1 = DESFireFactory.getInstance()
                                                .getDESFire( m_libInstance.getCustomModules() );
      try
      {
         objDESFireEV1.getReader().connect();

         Log.d( TAG, msg: "TapLinx version: " + NxpNfcLib.getTaplinxVersion());
         Log.d( TAG, msg: "Phone modell   : " + Build.MODEL );
         Log.d( TAG, msg: "Card details   : " + objDESFireEV1.getCardDetails().cardName );
         Log.d( TAG, msg: "Free memory    : " + objDESFireEV1.getFreeMemory() );
         Log.d( TAG, msg: "Total memory   : " + objDESFireEV1.getTotalMemory() );

         // More implementation follows here
      }
      catch( Throwable t )
      {
         t.printStackTrace();
      }
   }
}
```

Figure 10. Modifications in the main a, part 3

Figure 10 shows the implementation of the callback for NFC events. Method `onNewIntent()` (1) is called for any kind of “intents” and the method `cardLogic()` (2) is the implementation of card handling with TapLinx. The code snippet is willing only to handle card intents triggered from a MIFARE DESFire EV1. The TapLinx architecture returns a “card object” of that type which is used for all card operation. All supported methods are implemented in this object. Therefore, at the beginning of a card operation, the card identification and selection is required to choose the right object.

The third code modification in Figure 10 checks the card type and connect to the NFC adapter with `getReader().connect()`. There is no more implementation for this first introduction. In other code snippets begin at that point the card typical implementation. The main activity class file can be found in this location: `TapLinxTest/app/src/main/java/com/nxp/taplinxtest/MainActivity.java`.

3.2 Modifications in the project files

The final step in the preparation process is the linking of the TapLinx library into the project. This is done via modification in the Gradle file. Gradle is the underlaying build engine in Android Studio to bring all modules together.

TapLinx can be used in two ways. The first approach uses a reference entry in the Gradle file and the let Android Studio downloading, binding, and linking TapLinx. We call this way the “Maven approach”. The alternative is to download TapLinx manually, put the library into the project and direct Gradle to link this library into the project. We call this way the “AAR library approach”.

3.2.1 Modifications in the project files for using the “Maven approach”

The “Maven approach” uses TapLinx delivered via *Maven* repository loading from *Android Studio*. Usually the build engine resolves all dependencies and links the library into the project.

```
android {
    compileSdkVersion 23
    buildToolsVersion "22.0.1"
    defaultConfig {
        applicationId "com.nxp.taplinxtest"
        minSdkVersion 18
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
    repositories{
        flatDir{
            dirs 'libs'
        }
        maven{
            credentials{
                username "sdkuser"
                password "taplinx"
            }

