life.augmented

AN4430 Application note

Designing an NFC Android Application for M24SR and M24LR devices

Introduction

This application note explains how to build an Android application to be associated with STMicroelectronics NFC products belonging to M24SR and MLR series.

This application note is a guide to implement a customer dedicated android application to exchange data between android devices and ST NFC Dynamic tag or ST NFC transceiver; it’s mostly based on literature accessible from web and more specifically from Android developer web site.

Before demonstrating how to implement a simple NFC android application, a short Android Application overview is done by highlighting main Android application concepts. In a second part an introduction to NFC Application reader is done by using the NFC android API. The last section explains how to implement android solution to use enhanced features from ST products.

Contents

List of tables

List of figures

Glossary

IDE: Integrated Development environment

ADT: Android Development Tools

NFC: Near Field Communication

APK: android application package

ADB: Android Debug Bridge

AAPT: Android Asset Packaging Tool

UI: User Interface

RTD: Record Type Definition

RATS: Request For Answer To Select

PPS: Protocol and Parameter Selection

Reference Documents

ST-M24SR products datasheets from www.st.com

ST-M24LR products datasheets from www.st.com

Android reference web site

ISO/IEC 7816-4: Identification cards — Integrated circuit Cards - Organization, security and commands for interchange

IS0/IEC 14443: Identification cards -- Contactless integrated circuit cards

IS0/IEC 15693: Contactless integrated circuit cards -- Vicinity cards

1 Android Application Overview

1.1 About Android Operating System

“Android is an operating system based on the Linux kernel, and designed primarily for touchscreen mobile devices such as smartphones and tablet computers. Initially developed by Android, Inc., which Google backed financially and later bought in 2005, Android was unveiled in 2007 along with the founding of the Open Handset Alliance: a consortium of hardware, software, and telecommunication companies devoted to advancing open standards for mobile devices. The first publicly available smartphone running Android, the HTC Dream, was released on October 22, 2008.”(a)

1.2 Android System: Component architecture view

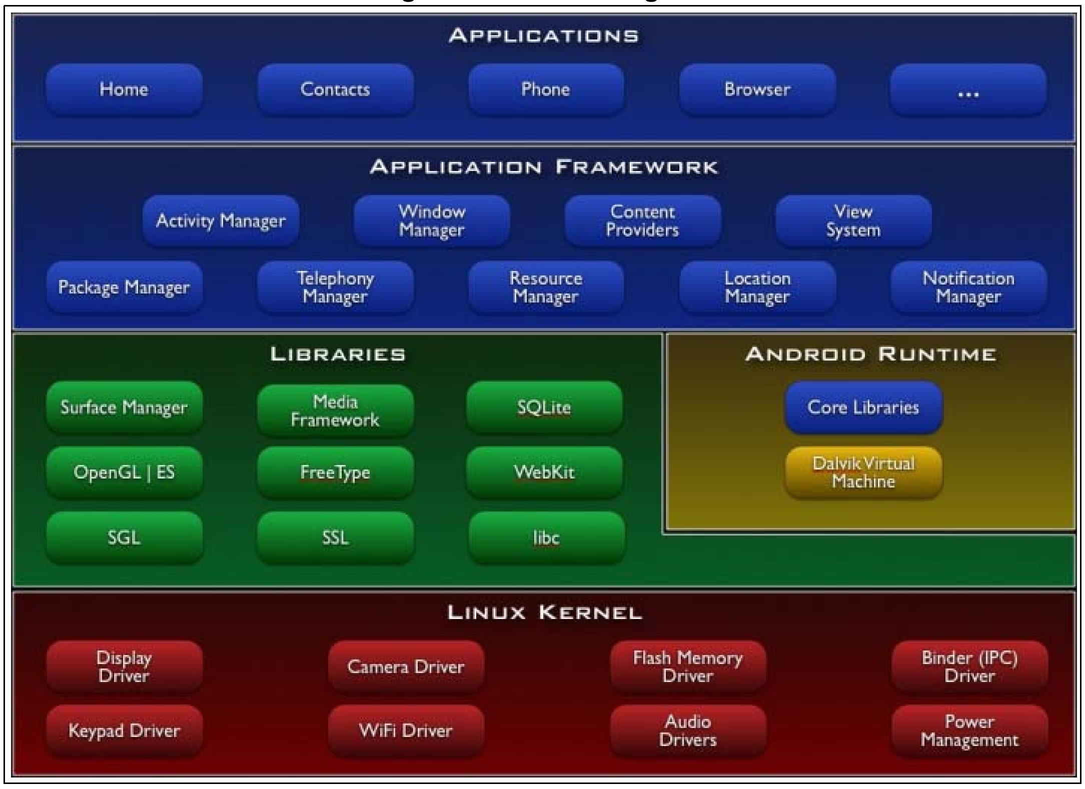
1.2.1 Android architecture overview

Basically android has the following layers (see *Figure 1*)

- Application written in java executing in Dalvik – (the aim of the current application note)
- Framework services and libraries written mostly in java
- Native library, daemons and services written in C or C++
- The Linux kernel which includes as every operating system:
  - Drivers for hardware,
  - Networking,
  - File system
  - Inter-process-communication

a. Definition from Wikipedia

Figure 1. Android diagram



NFC features can easily be mapped on this architecture.

