UM2306
User manual

ST25 software development kit

Introduction

This document describes the main features of the ST25 software development kit (STSW-ST25SDK001, hereinafter referred to as ST25SDK), detailing its content, explaining its architecture and providing practical tips on how to use the Java™ library.

The ST25SDK can be used for Java™ applications interfacing with all ST25 NFC/RFID and dynamic tags and with all tags that support ISO 15693/14443 and NFC Forum Type 4 and 5 specifications.

Contents

List of tables

List of figures

1 List of acronyms and notational conventions

1.1 Acronyms

AAR	Android™ Application Record: a special NDEF Record, defined by NFC Forum and triggering the opening or installation of an Android application or Android™ ARchive: an Android™ library that can contain source code, resource files, and an Android™ manifest.
ISO	International Organization for Standardization
ISO/IEC 14443	Identification cards – contactless integrated circuit cards – proximity cards
ISO/IEC 15693	Identification cards – contactless integrated circuit cards – vicinity cards
NDEF	NFC Data Exchange Format
NFC	Near field communication
NFC Forum	Association of industry actors, promoting NFC technology
PICC (VICC)	Proximity (vicinity) IC Card. A technology subset defined in ISO/IEC standards for cards, with a defined set of commands.
RF	Radio Frequency
Tag	PICC in form of a patch, key fob and the like, without own power source and not generating RF electromagnetic field, capable of communicating with a reader / writer.
URI	Unified Resource Identifier

1.2 Representation of numbers

The following conventions and notations apply in this document unless otherwise stated:

- **Binary numbers** are represented by strings of 0 and 1 digits shown with the most significant bit (MSB) on the left, the least significant bit (LSB) on the right, and “0b” added at the beginning. Example: 0b11110101.
- **Hexadecimal numbers** are represented by using numbers 0 to 9 and characters A to F, and adding "0x" at the beginning. The Most Significant Byte (MSB) is shown on the left and the Least Significant Byte (LSB) on the right. Example: 0xF5.
- **Decimal numbers** are represented without any trailing character. Example: 245.

2 Overview

The ST25 software development kit is a set of tools aimed at accelerating the development process of Java™ applications based on ST RF tags.

The ST25SDK is divided in three main parts:

- a Java™ Archive (jar) library file that can be used in Android™ or cross-platform development environments (on Windows®, Linux® or macOS®), along with its documentation
- Java™ libraries and associated resources to support various RF readers:
  - Android™ phone RF reader
  - CR95HF and ST25R3911B-DISCO reference boards
  - Third party readers, such as FEIG ELECTRONIC (Feig) MR102 and LR1002 from the OBID i-scan® product line
- Simple example projects:
  - Android™ Studio for a smartphone app
  - Eclipse™ for a Windows® desktop program

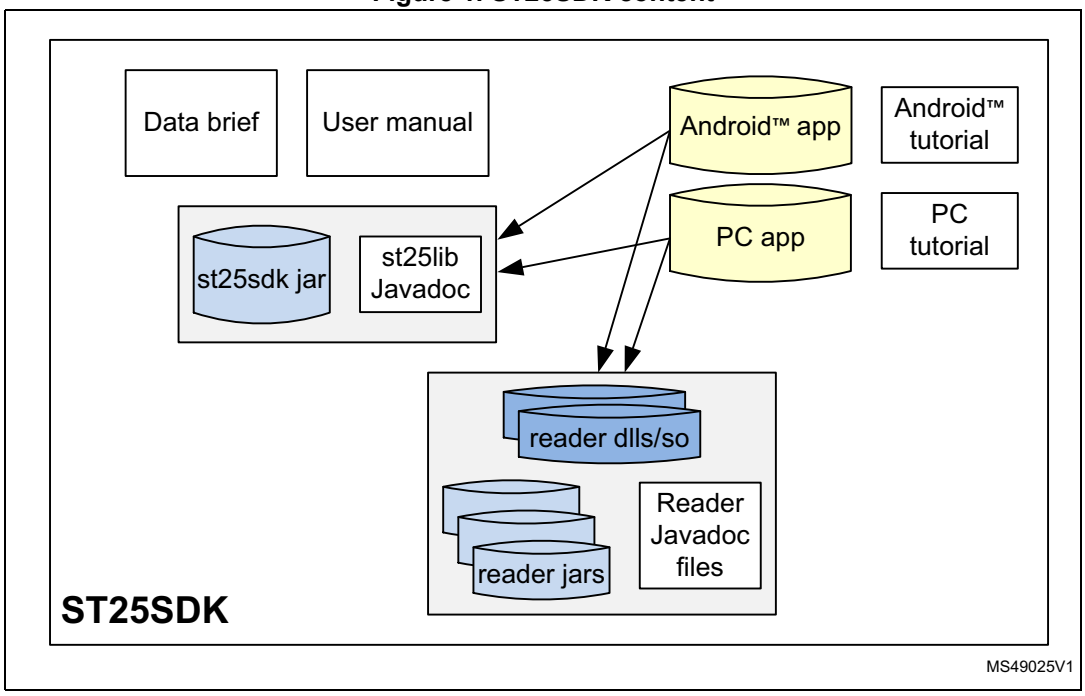
The main strengths of ST25SDK are:

- the libraries can run in any environment where a JVM (Java™ Virtual Machine) can be found
- a reader interface allows it to potentially communicate with any RF reader in the market.

3 Installation

3.1 SDK content

Figure 1. ST25SDK content



As shown in *Figure 1*, the kit is delivered as a zip file containing the following components:

- a .jar Java™ library file called st25sdk and its associated html documentation
- reader interface implementation classes for several RF readers and dynamic libraries
- native libraries for reader support (CRH95F, ST25R3911B-DISCO and some Feig models)
- basic applications source code
  - Android™ Java™ source code and executable
  - JavaFX source code and executable on Windows® 7/8 with support for CRH95F and ST25R3911B-DISCO

Table 1. ST25SDK file structure

Directory	Content
/Readme.txt	Readme file explaining the content and structure of the software package.
/st25sdk/lib/	Directory containing the ST25SDK library (jar file).
/st25sdk/documentation/	Javadoc documentation of ST25SDK library (contained in /st25sdk/lib/). It describes all the public APIs of the ST25SDK library.
/integration/android/lib/	Directory containing the Android™ Reader Interface AAR file, specific to Android™. When using an Android™ smartphone, the ST25SDK uses this reader interface to communicate with Android™ NFC API.
/integration/android/helper/	Directory containing some helper classes that can be used when building an Android™ NFC Application.
/integration/android/documentation/	Javadoc documentation of Android™ reader interface (available in /integration/android/lib/), and documentation of Android™ examples.
/integration/android/examples/	Some examples of Android™ projects based on the ST25SDK.
/integration/pc/documentation/	Documents for PC examples.
/integration/pc/examples/	Some examples of PC projects based on the ST25SDK.
/readers/lib	Java library implementing a generic RF reader interface.
/readers/feig/lib	Feig own Java library to drive their readers, and reader implementation for the st25sdk library.
/readers/feig/resources	Dynamic libraries from Feig.
/readers/feig/documentation	Javadoc for the Feig libraries.
/readers/st/lib/	Java™ library with ST reader implementations (demonstration boards based on CR95HF and ST25R3911B).
/readers/st/resources/	Dynamic libraries needed for ST readers.