            url "http://maven.taplinx.nxp.com/nexus/content/repositories/taplinx/"
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:23.0.0'
    testCompile 'junit:junit:4.12'
    compile('taplinx-android:nxpnfcandroidlib:1.2@aar') { transitive = true }
}
```

Figure 11. Modifications in the Gradle file for Maven repo binding (Android Studio 3.x)

Figure 11 shows the Gradle file for an Android Studio version 3.x and a Gradle version 2.x. The syntax in the lower red frame with the keyword “compile” is deprecated in the newer version Android Studio 4.0. In the older version, it was required to request Gradle

to resolve dependencies in the linked libraries per command. This is the yellow marked statement “transitive = true” in Figure 11.

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 29

    defaultConfig {
        applicationId "com.nxp.taplinx.xxx"
        minSdkVersion 23
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
                                                                    'proguard-rules.pro'
        }
    }
    repositories{
        flatDir{
            dirs 'libs'
        }
        maven{
            credentials{
                username "sdkuser"
                password "taplinx"
            }
            url "http://maven.taplinx.nxp.com/nexus/content/repositories/taplinx/"
        }
    }
}

dependencies {
    implementation fileTree(dir: "libs", include: ["*.jar"])
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'taplinx-android:nxpnfcandroidlib:1.7'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

Figure 12. Modifications in the Gradle file for Maven repo binding (Android Studio 4.0)

For Android Studio 4.0, the new syntax is “implementation” and all dependencies are resolved automatically (see Figure 12). This Gradle file is part of the skeleton project and can be found in this location: `TapLinxTest/app/app/build.gradle`.

3.2.2 Modifications in the project files for the “AAR Library” approach

The “AAR Library” approach uses TapLinx via the AAR library in the project folder. It is a static library which resides on a local device, no synchronizing in the build process is required. Projects, created by Android Studio contains a “libs” folder in the app project directory. You find this folder in the sample project: `TapLinxTest/app/libs`. This is the destination for all external libraries an Android Studio project want to use. At creation of a new project, this folder is empty.

For this approach, the AAR file is needed. Download the TapLinx AAR file as described in Section 1.3 and put the file into “libs”. Open the project settings via “File|Project Structure…” menu item and click to “+” button for adding a new module (see Figure 13).

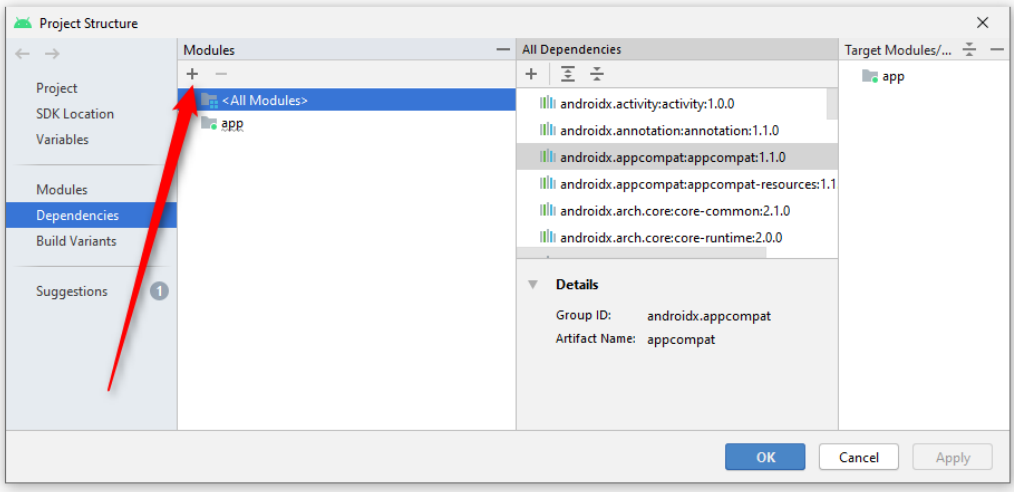

Figure 13. Add a AAR library to your project (part 1)

A dialog opens and you select the item “Import JAR/AAR Package”. After closing, another dialog opens and asks the library to import (see Figure 14). Enter here the path to the AAR file in “libs” directory.

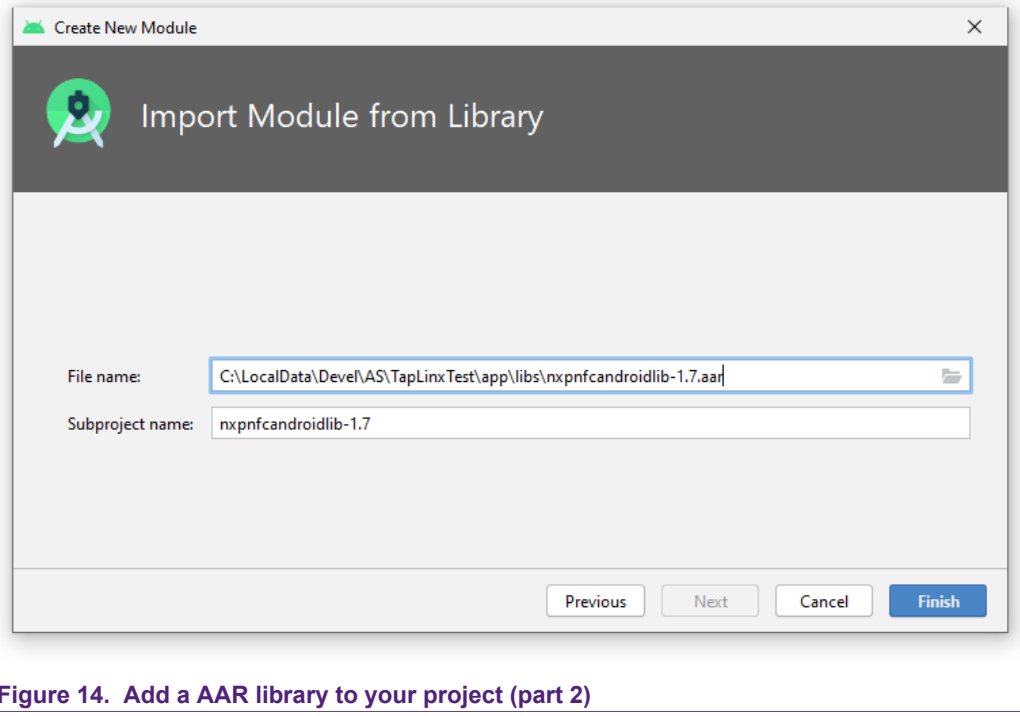

Figure 14. Add a AAR library to your project (part 2)

Click “Finish”. The library is now included into the project structure. The next step is to add this library into the dependency list of the project. Select the app project in “Modules” and click to the “+” button (indicated by a red arrow) and add a new dependency (see Figure 15).

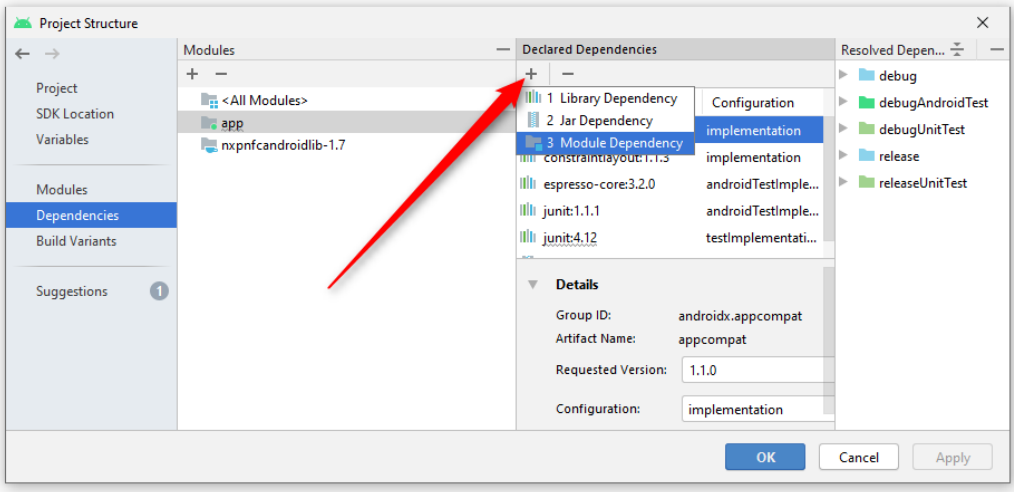

Figure 15. Add a AAR library to your project (part 3)

Select the library module (nxpnfcandroidlib-1.7) as dependency with the configuration "implementation" (see Figure 16).

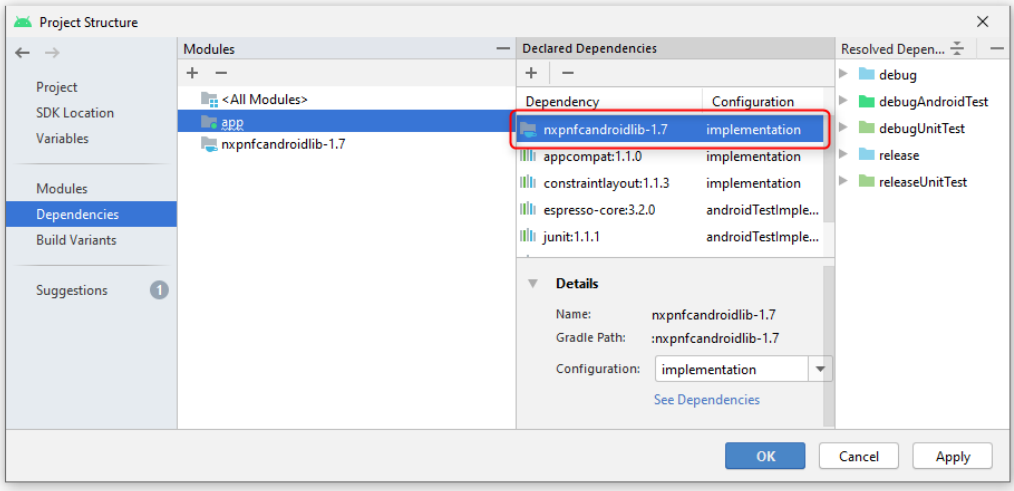

Figure 16. Add a AAR library to your project (part 4)

TapLinx is now integrated into the app project. The final step is to add Firebase core library into the project, because TapLinx use the Firebase Analytics library. The easiest way to achieve this is to add a single line in the Gradle app file. Figure 17 shows the Gradle app file. The changes marked with a red frame. The upper line was inserted by Android Studio in the previous adding TapLinx to the project. The lower line is inserted by hand for shortness.

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 29

    defaultConfig {
        applicationId "com.nxp.taplinx.taplinxtest"
        minSdkVersion 23
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
                                                                        'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: "libs", include: ["*.jar"])
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation project(path: ':nxpnfcandroidlib-1.7')
    implementation 'com.google.firebase:firebase-core:17.2.2'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