3.2 Android™ installation

3.2.1 Installation of the Android™ development environment

Android™ Studio and ST25SDK tools must be installed, they can be downloaded from the dedicated webpage on https://developer.android.com.

An example of Android™ App is available in the current ST25SDK package (see integration/android/examples/ST25AndroidDemoApp). This is a basic application using the ST25SDK to:

- read some information about the tag (name, UID and memory size)
- write an NDEF message on the tag

3.2.2 Installation of ST25SDK libraries

Once the Android™ development environment is ready, the user can add the ST25SDK libraries and helper classes:

- “st25sdk-x.y.z.jar”: this is the library containing ST25SDK.
- “st25_android_reader_interface-a.b.c-release.aar”: this is an Android™ archive containing the Android™ Reader Interface. This reader interface is used by the ST25SDK to communicate with the NFC API of your Android™ phone.
- a dependency on the library 'org.apache.commons:commons-lang3:3.5': this library is used by the ST25SDK, no need to download and add this library manually. It can be mentioned in build.gradle, then gradle will automatically download it from JCenter.
- an helper class called “TagDiscovery.java”: this helper facilitates the discovery of the NFC tag. It executes some commands to identify the tag, then instantiates an object corresponding to the tag, and then notifies the application. The application can use the instantiated object to communicate with the tag.

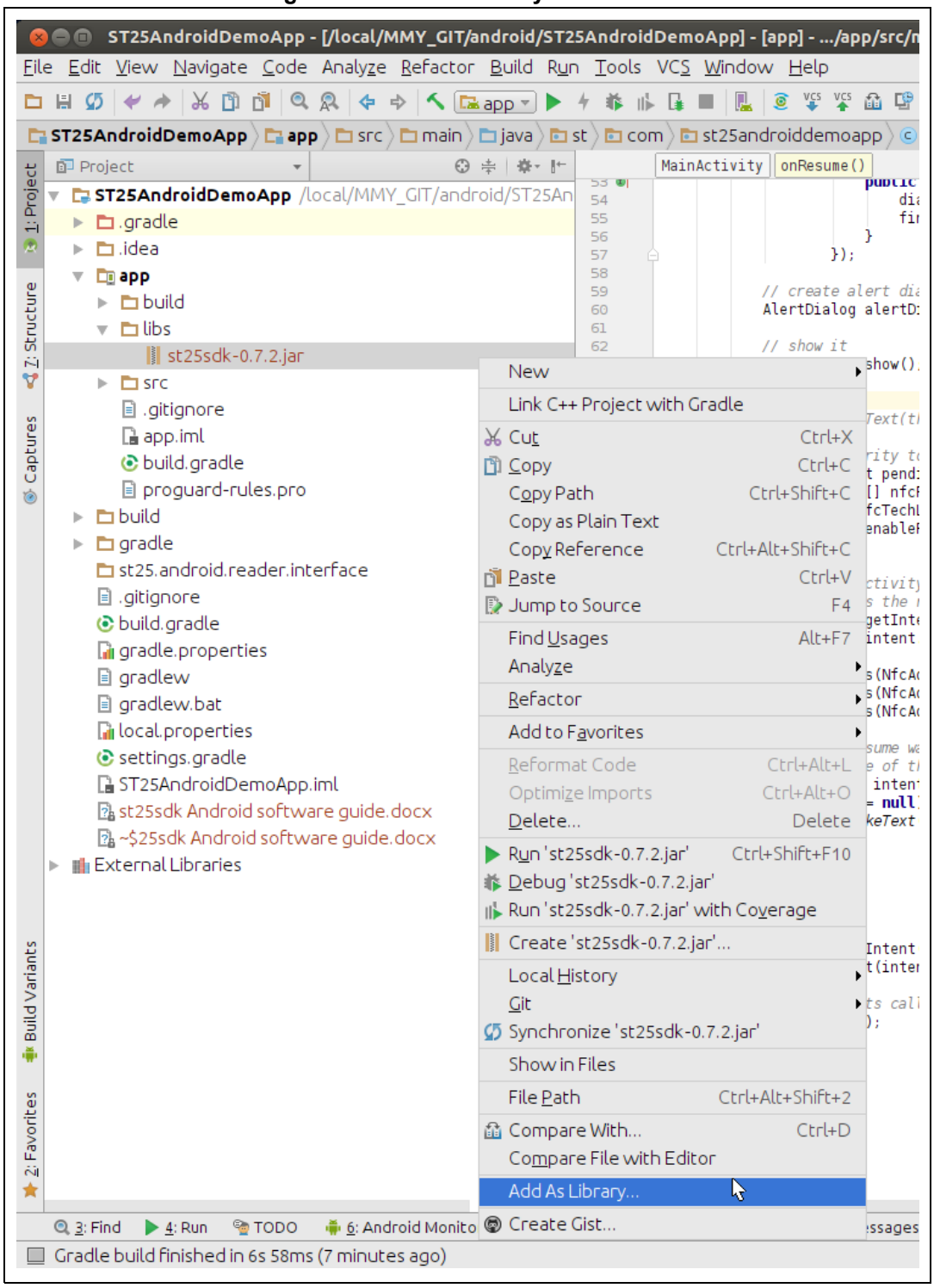
Installation of ST25SDK libraries

File *st25sdk-x.y.z.jar* is available in /st25sdk/lib/. It can be put directly in apps/libs directory of your application (create it if it is not present).

In the project explorer (on the left hand side), select the file “st25sdk-x.y.z.jar”, right click on it and click on “Add as library” (as shown in *Figure 2*).

This updates your build.gradle file to indicate this dependency.

Figure 2. ST25SDK library installation

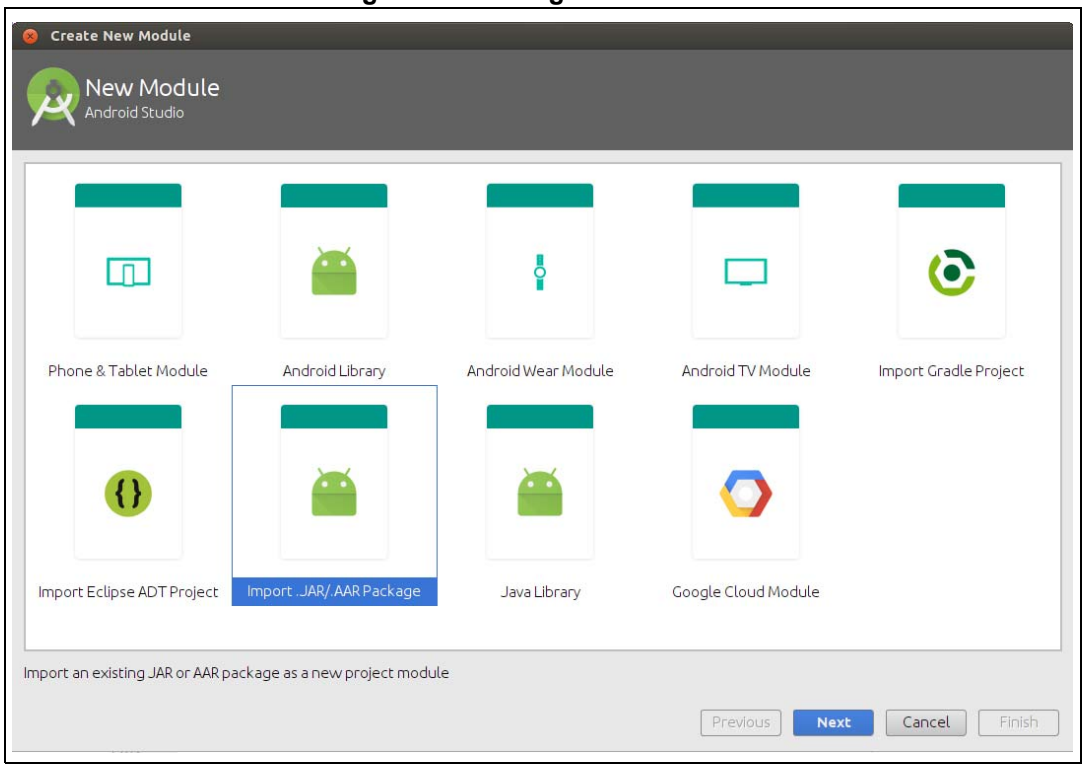


Installation of Android™ reader interface AAR

“st25_android_reader_interface-a.b.c-release.aar” is available in /integration/android/lib/.

In the menu, click on File → New → New module, select “import JAR/AAR package” and select the AAR file.

Figure 3. Creating a new module



This creates an “st25.android.reader.interface” module at the root of your project.

This module has its own gradle file. The files “settings.gradle” and “apps/build.gradle” are also updated to reference this new module.

Add dependency to 'org.apache.commons:commons-lang3:3.5'

The ST25SDK uses the library 'org.apache.commons:commons-lang3:3.5' so it should be indicated in the dependencies of your “apps/build.gradle” file

```
dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    // Needed by ST25 SDK
    compile 'org.apache.commons:commons-lang3:3.5'
}
```

Update MainActivity to call the TagDiscovery class

The helper file “TagDiscovery.java” is available in /integration/android/helper/

Add this class to one of your packages, for example to the one containing the “MainActivity.java” file.

The ST25AndroidDemoApp (in integration/android/examples/ST25AndroidDemoApp) contains an example showing how to use the TagDiscovery class.

onResume() of “MainActivity.java” is notified every times an NFC tag is tapped. This function can be executed for several reasons. From the intent, you can get the action that has triggered the resume of main activity and you can check if it is one of the following NFC events:

- ACTION_NDEF_DISCOVERED
- ACTION_TECH_DISCOVERED
- ACTION_TAG_DISCOVERED

```
Intent intent = getIntent();
String action = intent.getAction();
if (action.equals(NfcAdapter.ACTION_NDEF_DISCOVERED) ||
    action.equals(NfcAdapter.ACTION_TECH_DISCOVERED) ||
    action.equals(NfcAdapter.ACTION_TAG_DISCOVERED)) {
    // Your code here
}
```

In case of an NFC event, you can then add the following lines to start an AsyncTask that performs the tag discovery in a background thread

```
Intent intent = getIntent();
String action = intent.getAction();
if (action.equals(NfcAdapter.ACTION_NDEF_DISCOVERED) ||
    action.equals(NfcAdapter.ACTION_TECH_DISCOVERED) ||
    action.equals(NfcAdapter.ACTION_TAG_DISCOVERED)) {
    // If the resume was triggered by an NFC event, it will contain an
EXTRA_TAG providing
    // the handle of the NFC Tag
    Tag androidTag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    if (androidTag != null) {
        Toast.makeText(this, "Starting Tag discovery",
Toast.LENGTH_SHORT).show();
        // This action will be done in an Asynchronous task.
        // onTagDiscoveryCompleted() of current activity is called when
discovery is completed.
        new TagDiscovery(this).execute(androidTag);
        }
    }
```

This code identifies what kind of tag has been tapped (Type 4, Type 5, etc) and allocates the appropriate ST25SDK Tag object. For example, when tapping an ST25DV64K tag, an ST25DVtag is instantiated.

Note: Tag discovery should be done in an AsyncTask because the Android UI thread should not be blocked.

A Listener can be defined in your MainActivity, it is called when the tag discovery is completed

```
public class MainActivity extends AppCompatActivity implements
TagDiscovery.onTagDiscoveryCompletedListener {
```

```
    …
    @Override
    public void onTagDiscoveryCompleted(NFCTag nfcTag, TagHelper.ProductID
productId) {
        if (nfcTag != null) {
            String tagName = nfcTag.getName();
            Toast.makeText(this, "Tag discovery done. Found tag: " +
tagName, Toast.LENGTH_LONG).show();
        } else {
            Toast.makeText(this, "Tag discovery failed!",
Toast.LENGTH_LONG).show();
        }
    }
}
```

An nfcTag object is now available, it can be used to communicate with the tag.