Figure 17. The app's Gradle file for AAR approach

Different TapLinx versions need different versions of Firebase to be linked. Table 2 shows the dependency of the last TapLinx versions and the appropriate Google libraries.

Table 2. Dependencies of TapLinx to used Google library versions

TapLinx library version	Google analytics/firebase version required
1.4.1	com.google.android.gms:play-services-analytics:15.0.2
1.5	com.google.android.gms:play-services-analytics:16.0.3
1.6	com.google.firebase:firebase-core:16.0.7
1.7	com.google.firebase:firebase-core:17.2.2

The benefit of this approach is the ability to work completely offline. The library is part of the local project files, no synchronizing is required with the NXP server.

4 How to start with TapLinx for desktop

Since version 1.7 TapLinx is also available for Java desktop development. It bases on the source repository of TapLinx Android. Most of the product classes for the MIFARE and NTAG products can be used also for TapLinx Desktop. The main difference is the missing of signaling methods in TapLinx Desktop. In Android, a tag in front of the NFC reader will be detected and a signaling mechanism calls a method in the Android user app. TapLinx Desktop bases on PC/SC Readers connected to the device. There is no such signaling mechanism available. So, a TapLinx Desktop app must poll the PC/SC interface for detecting a card in the field.

4.1 The TapLinx desktop sample app

TapLinx Desktop comes as archive with single JAR libraries for each MIFARE or NTAG family product. Not all libraries are required for a project (the sample app uses all JAR libraries as shown in folder “lib” in the Eclipse project). If the project only uses a MIFARE DESFire for instance, then only the desfire-x.y.jar is need in the custom project. But the library management in file librarymanager-x.y.jar is always required.

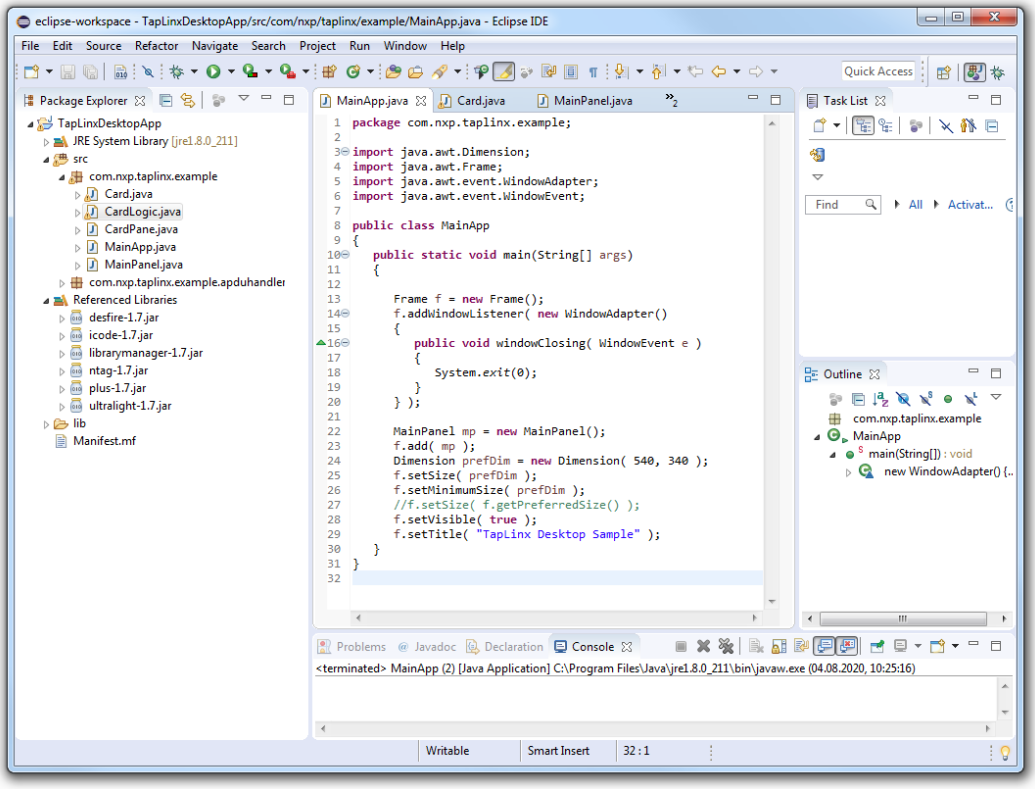

Figure 18. Eclipse with TapLinx desktop libraries

This application note also contains a TapLinx desktop example project, based on Java AWT. The simple GUI opens a window and allows it to select one of the installed PC/SC NFC readers. If a MIFARE DESFire is put to the reader and the button “Read Again” is clicked, the application starts reading and tries to authenticate to the default PICC Master Key and read the applications used on the card.

For the impatient reader, the link between TapLinx and Java AWT is implemented in class `com.nxp.taplinx.example.MainPanel`. The method uses the class

`com.nxp.taplinx.example.CardLogic` which encapsulates the access to card interface of TapLinx Desktop. The class `CardLogic` does not only contain code for the MIFARE DESFire, you will found code samples for other products as well in this class. Note, this simple app is not a “full feature app” and gives only an idea how to start with the own TapLinx Desktop app programming (see Figure 19 for a screenshot of this app).

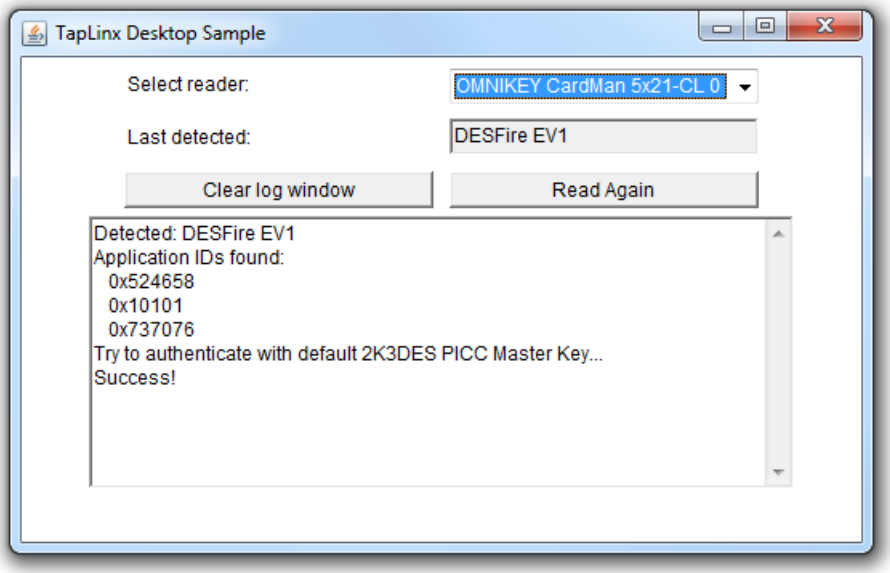

Figure 19. Simple desktop application based on TapLinx Desktop

The sample app was created with the *Eclipse IDE for Java Developers*, version 2019-06 (4.12.0). Refer [Eclipse] for the download of Eclipse IDE. The source folder of the project was saved in the archive `TapLinxTest-Desktop.zipnxp`. The archive can be found in the attachment of this document. Refer to Section 1.2 on how to retrieve the attachment of this document.

5 Short introduction into the MIFARE DESFire architecture

The methods of the TapLinx SDK are built directly on the native command set of the certain product. This chapter gives a deeper look to a product to help developers to find the right decisions for their own application.

We use one of the successful products of the MIFARE family: the MIFARE DESFire. At the time of writing this AN, the MIFARE DESFire EV2 is in market and the MIFARE DESFire EV3 is being released. But for giving a short introduction, we focus to the MIFARE DESFire EV1. All later released products have additional features, but they still share the same architectural basis of the MIFARE DESFire EV1. A developer can read this chapter and use in its project the MIFARE DESFire EV2.

5.1 MIFARE DESFire EV1 architecture

The overview in this section is an abstract of the documents:

- MIFARE DESFire EV1 - Functionality of implementations on smart card controllers, [MF3ICDx1]
- MIFARE DESFire EV1 - Implementation hints and examples [AN0945]

Please ask your local NXP distributor for the documents if you want to go into more details. For both documents, a signed NDA is required.

The MIFARE DESFire EV1 is a “multi-application product” which means that different entities can put their data securely without impair and touching the other data.