3.3 Installation in a PC environment

In addition to Android, the ST25 library can be used as a building block in PC programs. As the library is in jar format, you can build applications where a JVM can run (on Windows®, Linux® and macOS®).

A step-by-step user guide for a basic PC application can be found in the ST25SDK examples folder (integration\pc\documentation\ST25PcDemoApp_software_guide.pdf). The example project is written for Windows® in Eclipse™, but the library can be used in any development environment (such as NetBeans and IntelliJ, or operating systems supporting the Java™ Run-time Environment).

4 ST25 library architecture overview

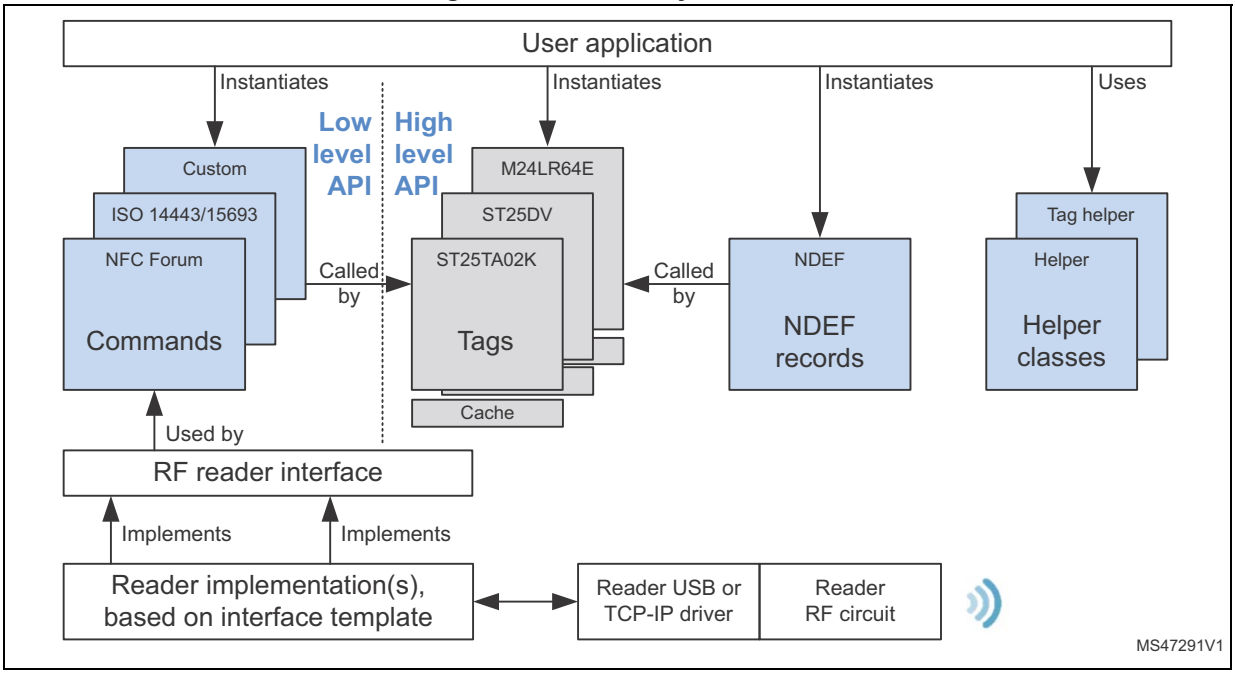
Architecture and design choices for the ST25 library have the following targets:

- Object oriented API
- Reusable code for all kinds of applications
- Cross platform / cross reader solution
- Data caching

The library API (Application Programming Interface) is divided in three levels (see *Figure 4*):

- High level Tags API
  - Provides the highest level of abstraction, making it easy to use the tag specific features
  - Checks method parameters and throws an exception if illegal values are used
  - Generic NFCTag, Type4Tag and Type5Tag classes
  - Specific ST25DVTag, LRiTag classes for example
- Low level Commands API
  - Very close to standards and tag datasheets
  - Almost no parameter verification is done, so can be used by test applications to check the behavior of the tag when sending RF commands with any parameter, including incorrect ones
  - Useful for multi-tag industrial/commercial applications to communicate with several tags at once
- Helper classes
  - For data type conversion and tag identification

Figure 4. ST25 library overview



An RF reader Interface allows the ST25 library to abstract the interactions with an RF reader. Some reader interface implementations are delivered with the ST25SDK for:

- Android™ smartphones
- ST readers (CR95HF and ST25R3911B-DISCO boards)
- a subset of Feig OBID i-scan® devices (MR102 and LR1002)

4.1 Application Programming Interfaces

4.1.1 The Tags API

For applications targeting a single tag (e.g. smartphone apps), the natural API to use is the high level Tags API. As shown in *Figure 5*, the developer can instantiate a tag object for a generic Type 4 or Type 5 tag, or a specific tag class.

All tags inherit from the NFCTag class that defines methods common to all (see ST25SDK library javadoc for a complete list).

A tag object maps the RF commands from the relevant ISO and NFC protocols plus all other custom commands specific to STMicroelectronics products. This mapping calls the methods defined in the Commands API.

The files of the Tags API are contained in several packages:

- com.st.st25sdk for NFCTag.java
- com.st.st25sdk.type4a for NFC Forum Type 4A and ISO14443-A tags
- com.st.st25sdk.type5 for NFC Forum Type 5 and ISO15693 tags
- com.st.st25sdk.iso14443sr for ISO14443B-SR tags

4.1.2 The Commands API

The Commands API is a set of low-level classes used to send unitary RF commands and receive responses. The user sends commands from a reader to a single or to multiple tags (for Type 5 commands, by defining the value of the request flags). The files of the Commands API are contained in the com.st.st25sdk.command package.

The Commands method prototypes are as close as possible to their definitions in the ISO or NFC Forum specifications and to the behavior described in ST datasheets.

Take as example of the NFC Forum Type 5 / ISO15693 *readSingleBlock()* prototype

```
public  byte[] readSingleBlock(byte blockAddress, byte flag, byte[] uid)
throws STException;
```

The user can see that:

- it returns a byte array, corresponding to the tag response as transmitted by the reader. (this means that the ISO15693 response flag byte is included as the first element of the byte array)
- the method parameters follow the types defined in the ISO command

There is no check on the method values in this API (some checks are performed in the Tags API). Hence, the ST25SDK user can really send any value of flag, block address and UID, even those not recognized by the tags.

4.1.3 The Helper classes

The Helper class contains methods used throughout the ST25SDK library to convert data types (for example to convert byte arrays to hexadecimal strings) or to perform operations on byte arrays (concatenate, reverse, and other types).

The TagHelper class can be used to identify ST tags from their UID.

Helper and TagHelper classes are located in package com.st.st25sdk.

4.1.4 The reader interface

The RF reader interface is a contract between the ST25SDK library and all reader classes. It ensures that all readers implement the same command set, making the library reader-independent.

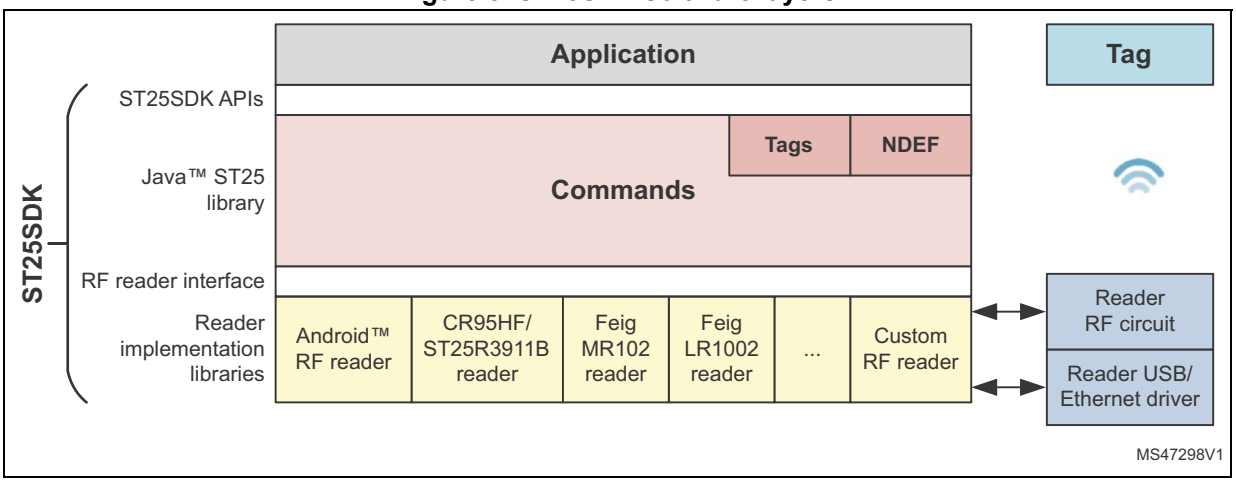
The ST25SDK contains several reader interface implementations, along with the needed resource files for ST CR95HF and ST25R3911B-DISCO boards, and for Feig MR102 and LR1002.

The RFReaderInterface.java file is located in package com.st.st25sdk.

4.2 The layer view

Figure 5 takes the flow of *Figure 4* and rearranges it in software layers.

Figure 5. ST25SDK software layers



The Application layer uses the ST25SDK library through the Commands, Tags or Helper classes. In turn, the APIs call methods defined in the RF reader interface to send commands to specific reader implementations.

The Java™ implementation then makes native calls to dynamic libraries that communicate with the reader firmware via USB, Ethernet or other protocols.

In the case of Android™ reader interface, the commands are transmitted to the Android™ NFC API (which in turn calls some Native Interface commands to communicate with the NFC controller present on the smartphone).

The reader then sends the Radio Frequency commands to the target tag and sends the response back to the reader interface via the *transceive()* method. This information then moves up the stack up to the Application layer.

5 Code examples

5.1 High level Tag API

All the following examples assume that you have instantiated an NFCTag object corresponding to the kind of tag you want to communicate with.

For example, when using an ST25DV64K tag, you should instantiate an ST25DVTag object. If you want to use an M24SR64K tag, you should instantiate an M24SR64KTag object.

All those tag objects extend the NFCTag class.

5.1.1 Read/write CCFile

The NFCTag class has some methods to interact with the tag CC File

```
public abstract byte[] readCCFile() throws STException;
public abstract void writeCCFile() throws STException;
public abstract void initEmptyCCFile() throws STException;
public abstract int getCCFileLength() throws STException;
public abstract byte getCCMagicNumber() throws STException;
public abstract byte getCCMappingVersion() throws STException;
public abstract byte getCCReadAccess() throws STException;
public abstract byte getCCWriteAccess() throws STException;
public abstract int getCCMemorySize() throws STException;
```

These methods are abstract because each tag class has its own implementation.

If a method is not available on one kind of tag, a NOT_SUPPORTED STException is raised. This is the case for *writeCCFile()* on Type 4 tags, because CC File of Type 4 tag cannot be written.

Here is an example showing how to retrieve the current CC File of a tag

```
NFCTag tag;
byte[] ccFile = tag.readCCFile();
```

And here is how to write a default CC File (corresponding to the tag type)

```
tag.initEmptyCCFile();
tag.writeCCFile();
```

Note: When calling the method writeNdefMessage(), the ST25SDK checks if a valid CC File is present. If not, it writes one, so most of the times you do not have to write the CC File yourself.

5.1.2 Read/write NDEF

Here is an example showing how to instantiate a NDEF message and write it on the tag. This NDEF message contains the URI Record “http://www.st.com/st25“

```
// Create a NDEFMsg
NDEFMsg ndefMsg = new NDEFMsg();

// Create a URI record containing http://www.st.com
```

```
UriRecord uriRecord = new UriRecord(NDEF_RTD_URI_ID_HTTP_WWW,
"st.com/st25");

// Add the record to the NDEFMsg
ndefMsg.addRecord(uriRecord);

// Write the NDEFMsg into the tag
mNfcTag.writeNdefMessage(ndefMsg);
```

If no STException happens, it means that the Write was successful.

In your application, you must be able to handle a possible STException. Your application should catch the STExceptions and decide on the behavior in case of error, that is, trying again, informing the user of the problem, requesting a password, or other options.

An example of how to catch an STException is given in *Type 5 password management*.

The code to read the NDEF message present in a tag is even simpler

```
NDEFMsg ndefMsgRead = tag.readNdefMessage();
```

5.1.3 Read/write data blocks

The previous paragraph explains how to read or write NDEF formatted data. It is also possible to read or write some proprietary data (in binary format) through the following API

```
public abstract byte[] readBytes(int byteAddress, int sizeInBytes) throws
STException;
public abstract void writeBytes(int byteAddress, byte[] data) throws
STException;
```

Here is an example showing how to read six bytes from the byte 23

```
int startAddress = 23;
int numberOfBytes = 6;
Byte[] data = tag.readBytes(startAddress, numberOfBytes);
```

And here is how to write some binary data to the byte address 0x123

```
byte[] dataToWrite = new byte[] {0x01, 0x02, 0x03, 0x04};
int address = 0x123;
tag.writeBytes(address, dataToWrite);
```

A clarification about Type 5 tags

Type 5 tags have an architecture organized in “blocks”. A block is frequently a group of four bytes (but it can be different), the size is given by the CC File and getSystemInfo command.