The memory is organized by so called “applications”. An application can be obtained as a directory or a user space which is separated from other directories and other user spaces. An application can contain files and keys which can be used to protect the access of files. Each application has an identifier, the three-byte AID.

A new card is always delivered blank, without any installed applications. The root level of the card is called PICC level and is characterized through the AID = 0x000000. By selecting the application with AID = 0x000000, the PICC level (so the card root level) is selected.

On the PICC level, no files can be directly created, but it is possible to create applications on the PICC level. Further on the PICC level offers the possibility to make multiple configuration settings and contains card management keys like the PICC Master Key.

The PICC Master Key is known by the card issuer and with the appropriate settings there is no reason to share this key with third-party entities. If it is generally allowed (see the access permission list below) by card issuer, a third party can create its own application and used it for storing files in it.

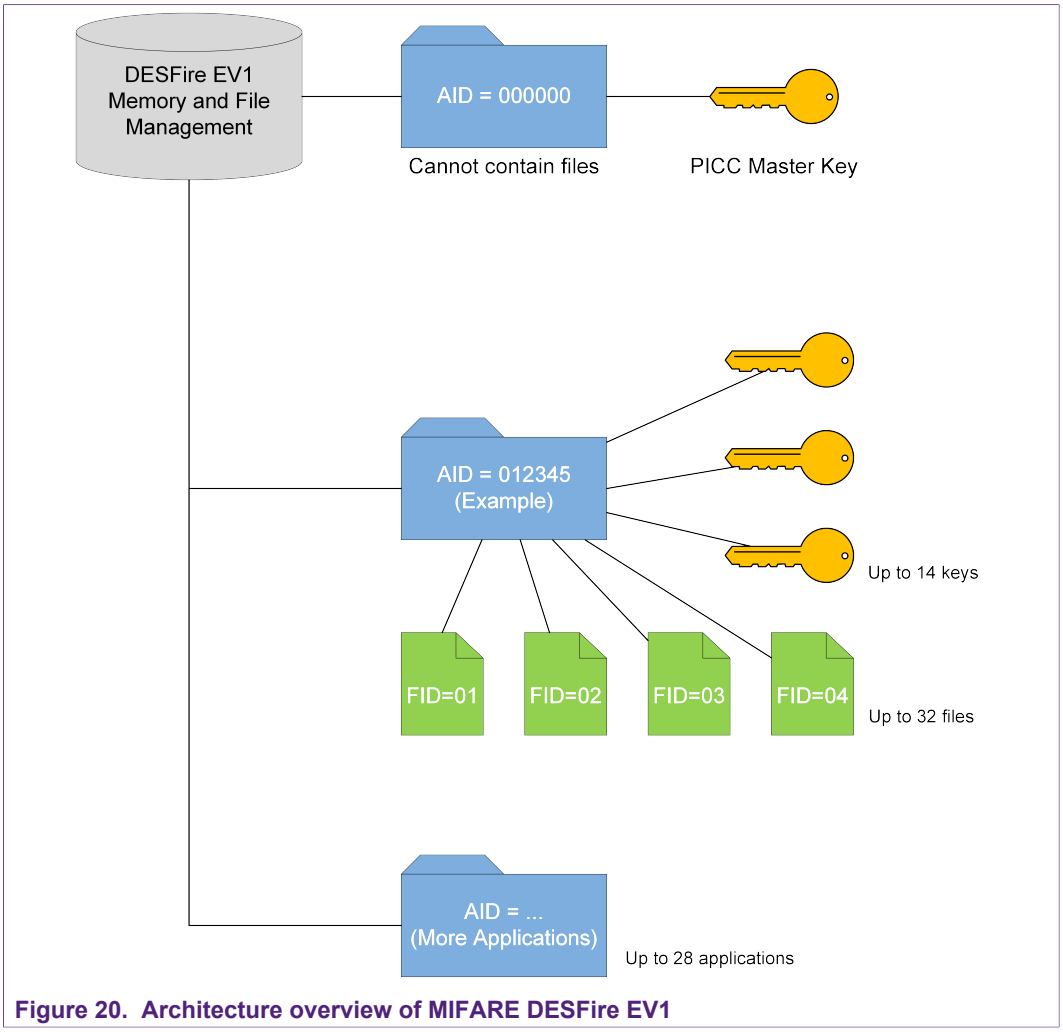

Figure 20. Architecture overview of MIFARE DESFire EV1

Except for application AID = 0x000000, there exist no other applications on a blank card. The first thing to start is to create a user application with an AID different of 0x000000. While creating the application, the following information must be defined:

- The number of keys used within this application.
- Application access permission settings:
  - Define access key permissions
  - Configuration is changeable without Master Key yes/no
  - Free list directory allowed without Master Key yes/no
  - Free create/delete files allowed without Master Key yes/no
  - Allow change of the Master Key yes/no
- Define the crypto method of this application:
  - DES or 2K3DES cipher (not recommended)
  - 3K3DES cipher (not recommended)
  - AES128 cipher (recommended)

The PICC Master Key is available for authentication to the IC and is also important for all configuration and administration actions that can be taken on the IC. It has the key number 0x00 on PICC level. Also, on application level, the application key number 0x00 acts as the Application Master Key, having usually configuration purpose of the

application. Each application can be created with a flexible number of keys, ranging from no keys to up to 14 keys per application.

The 2K3DES/3K3DES cipher is available to be compatible with former customer infrastructures. This cipher should not be used in new projects.
As a recommendation: AES128 is the strongest cipher on a DESFire EV1 and should always be preferred for a maximum protection!

After creating an application, in the next step files can be created within this application.

5.2 Select or change between applications

To deal with applications, it is elementary to understand the MIFARE DESFire architecture. So, all operations like creating files or read from files always relate to an application. This is very similar to the concept of a directory in a PC file system. The command `createApplication()` creates an application which then can be selected. Inside these applications, files can be created. Before any other command takes place, always a `selectApplication()` should be used to set the DESFire in a known state. If no applications exist, `selectApplication(0x000000)` should be used to address the PICC Master Key for a following authentication.

The command `getApplicationIDs()` retrieves all available applications on a card. This command allows it to check if the target application exists on the card. This command is helpful to prevent running into an error if a user taps a different MIFARE DESFire EV1 to the reader which does not contain the desired application.

5.3 Protected access with an authentication

It is very important to understand the authentication concept for protecting the access to the application. An authentication always uses a key which is used for protecting the following communication. If the authentication fails, no other accessing method (reading or writing) can be executed. Only several commands like get key version or authenticate can be executed.

The authentication verifies that the participant knows the same secret as on the card (the authentication key) and can be trusted and allow them the next operations like change a key or remove a file.

To read a file in encrypted communication mode also it is necessary to authenticate with the correct key, according to the access right settings of the targeted file. This key is used for used for authentication and session keys are derived which are used for the following MAC and encryption operations. A typical sequence could be:

```
com.nxp.nfclib.desfire.IDESFireEV1 desfireObj = …

// Select user application
desfireObj.selectApplication(…);

// Authentication with encryption key
desfireObj.authenticate(…);

// Read encrypted data
desfireObj.readData(…);
```

5.4 File communication and access modes

With the command `createFile()` on the one hand, a communication mode must be defined and on the other hand an access permission of the file must be defined. Caution, the one affects the other!

A user can set individual access permissions for “read access”, “write access”, combined “read and write access” and “change access”. The permission is expressed with a byte 0x00 … 0x0F. Hence the values 0x00 until 0x0D relate to the appropriate key numbers and 0x0E means “free access” and 0x0F “no access”. A total number of 14 keys can be used in a single MIFARE DESFire application.

A user can also define the three communication modes “plain”, “MACed” and “fully encrypted”. Table 3 shows this relation:

Table 3. Communication modes in relation to access permissions

Communication mode	Access permission	Result
Plain	Key (0x00...0x0D)	Requires authentication with the specified key, data communication in plain text
Plain	Free (0x0E)	No authentication needed, data communication in plain text
MACed	Key (0x00...0x0D)	Requires authentication with the specified key, data communication in plain text, with a MAC appended to ensure integrity
MACed	Free (0x0E)	No authentication needed, data communication in plain text
Encrypted	Key (0x00...0x0D)	Requires authentication, data communication encrypted
Encrypted	Free (0x0E)	No authentication needed, data communication in plain text

Particularly, a file which is created to use encrypted communication can be annulled to plain communication if a key number of 0x0E is defined, what might be desired in some scenarios.

5.5 File types

The MIFARE DESFire EV1/EV2 offers several file types which make it easier to handle backups and file protection scenarios against damage. “Against damage” means in this context that the tag is removed from the reader station while the writing process is pending! This is a typical use case which must be considered.

The MIFARE DESFire IC offers for all files which are protected by a backup mechanism (BackupDataFile, ValueFile, CyclicRecordFile, LinearRecordFile) an anti-tearing protection as well as an automatic backup management. Backup files use an internal mirror image where a copy of the entire content is saved. Every modification is done in the mirror image until the command `commitTransaction()` is called. With this command, the main image is updated with the content of the mirror image. If a tearing (an unexpected removal of the card from the field or any other scenario that causes communication break during writing), while the mirror image is written, the main image is untouched and the data in the file always stays consistent. Any change to the mirror image can also be invalidated intentionally with `abortTransaction()`.

Value files store only one integer value but make it easy to deal with increment and decrement operations. Any sequence of changes like `credit()` (increase

the value) and `debit()` (decrease the value) must be terminated also with `commitTransaction()` or `abortTransaction()`.

5.6 Key management

An application can contain several keys for different purposes. The Application Master Key should be used only for setting the application permissions and separated from keys for reading and writing. In case a key is compromised and must be changed with another value, a key version should be used to take care the correct key is used. Typically, a first issued key start with key version 0. The command `getKeyVersion()` can always be used without any authentication to ensure that the correct key is used on this particular card.

The last step in a typical personalization process is to change the default keys against the customer values. As a recommendation: a key should always have generated from a true random generator. The command `changeKey()` changes the old key value with the new key value. In the case the card is already issued, a change of key should always increment the key version too.

There is no way to read a key value from the MIFARE DESFire IC. Prior to changing a key, it is mandatory to authenticate. After a key change took place, an authentication can be executed to check if the key was updated correctly.

6 Some typical use cases of TapLinx programming

In this section, we introduce some use-cases and scenarios of transactions with MIFARE and NTAG products and show how they can be handled with TapLinx. These are procedures which every developer needs to know to be successful and efficient with TapLinx.

All examples and code snippets in this chapter are developed and built with Android Studio 4.0 and TapLinx version 1.7. Android Studio is available from [AS]. Section 1.3 explains how to get TapLinx as separate library.

6.1 Interacting with a MIFARE Classic EV1

As already mentioned TapLinx uses the Android NFC Adapter library and therefore, it must use the underlying base classes for connecting with the contactless card. Therefore, to getting the MIFARE Classic EV1 object it needs an extra step. This extra step is retrieving first the `Tag` object and then the MIFARE Classic EV1 object. This extra step is needed for cards using ISO14443-3 protocol like the MIFARE Classic and is not required for cards using the IEC14443-4 protocol.

The code snippet in Figure 21 shows how to read block 0 of sector 0. Before any block can be accessed, the sector must be authenticated with one of keys A or B. You can use the key for different purposes like reading a block with key B and writing to it with key A. Refer to section 8.7 of [MF1S50YYX_V1] for information about memory access permission. TapLinx offers a helper function `blockToSector()` to get the sector number from a given block number used in the authentication function. The authentication uses the default key, defined in `DEFAULT_MIFARE_KEY`. Keep in mind, block 0 in the first sector contains the card UID and is not writeable. After the authentication, block 0 is read. Keep in mind, the authentication allows it to read or write any block in the sector. Only if another sector should be accessed, the authentication must be repeated to the new sector.

The MIFARE Classic EV1 allows it to increment and decrement a so called “value block”. A value block is a 16-byte data block and manages a 32-bit signed integer. It is used typically for currency values where only adding subtracting of values are performed. TapLinx offers commands for increment and decrement but no command for initializing a value block. Such block must be initialized external by an array of bytes. Figure 21 shows the value block for the value 1 in the array `DEFAULT_VALUE_BLOCK`. For detailed description of value blocks, refer [MF1S50YYX_V1], section 8.6.2.1.

The log output of the code snippet shows Figure 22. An attentive reader will notice that the 32-bit integer shows the least significant byte first. All MIFARE products use *LSB first* notation in their number representation.

A developer who wants to use a MIFARE Classic must keep in mind that not all devices support the MIFARE Classic IC-based contactless card interface. The reason is the proprietarily cipher of the MIFARE Classic which must be implemented in hardware in the reader IC. All NXP reader IC supports the MIFARE Classic but this is not true necessarily for all other reader manufacturers. Unfortunately, there is no way to check this capability from user software on an Android device and there is also no way to overwrite this behavior with user code!

A MIFARE Classic might not be supported on all Android devices if the built-in hardware has no support for this product.

Unfortunately, the locked Android firmware does not allow to verify the hardware for supporting or to overwrite this behavior.

```
public static final byte[] DEFAULT_KEY_AES =
{                                                   // Default AES key for MIFARE DESFire
   (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
   (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
   (byte)0x00, (byte)0x00
};

public static final byte[] DEFAULT_VALUE_BLOCK =
{                                                   // Default value block of value 1
   (byte)0x01, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0xfe, (byte)0xff, (byte)0xff,
   (byte)0xff, (byte)0x01, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x5a, (byte)0xa5,
   (byte)0x5a, (byte)0xa5
};

public static final byte[] DEFAULT_MIFARE_KEY =
{                                                 // Default key for MIFARE Classic
   (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff
};

private void cardLogic( final Intent intent )
{
   if( MIFAREClassicEV1 == m_libInstance.getCardType( intent ) )
   {
1     Tag tag = intent.getParcelableExtra( NfcAdapter.EXTRA_TAG );
      if( null != tag )
      {
2        IMFClassicEV1 objClassicEV1 = ClassicFactory.getInstance()
                                                      .getClassicEV1( MifareClassic.get(tag) );
         try
         {
            objClassicEV1.getReader().connect();