For Type 5 tag, the following functions are defined in Type5Tag class and allow to read or write some blocks

```
public ReadBlockResult readBlocks(int firstBlockAddress, int sizeInBlocks)
throws STException;
public void writeBlocks(int firstBlockAddress, byte[] data) throws
STException;
```

The principle is the same except that:

- the addresses are block addresses
- the data byte array contains an integer number of blocks

5.1.4 Examples of ST-specific features

Multi-area

Several STMicroelectronics tags implement a multi-area feature. Those tags can be configured to have one or more independent memory areas.

The interface “MultiAreaInterface” defines some methods to use the multi area feature.

For example you can use *getNumberOfAreas()* to retrieve the current number of areas and *getMaxNumberOfAreas()* to get the maximum number of areas possible on this tag.

Having some independent areas makes it possible to set some different Read and Write permissions for each of them. For example, the user may want to split the tag memory in two areas, the first one with read and write access allowed for everybody and the second one protected by password for read and write access (for example to store private information).

Areas are numbered from 1 to N, where is N is the current number of areas.

The first area (Area 1) is the one containing the NDEF File described by NFC Forum. Generic functions like *readNdefMessage()* and *writeNdefMessage()* are reading or writing into Area 1.

Consequently, it is completely equivalent to call

```
tag.readNdefMessage()
```

or

```
int area = 1;
tag.readNdefMessage(area)
```

In both cases, the content of Area 1 is read.

Note: Type 4 tags use the concept of files. Each file is identified by a FileId. The file should be selecting before doing any action on it.

STMicroelectronics M24SR tags are able to support several independent files. Those files have the fieIds 0x0001, 0x0002, 0x0003 and so on.

The M24SR tag implement the MultiAreaInterface: each area is mapped to a specific file. For example, a request to r*eadBytes()* in Area 3 leads to a selection of file 0x0003, and a reading of its content.

```
int area = 3
byte[] data = tag.readBytes(area);
```

Type 5 password management

When reading or writing data, it can happen that the tag is protected in Read or in Write. In this case an STException occurs.

Here is an example with the function *readNdefMessage()*

```
try {
    int area = 1;
    NDEFMsg ndefMsg = myTag.readNdefMessage(area);
} catch (STException e) {
    switch (e.getError()) {
    case TAG_NOT_IN_THE_FIELD:
```

```
            // The tag is no more in the field
            return TAG_NOT_IN_THE_FIELD;
        case INVALID_CCFILE:
            // The CCFile contained in this tag is invalid
            break;
        case INVALID_NDEF_DATA:
        // The NDEF file contained in this tag is invalid
        break;
        case WRONG_SECURITY_STATUS:
            // This Exception will happen with Type4 tags if the tag is
protected in Read
            break;
        case ISO15693_BLOCK_PROTECTED:
            // This Exception will happen with Type5 tags if the tag is
protected in Read
            break;
        default:
            // Another error occured
            e.printStackTrace();
        }
}
```

If the STException ISO15693_BLOCK_PROTECTED happens on a Type 5 tag, it means that this area of the tag is protected in Read.

The code above allows the application to catch the STException and to decide what to do.

In case of area protected in Read by password, the application should call the following function to know the ‘passwordNumber’ protecting this memory area

```
int passwordNumber = type5Tag.getPasswordNumber(area);
```

The application can then display a popup indicating that this area of the tag is protected in Read, and asking to enter the area password.

When the user has typed the password, it should be presented to the tag through the function *presentPassword()*

```
byte[] password;
int passwordNumber = type5Tag.getPasswordNumber(area);
tag.presentPassword(passwordNumber, password);
```

If this function is successful (that is, no STException is raised), the application can try to read the NDEF message again

```
NDEFMsg ndefMsg = myTag.readNdefMessage(area);
```

This time, the read should be successful because the read access has been unlocked.

The same scheme is used with the function *writeNdefMessage()* when a memory area is protected in Write.

Type 4 passwords

On Type 4 tags, the STException WRONG_SECURITY_STATUS occurs if a file is protected in Read or in Write. With Type 4 tags, it is not sufficient to do

```
tag.presentPassword(area, password);
NDEFMsg ndefMsg = myTag.readNdefMessage(area);
```

because the following actions should be done step-by-step:

- select the appropriate file (corresponding to this area)
- present the password
- read the NDEF message

As a work around, another *readNdefMessage()* function has been defined, it has one additional argument providing the read password

```
byte[] readPassword;
NDEFMsg ndefMsg = stType4tag.readNdefMessage(fileId, readPassword)
```

Here too, the same scheme is used for *writeNdefMessage()*. A function enables to select the file, write the password and write an NDEF message

```
NDEFMsg ndefMsg;
byte[] writePassword;
stType4tag.writeNdefMessage(fileId, ndefMsg, writePassword)
```

5.2 Low level Command API

The Command API interacts directly with a reader interface rather than with a tag. This gives a larger scope of operation, the user can now send commands to several tags at once.

Here is an example of an ISO15693 discovery command

```
// Create new cr95/3911b reader object from native library
STReader myReader = new STReader();
myReader.connect();

// Retrieve low level RF interface for that reader
RFReaderInterface readerInterface = myReader.getTransceiveInterface();
Iso15693Command iso15693Cmd = new Iso15693Command(readerInterface, null);
byte[] uid = iso15693Cmd.inventory()[0];  // First element from inventory
```

In the example above, once you have retrieved a reader interface as defined in RFReaderInterface, you can instantiate an ISO15693 Command object containing all RF commands contained in the ISO15693-3 specifications.

Note: The user can optionally give an UID value to the ISO15693 constructor. This parameter must be dereferenced if it is needed by a command (for example, in readSingleBlock() if the Addressed Mode bit of the flag parameter is set).