            Log.d( TAG,  msg: "TapLinx version: " + NxpNfcLib.getTaplinxVersion() );
            Log.d( TAG,  msg: "Phone modell   : " + Build.MODEL );
            Log.d( TAG,  msg: "Card details   : " + objClassicEV1.getCardDetails().cardName );
            Log.d( TAG,  msg: "Total memory   : " + objClassicEV1.getTotalMemory() );

             int blockIndex = 0;                    // Block to read
3            objClassicEV1.authenticateSectorWithKeyA( objClassicEV1.blockToSector( blockIndex ),
                                                                        DEFAULT_MIFARE_KEY );
                                                    // Read block data
4            byte[] data = objClassicEV1.readBlock( blockIndex );
             Log.d( TAG,  msg: "Block " + Integer.toString( blockIndex ) + ": " +
                                                          Utilities.byteToHexString( data ) );
             blockIndex = 1;                        // Block to write value "1"
5            objClassicEV1.writeBlock( blockIndex, DEFAULT_VALUE_BLOCK );
             data = objClassicEV1.readBlock( blockIndex );
             Log.d( TAG,  msg: "Block " + Integer.toString( blockIndex ) + ": " +
                                                          Utilities.byteToHexString( data ) );
                                                    // Increment by "2"
6            objClassicEV1.increment( blockIndex,  i1: 2 );
             objClassicEV1.transfer( blockIndex );  // Write back
                                                    // Read back modified block
             data = objClassicEV1.readBlock( blockIndex );
             Log.d( TAG,  msg: "Block " + Integer.toString( blockIndex ) + ": " +
                                                          Utilities.byteToHexString( data ) );
         }
         catch( Throwable t )
         {
            t.printStackTrace();
         }
      }
   }
}
```

Figure 21. Reading data from a MIFARE Classic EV1

What can a developer do to ensure its MIFARE Classic app works on a particular device? Well, NXP offers an Android app *TagInfo* to check NXP contactless products on

the device. If TagInfo detect the MIFARE Classic and gives some information about the card, then the MIFARE Classic is supported. Otherwise TagInfo shows an inoperability message. TagInfo is available free of charge at Google Play from this link: [TagInfo]

```
13175-13175/com.nxp.taplinx.taplinxtest D/MainActivity: onNewIntent
13175-13175/com.nxp.taplinx.taplinxtest D/MainActivity: TapLinx version: 1.7
13175-13175/com.nxp.taplinx.taplinxtest D/MainActivity: Phone modell   : SAMSUNG-SM-G930A
13175-13175/com.nxp.taplinx.taplinxtest D/MainActivity: Card details   : MIFARE Classic EV1 1K
13175-13175/com.nxp.taplinx.taplinxtest D/MainActivity: Total memory   : 1024
13175-13175/com.nxp.taplinx.taplinxtest D/MainActivity: Block 0: 04[redacted]0884400C82000000000
13175-13175/com.nxp.taplinx.taplinxtest D/MainActivity: Block 1: 01000000FEFFFFFF010000005AA55AA5
13175-13175/com.nxp.taplinx.taplinxtest D/MainActivity: Block 1: 03000000FCFFFFFF030000005AA55AA5
```

Figure 22. Log output from code snippet above

6.2 Authentication and key change on a MIFARE DESFire

A TapLinx user who wants to communicate with a MIFARE DESFire faces two typical use-cases: the authentication to a key and the change of keys as part of the personalization. Either a MAC protected or an encrypted communication requires a successful authentication before the file can be accessed. Figure 23 shows a code snippet where a file is read in encrypted communication mode. As recommended in Section 5.1 the cipher mode is AES.

```
public static final byte[] READ_KEY_AES_1 =
{                                                   // AES key for file #1
   (byte)0x01, (byte)0x01, (byte)0x01, (byte)0x01, (byte)0x01, (byte)0x01, (byte)0x01,
   (byte)0x01, (byte)0x01, (byte)0x01, (byte)0x01, (byte)0x01, (byte)0x01, (byte)0x01,
   (byte)0x01, (byte)0x01
};

public static final byte[] READ_KEY_AES_2 =
{                                                   // AES key for file #2
   (byte)0x02, (byte)0x02, (byte)0x02, (byte)0x02, (byte)0x02, (byte)0x02, (byte)0x02,
   (byte)0x02, (byte)0x02, (byte)0x02, (byte)0x02, (byte)0x02, (byte)0x02, (byte)0x02,
   (byte)0x02, (byte)0x02
};

private void cardLogic( final Intent intent )
{
   Key keyDefault;
   KeyData keyDataDefault;

   if( DESFireEV2 == m_libInstance.getCardType( intent ) )
   {
      IDESFireEV2 objDESFireEV2 = DESFireFactory.getInstance()
                                                .getDESFireEV2( m_libInstance.getCustomModules() );
      try
      {
         objDESFireEV2.getReader().connect();
                                                //Read key for file #1
      1  keyDefault = new SecretKeySpec( READ_KEY_AES_1, algorithm: "AES" );
         keyDataDefault = new KeyData();
         keyDataDefault.setKey( keyDefault );

      2  objDESFireEV2.selectApplication( i: 1 );
                                                // Authenticate to card key #1 (read key) in EV1 mode
      3  objDESFireEV2.authenticate( i: 1, IDESFireEV1.AuthType.AES,
                                                                KeyType.AES128, keyDataDefault );
         Log.d( TAG, msg: "Authentication card key #1 successful" );
                                                // Read from file #1 with key #1
      4  byte[] data = objDESFireEV2.readData( i: 1, i1: 0, i2: 16,
                                                 IDESFireEV1.CommunicationType.Enciphered, i3: 16 );
         Log.d( TAG, msg: "Read from file 1: " + Utilities.byteToHexString( data ) );
      }
      catch( Throwable t )
      {
         t.printStackTrace();
      }
   }
}
```

Figure 23. Authentication example for MIFARE DESFire EV1

The authentication must be repeated if a different key is used. An authentication creates a new communication session and this is unfortunately a costly operation. For the MIFARE DESFire EV2, an alternate approach exists. It was introduced with the new Secure Message mode of the MIFARE DESFire EV2. The method `authenticateEV2First()` is similar to the `authenticate()` method of the MIFARE DESFire EV1. But if a new key is used in the same application, then the user should use the method `authenticateEV2NonFirst()` which is less costly (and more efficient) because the current session is still used and only the cipher is updated. This approach is shown in snippet Figure 24.

```
objDESFireEV2.getReader().connect();
                                    //Read key for file #1
keyDefault = new SecretKeySpec( READ_KEY_AES_1, algorithm: "AES" );
keyDataDefault = new KeyData();
keyDataDefault.setKey( keyDefault );

objDESFireEV2.selectApplication( i: 1 );
                                    // Authenticate to card key #1 (read key) in EV2 mode
1 objDESFireEV2.authenticateEV2First( i: 1, keyDataDefault, byPcdCaps );
Log.d( TAG, msg: "Authentication card key #1 successful" );
                                    // Read from file #1 with key #1
2 byte[] data = objDESFireEV2.readData( i: 1, i1: 0, i2: 16,
                                        IDESFireEV1.CommunicationType.Enciphered, i3: 16 );
Log.d( TAG, msg: "Read from file 1: " + Utilities.byteToHexString( data ) );