5.3 TagHelper API

The code excerpt below gives an example of the TagHelper class usage

```
// Create new reader object from native library
STReader myReader = new STReader();
myReader.connect();
```

```
// Retrieve low level RF interface for that reader
RFReaderInterface readerInterface = myReader.getTransceiveInterface();
NfcTagTypes tagType = readerInterface.decodeTagType(uid);
NFCTag recognizedNFCTag;
ProductID productName;
// Call the TagHelper method corresponding to the tag type
if (tagType == NfcTagTypes.NFC_TAG_TYPE_V) {
    productName = TagHelper.identifyTypeVProduct(readerInterface, uid);
} else if (tagType == NfcTagTypes.NFC_TAG_TYPE_4A) {
    productName = TagHelper.identifyType4Product(readerInterface, uid);
} else {
    productName = TagHelper.identifyProduct(readerInterface, uid);
}
switch (productName) {
    case PRODUCT_ST_ST25DV04K_I:
    case PRODUCT_ST_ST25DV04K_J:
    case PRODUCT_ST_ST25DV16K_I:
    case PRODUCT_ST_ST25DV16K_J:
    case PRODUCT_ST_ST25DV64K_I:
    case PRODUCT_ST_ST25DV64K_J:
        recognizedNFCTag = new ST25DVTag(readerInterface, uid);
        recognizedNFCTag.setName(productName.toString());
        break;
…
    case PRODUCT_ST_ST25TA02K_D:
        recognizedNFCTag = new ST25TA02KDTag(readerInterface, uid);
        break;
…
    case PRODUCT_GENERIC_TYPE5:
        // Non ST or unrecognized Type 5 products
        recognizedNFCTag = new Type5Tag(readerInterface, uid);
        recognizedNFCTag.setName(productName.toString());
        break;
}
```

In this example, the TagHelper method has been used to identify an ST tag based on its UID. The ProductID values indicate the tag that should be instantiated in order to benefit from all the specific features.

6 Reader support

The RF reader interface defines the methods to be implemented by a reader in order to work correctly with the ST25SDK library.

The ST25SDK contains a few ready-to-use implementations.

6.1 Android™ reader interface

The Android™ Reader Interface is mapped to Android™ NFC API, it can be found in the integration/android/lib/ directory.

The implementation differs depending of the tag technology:

- Type 4A tags will be mapped to Android NfcA API
- Type 5 tags will be mapped to Android NfcV API

In both cases, the main function is the *transceive()* function. The *transceive()* function of the Reader Interface is routed to the *transceive()* function of NfcA or NfcV API.

6.2 Reader interfaces provided for ST reference readers

The ST25SDK can easily operate with STMicroelectronics reader devices, available on CR95HF and ST25R3911B-DISCO boards.

The control of these NFC readers is performed by dedicated libraries:

- cr95.dll
- st25r3911.dll

The ST25SDK also features a Java™ Native Interface, named streader.dll, used to route and translate the requests from the Java™ program to the expected reader library.

From a Java™ program, the methods of this JNI can be accessed through two different classes:

- STReaderTransceiveImplementation: for the methods defined in the RFReaderInterface (see *Section 4.1.4: The reader interface*), the most important functions being:
  - send data to the tags
  - receive their responses
  - run the inventory process
- STReader: for all other required methods, such as reader connection, disconnection or configuration, and getting information from the reader.

6.3 Third party reader support

Any reader can be supported, as long as an RFReaderInterface implementation for the ST25SDK and a JNI to communicate with the reader native libraries exist.

6.3.1 Feig reader support

The ST25SDK includes libraries to support readers of the Feig OBID i-scan® reader line, among them MR102 and LR1002.

Resource files delivered in the ST25SDK correspond to the files found in Feig SDK ID_ISC.SDK.Java-V4.7.0:

- readers\feig\lib\OBIDISC4J.jar is the Java library, refer to Feig documentation for instructions
- readers\feig\documentation\OBIDISC4J Javadoc is the Javadoc created by Feig for the OBIDISC4J library
- readers\feig\resources is the directory containing run-time libraries

Go to readers\feig\lib\st25pc-model-readers-feig.jar for the RFReaderInterface implementation. This library needs to be in your project build path if you intend to use Feig readers with the st25sdk.jar library.

To use the Feig reader support in your project, make sure to add a link to the OBIDISC4J.jar and st25pc-model-readers-feig.jar libraries in your Java Build path.

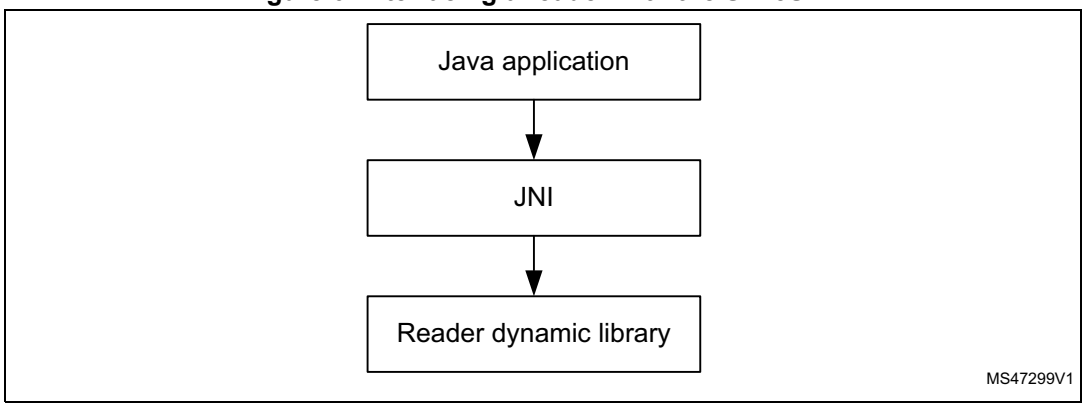
Also, check that the run-time libraries path in feig\resources appears in the native library location of your project at execution time.

6.4 Implementing the reader interface

The ST25SDK makes it possible to efficiently build an application for NFC, especially when using a smartphone, some Feig readers or an STMicroelectronics reader. It is also possible to use the ST25SDK with other reader hardwares.

Readers often come with a DLL or a Shared Object library, to manage the communication between the reader and an application running on the computer. To leverage such libraries from a Java™ application, it is required to implement a Java™ Native Interface layer (JNI). The ST25SDK provides built-in JNIs to control CR95HF- and ST25R3911B-based readers through a USB connection.