keyDefault = new SecretKeySpec( READ_KEY_AES_2, algorithm: "AES" );
keyDataDefault = new KeyData();
keyDataDefault.setKey( keyDefault );
                                    // Authenticate to card key #2 (read key) in EV2 mode
3 objDESFireEV2.authenticateEV2NonFirst( i: 2, keyDataDefault );
Log.d( TAG, msg: "Authentication card key #2 successful" );
                                    // Read from file #2 with key #2
4 data = objDESFireEV2.readData( i: 2, i1: 0, i2: 16,
                                        IDESFireEV1.CommunicationType.Enciphered, i3: 16 );
Log.d( TAG, msg: "Read from file 2: " + Utilities.byteToHexString( data ) );
```

Figure 24. Authentication example for MIFARE DESFire EV2

Another important use case in a MIFARE DESFire personalization is the change of default keys at the end of the personalization. A user should never release a card with default keys for security reasons! The PICC Master Key is for a blank card all bytes zeros and 2K3DES cipher. This should be changed to AES cipher and a random key value. True random values as key values should always preferred not to give a security flaw with predicted values like FFFF…, 0123… etc. In this paper, we use “weak key values” only for clarity and to make the examples easier understandable.

Before a key can be changed, the changing instance must prove it has the permission for the change. This permission is the knowledge of the key to be changed! Therefore, before a key can be changed, an authentication to that key must be executed. Figure 24 shows a code snippet of the change of the PICC Master Key. To address the PICC Master key, a `selectApplication(0)` must be executed first. The old key is the default 2K3DES key and the new key is a AES. The TapLinx method `changeKey()` make the change of the cipher and the key value. Notice: `THREEDES` in `authenticate()` stands for 2K3DES cipher.

```
public static final byte[] DEFAULT_KEY_2K3DES =
{                                                   // Default key for MIFARE DESFire
   (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
   (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
   (byte)0x00, (byte)0x00
};

public static final byte[] NEW_KEY_AES =
{                                                   // New AES key
   (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff,
   (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff,
   (byte)0xff, (byte)0xff
};

private void cardLogic( final Intent intent )
{
   Key keyDefault, keyNew;
   KeyData keyDataDefault, keyDataNew;

   if( DESFireEV1 == m_libInstance.getCardType( intent ) )
   {
      IDESFireEV1 objDESFireEV1 = DESFireFactory.getInstance()
                                                .getDESFire( m_libInstance.getCustomModules() );
      try
      {
         objDESFireEV1.getReader().connect();
                                    // Address PICC Master Key
         objDESFireEV1.selectApplication( i: 0 );
         Log.d( TAG, msg: "PICC Master Key selected" );

         keyDefault = new SecretKeySpec( DEFAULT_KEY_2K3DES, algorithm: "DESede" );
         keyDataDefault = new KeyData();
         keyDataDefault.setKey( keyDefault );

         objDESFireEV1.authenticate( i: 0, IDESFireEV1.AuthType.Native,
                                                        KeyType.THREEDES, keyDataDefault );
         Log.d( TAG, msg: "Authenticated to default key" );

         objDESFireEV1.changeKey( i: 0, KeyType.AES128, DEFAULT_KEY_2K3DES,
                                                                  NEW_KEY_AES, (byte)0 );
         Log.d( TAG, msg: "Key changed" );
      }
      catch( Throwable t )
      {
         t.printStackTrace();
      }
   }
}
```

Figure 25. Change of key example for MIFARE DESFire EV1

6.3 Using a MIFARE Ultralight C

Often is a contactless card asked, where the price is the deciding factor. Such cards should be used in an area where typically paper tickets are used. For this area, the MIFARE Ultralight family is right option.

The MIFARE Ultralight has a memory structure very similar to the NTAG family. You find feature registers in the first memory addresses and following it, user memory registers. 4 bytes are grouped as “page” and is usually read and written in a single command. A memory area can be protected with a password and a successful authentication must be executed before a page in this area can be accessed. The code snippet in Figure 26 shows the authentication with the default password and a read and write of a single page of user memory on a MIFARE Ultralight C. Refer to [MF0ICU2] for a detailed data sheet and a description of all features.

```
public static final byte[] DEFAULT_KEY_ULTRALIGHT_C =
{                                                   // Default key for MIFARE Ultralight C
   (byte)0x49, (byte)0x45, (byte)0x4D, (byte)0x4B, (byte)0x41, (byte)0x45, (byte)0x52,
   (byte)0x42, (byte)0x21, (byte)0x4E, (byte)0x41, (byte)0x43, (byte)0x55, (byte)0x4F,
   (byte)0x59, (byte)0x46
};

private void cardLogic( final Intent intent )
{
   Key key;
   KeyData keyData;

   if( CardType.UltralightC == m_libInstance.getCardType( intent ) )
   {
      Log.d( TAG,  msg: "Ultralight C found" );
      IUltralightC objUlCardC = UltralightFactory.getInstance()
                                                 .getUltralightC( m_libInstance.getCustomModules() );
      try
      {
         objUlCardC.getReader().connect();
                                                   // Timeout to prevent exceptions
         objUlCardC.getReader().setTimeout( 2000 );
                                                   // MIFARE Ultralight C default key
         key = new SecretKeySpec( DEFAULT_KEY_ULTRALIGHT_C,  algorithm: "DESede" );
         keyData = new KeyData();
         keyData.setKey( key );

         objUlCardC.authenticate( keyData );
         Log.d( TAG,  msg: "Authenticated" );
                                                   // Write dummy  to page 4 (first user page)
         objUlCardC.write( i: 4, new byte[] { 0x01, 0x02, 0x03, 0x04 } );
         Log.d( TAG,  msg: "Data written" );
         byte[] data = objUlCardC.read( i: 4 );    // Read dummy data back
         Log.d( TAG,  msg: "Read data: " + Utilities.byteToHexString( data ) );
      }
      catch( Throwable t )
      {
         t.printStackTrace();
      }
   }
}
```

Figure 26. Authentication, reading and writing example for MIFARE Ultralight C

The code snippet in Figure 26 assumes that a MIFARE Ultralight C with factory configuration is used. The default password `KEY_ULTRALIGHT_C` is not all bytes zero! In a default configuration, all bytes can be read and a protect memory area must be set in the configuration registers. Anyhow, we show the authentication for demonstration purposes.

```
18255-18255/com.nxp.taplinx.taplinxtest D/MainActivity: Ultralight C found
18255-18255/com.nxp.taplinx.taplinxtest D/MainActivity: Authenticated
18255-18255/com.nxp.taplinx.taplinxtest D/MainActivity: Data written
18255-18255/com.nxp.taplinx.taplinxtest D/MainActivity: Read data: 010203040006011011FF000000000000
```

Figure 27. Log output of the code snippet

Figure 27 shows the log output. Take in mind, writing a page takes 4 bytes, reading a page gives 16 bytes back. So, the read buffer contains 4 consecutive pages.

6.4 Authentication and read from a MIFARE Plus EV1

The MIFARE Plus EV1 is the enhanced successor product of the MIFARE Classic. Therefore, the memory structure is inherited. However, the security architecture of the MIFARE Plus is different. Instead of using a proprietary cipher, AES with 128 bit is used! The MIFARE Plus can be used in different security levels, but we focus to security level SL3 only. For this security level, the keys A and B in the sector trailer of each block have no use anymore and can be used as data storage. The AES keys reside in a different memory area, from address 0x4000 upwards.

```
public static final byte[] NEW_KEY_AES =
{                                                   // New AES key
   (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff,
   (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff,
   (byte)0xff, (byte)0xff
};

private void cardLogic( final Intent intent )
{
   Key key;
   KeyData keyData;
   byte[] byData;

   if( CardType.PlusEV1SL3 == m_libInstance.getCardType( intent ) )
   {
      Log.d( TAG,  msg: "Plus EV1 found" );
      IPlusEV1SL3 objPlusSL3 = PlusFactory.getInstance()
                                          .getPlusEV1SL3( m_libInstance.getCustomModules() );
      try
      {
         objPlusSL3.getReader().connect();      // Timeout to prevent exceptions in authenticate
         objPlusSL3.getReader().setTimeout( 2000 );

         Log.d( TAG,  msg: "Card type: " + objPlusSL3.getType().getTagName() );
         Log.d( TAG,  msg: "ATQA: " + Utilities.dumpBytes( objPlusSL3.getCardDetails().atqa ) );
         Log.d( TAG,  msg: "SAK: " + Integer.toHexString( objPlusSL3.getCardDetails().sak ) );
                                                // Use key FFF...
         key = new SecretKeySpec( NEW_KEY_AES,  algorithm: "AES" );
         keyData = new KeyData();
         keyData.setKey( key );

         byte[] byPcdCap = new byte[] { 0 };
         objPlusSL3.authenticateFirst(  i: 0x4000, keyData, byPcdCap );
         Log.d( TAG,  msg: "authenticated to sector 0 (0x4000)" );
                                                // Read block 0 in sector 0
         byData = objPlusSL3.read(  IPlusSL3.ReadMode.Plain_ResponseNonMACed_CommandMACed,  i: 0 );
         Log.d( TAG,  msg: "Read block 0 (sector 0): " + Utilities.byteToHexString( byData ) );
                                                // Go to sector 1
         objPlusSL3.authenticateNonFirst(  i: 0x4002, keyData );
         Log.d( TAG,  msg: "authenticated to sector 1 (0x4002)" );
                                                // Read block 4 (sector 1)
         byData = objPlusSL3.read( IPlusSL3.ReadMode.Plain_ResponseNonMACed_CommandMACed,  i: 4 );
         Log.d( TAG,  msg: "Read block 4 (sector 1): " + Utilities.byteToHexString( byData ) );
      }
      catch( Throwable t )
      {
         t.printStackTrace();
      }
   }
}
```

Figure 28. Authentication, reading and example for MIFARE Plus EV1

The code snippet in Figure 28 shows the authentication to sector 0 with AES key 0x4000. We use Secure Messaging as introduced in Section 6.2, so `authenticateFirst()` is used. Then we continue to sector 1 by authenticate to key 0x4002 with `authenticateNonFirst()`. Take in mind, two keys control the access permissions of one sector. Keys 0x4000 and 0x4001, for instance, play the same role as keys A and B for the access permission for the MIFARE Classic. Refer to [MF1P(H)x1y1] for detailed information about the access permission.

```
11862-11862/com.nxp.taplinx.taplinxtest D/MainActivity: onNewIntent
11862-11862/com.nxp.taplinx.taplinxtest D/MainActivity: Plus EV1 found
11862-11862/com.nxp.taplinx.taplinxtest D/MainActivity: Card type: PlusEV1 SL3 X
11862-11862/com.nxp.taplinx.taplinxtest D/MainActivity: ATQA: 0x4400
11862-11862/com.nxp.taplinx.taplinxtest D/MainActivity: SAK: 20
11862-11862/com.nxp.taplinx.taplinxtest D/MainActivity: authenticated to sector 0 (0x4000)
11862-11862/com.nxp.taplinx.taplinxtest D/MainActivity: Read block 0 (sector 0): 04[redacted]80204400800000001316
11862-11862/com.nxp.taplinx.taplinxtest D/MainActivity: authenticated to sector 1 (0x4002)
11862-11862/com.nxp.taplinx.taplinxtest D/MainActivity: Read block 4 (sector 1): 00112233445566778899AABBCCDDEEFF
```

Figure 29. Log output of the code snippet above

One note about the key settings. At factory, the MIFARE Plus is in SL0 where keys must be defined. Usually the card is switched from SL0 to SL1. This is the “MIFARE Classic compatibility mode”. Before the switch from SL1 to SL3 can be done, some keys must be defined. This includes also the sector AES keys at address 0x4000. In our example, the default key has a value of 0xFFFF…FF. Take in mind that some distributors sell the MIFARE Plus EV1 “pre-configured” in SL3. In this case, the keys must be known to personalize the card properly. Refer [MF1P(H)x1y1] and AN3729 for a detailed description for security level switch and configuration.

Figure 29 shows the log output of the code snippet from Figure 28. As for the MIFARE Classic, also the MIFARE Plus EV1 carries the UID in block 0 of sector 0.

7 References

AN0945	MIFARE DESFire EV1 — Features and Hints, *Rev. 3.4 — 28 June 2011*
AN3387	Feature and Functionality Comparison between MIFARE DESFire EV1 and EV2, *Rev. 1.2 — 5 January 2016*
AN3630	MIFARE DESFire EV2 Features and Hints, *Rev. 1.3 — 28 May 2019*
AN3729	MIFARE Plus EV1 - Features and Hints, *Rev. 1.1 — 29 October 2019*
AS	Android Studio, Download: https://developer.android.com/studio
Eclipse	Eclipse IDE for Java Developers: https://www.eclipse.org/downloads/packages/
MF0ICU2	MIFARE Ultralight C - Contactless ticket IC, *Rev. 3.3 — 30 July 2019*, https://www.nxp.com/docs/en/data-sheet/MF0ICU2.pdf
MF1P(H)x1y1	MIFARE Plus EV1, *Rev. 3.2 — 6 December 2018*
MF1S50YYX_V1	MIFARE Classic EV1 1K — Mainstream contactless smart card IC for fast and easy solution development, *Rev. 3.2 — 23 May 2018*, https://www.nxp.com/docs/en/data-sheet/MF1S50YYX_V1.pdf
MF3Dx2	MIFARE DESFire EV2 contactless multi-application IC, *Rev. 3.1 — 25 April 2017*
MF3ICDx1	MIFARE DESFire EV1 — Functionality of implementations on smart card controllers, *Rev. 3.5 — 04 April 2016*
TagInfo	NFC TagInfo by NXP, *4.24.6 — 01 July 2020*, https://play.google.com/store/apps/details?id=com.nxp.taginfolite

8 Legal information

8.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

8.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors. In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory. Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification. Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products. NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer. In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages. Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

Translations — A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — While NXP Semiconductors has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP Semiconductors accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

8.3 Licenses

Purchase of NXP ICs with NFC technology

Purchase of an NXP Semiconductors IC that complies with one of the Near Field Communication (NFC) standards ISO/IEC 18092 and ISO/IEC 21481 does not convey an implied license under any patent right infringed by implementation of any of those standards. Purchase of NXP Semiconductors IC does not include a license to any NXP patent (or other IP right) covering combinations of those products with other products, whether hardware or software.

8.4 Trademarks

Notice: All referenced brands, product names, service names and trademarks are the property of their respective owners.

MIFARE — is a trademark of NXP B.V.

DESFire — is a trademark of NXP B.V.

ICODE and I-CODE — are trademarks of NXP B.V.

MIFARE Plus — is a trademark of NXP B.V.

MIFARE Ultralight — is a trademark of NXP B.V.

MIFARE Classic — is a trademark of NXP B.V.

NTAG — is a trademark of NXP B.V.

NXP — wordmark and logo are trademarks of NXP B.V.

Tables

Figures

Contents

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 1 September 2020
Document identifier: AN11876
Document number: 617120