Figure 6. Interfacing a reader with the ST25SDK



6.4.1 Implementing your own reader interface

To use a JNI, the application has first to declare some native methods, such as

```
public class ReaderAPI {
    public native boolean connect();
}
```

Note that the class implementing the native methods should also load the JNI library in its constructor, as, for example

```
public ReaderAPI() {
    System.load(myReaderJni);
}
```

A user can then run the JAVAH utility from the JDK, providing in arguments the qualified java path to the class declaring the native methods.

```
${JDK}/bin/javah.exe com.myApp.ReaderAPI
```

The JAVAH utility will generate a C/C++ header file defining, for each declaration of a native Java™ method, a unique C/C++ function prototype, such as

```
JNIEXPORT jboolean JNICALL
Java_com_myApp_ReaderAPI_connect(JNIEnv*,jobject);
```

This function prototype uses JNI-specific types that are defined in the jni.h header file, located in the JDK include directory (to be included when building the JNI). This function has two interesting parameters:

- The JNIEnv parameter is a pointer to a structure providing the Java™ JNI functions. It can be used from native code to call the JVM, for example to create new Java™ objects.
- The jobject parameter is a reference to the Java™ object inside which the native method is declared.

As the last step, the function must be implemented calling the reader library when required, for instance

```
JNIEXPORT jboolean JNICALL Java_com_myApp_readerAPI_connect(JNIEnv*
env,jobject obj) {
   // this is a call to the Reader DLL
   connectMyReader();
}
```

6.4.2 Services to implement in the reader JNI

The ST25SDK has an interface called RFReaderInterface, defining basic methods required by the ST25SDK to interact physically with the tags (see *Section 4.1.4: The reader interface*). Some of the methods defined in this interface (like transceive or inventory), must call (or be themselves) native methods.

However, in order to use an external reader, the application requires some other methods to connect/disconnect, configure or get info from the reader. These services too are achieved through the reader JNI.

Table 2 is a shortlist of the usually required methods in a reader JNI.

Table 2. Reader JNI methods

Method	Description
Connect	Enables the communication with the reader.
Disconnect	Disables the communication with the reader.
CheckConnection	Verifies that the communication is still alive.
GetName	Differentiates when several readers are available.
GetLibraryVersion	Verifies compliance between the JNI version and the native library version.
GetReaderVersion	Verifies compliance between the native library version and the reader hardware/firmware version.
ConfigureMode	Updates reader settings for a particular ISO/NFC protocol.
GetMaxTxLength(1)	Limits the amount of data sent to the reader.
GetMaxRxLength(1)	Allocates enough memory to receive data from the reader.
Inventory(1)	Manages the anticollision process (often performed more efficiently by the reader itself than the application).
Transceive(1)	Sends commands to the tags and to receive responses.

1. Required by the ST25SDK RFReaderInterface.

7 Annex

7.1 Useful links

http://www.st.com/en/nfc/st25-nfc-rfid-tags-readers.html is the main entry point concerning STMicroelectronics ST25 NFC products, as it contains all ST25 resources:

- product documentation
- ST25SDK download (package containing the libraries, the documentation and some demo applications)
- ST25 tools and applications (Android™ APK, PC App Installer).
- samples and Discovery boards

An Android™ Application, built over the ST25SDK, is available for download on Android™ PlayStore. This application shows the features of STMicroelectronics ST25 NFC tags.

7.2 Android™ execution context

We have seen how to create a simple NDEF message in a few lines of code.

Here is an example of NDEF message containing the URI http://www.st.com/st25

```
// Create a NDEFMsg
NDEFMsg ndefMsg = new NDEFMsg();

// Create a URI record containing http://www.st.com
UriRecord uriRecord = new UriRecord(NDEF_RTD_URI_ID_HTTP_WWW,
"st.com/st25");

// Add the record to the NDEFMsg
ndefMsg.addRecord(uriRecord);

// Write the NDEFMsg into the tag
mNfcTag.writeNdefMessage(ndefMsg);
```

The call of *writeNdefMessage()* is going to do some “transceive” to communicate with the NFC tag. On Android, those transceives should not be executed in the UI Thread context because they may take some time, so they would block the UI thread and create visible interferences.

To avoid this issue, use an AsyncTask

```
/**
 * Async Task writing a NDEF message into the tag
 */
private class asyncTaskWriteUriNdefMessage extends AsyncTask<Void, Void,
ActionStatus> {
    public asyncTaskWriteUriNdefMessage() {
    }

    @Override
```

```
    protected ActionStatus doInBackground(Void... param) {
        ActionStatus result;
        try {
            // Create a NDEFMsg
            NDEFMsg ndefMsg = new NDEFMsg();

            // Create a URI record containing http://www.st.com
            UriRecord uriRecord = new UriRecord(NDEF_RTD_URI_ID_HTTP_WWW,
"st.com/st25");

            // Add the record to the NDEFMsg
            ndefMsg.addRecord(uriRecord);

            // Write the NDEFMsg into the tag
            mNfcTag.writeNdefMessage(ndefMsg);

            // If we arrive here, it means that no STException occured so
the write was successful
            result = ActionStatus.ACTION_SUCCESSFUL;

        } catch (STException e) {
            switch (e.getError()) {
                case TAG_NOT_IN_THE_FIELD:
                    result = ActionStatus.TAG_NOT_IN_THE_FIELD;
                    break;
                default:
                    e.printStackTrace();
                    result = ActionStatus.ACTION_FAILED;
                    break;
            }
        }
        return result;
    }

    @Override
    protected void onPostExecute(ActionStatus actionStatus) {
        switch(actionStatus) {
            case ACTION_SUCCESSFUL:
                Toast.makeText(MainActivity.this, "Write successful",
Toast.LENGTH_LONG).show();
                break;
            case ACTION_FAILED:
                Toast.makeText(MainActivity.this, "Write failed!",
Toast.LENGTH_LONG).show();
                break;
            case TAG_NOT_IN_THE_FIELD:
```

```
                Toast.makeText(MainActivity.this, "Tag not in the field!",
Toast.LENGTH_LONG).show();
                break;
            }
            return;
        }
    }
```

The function *doInBackground()* is executed by a background thread. It will do the requested work, that means create a NDEF message and write it into the tag. At the end, it will return a status to indicate if the action was successful or not.

onPostExecute() is then executed to do the post processing. A typical action is to display a message for the user to inform him if the action was successful or not. It is very important to note that this function is called in the UI Thread context so it is safe to call or update some UI elements (e.g. display an alert, or show a toast message). Here only a toast message is displayed to indicate if the write NDEF message action was successful.

8 Revision history

Table 3. Document revision history

Date	Revision	Changes
15-Nov-2017	1	Initial release.