NFC phone has to be NFC capable with an HW NFC chips controlled by a NFC driver provided by the chip constructor. This driver is running in the Linux kernel with super user rights. Upon this driver a NFC stack also provided by the chip constructor or by the phone constructor located in libraries layers ensures interfacing between driver low level layer and android stack which implements the android API in application framework and more specifically the NFC Android API overviewed later on in this application note.

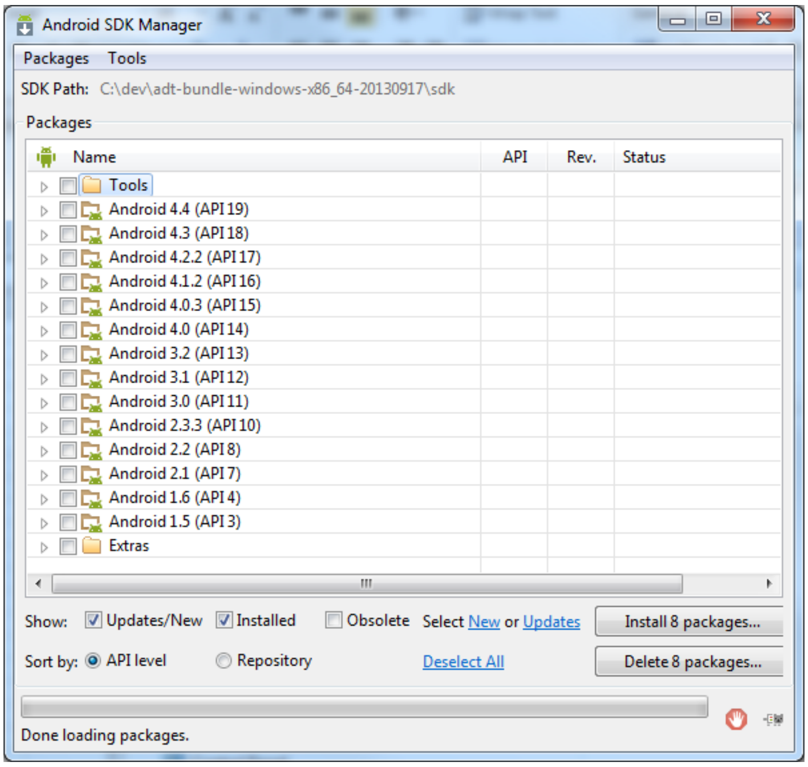
1.2.2 Android API

Android API is the upper software layer exposed to the developer. This API is based on the core Java APIs and consists of a set of packages defining classes that ease the use of android feature and the device components. This API is built with multiple packages like:

- Widget – UI components,
- Telephony – cell network, call, GSM, CDMA
- Media – audio, video
- Content – content providers,
- Database – SQLLite
- XML – SAX, pull parser
- Hardware – camera, sensor, usb, nfc
- ..

As Android is in constant development phase, packages are still improved by the android consortium, new features are added. Therefore, as some feature appears, API version defined a set of available features, meaning that some features are not available for lower API revision. As example NFC functionality appears at API level 9 restricted to NDEF message access and has been improved since this API revision. Figure 2 Android revision with corresponding API levels shows the whole android revision and the associated API levels.

Figure 2. Android revision with corresponding API levels



Android SDK Manager

Packages Tools

SDK Path: C:\dev\adt-bundle-windows-x86_64-20130917\sdk

Packages

Name	API	Rev.	Status
Tools			
Android 4.4 (API 19)			
Android 4.3 (API 18)			
Android 4.2.2 (API 17)			
Android 4.1.2 (API 16)			
Android 4.0.3 (API 15)			
Android 4.0 (API 14)			
Android 3.2 (API 13)			
Android 3.1 (API 12)			
Android 3.0 (API 11)			
Android 2.3.3 (API 10)			
Android 2.2 (API 8)			
Android 2.1 (API 7)			
Android 1.6 (API 4)			
Android 1.5 (API 3)			
Extras			

Show: [✓ Updates/New] [✓ Installed] Obsolete Select New or Updates Install 8 packages...

Sort by: [✓ API level] Repository Deselect All Delete 8 packages...

Done loading packages.

1.3 Android Application Introduction

1.3.1 Application fundamentals

Android application is primarily written in java programming language. Source code is converted to java class files by the Java compiler. Then, Android SDK converts these Java classes into optimized Dalvik executable file(s) (.dex suffix). Using the AAPT tool, Dex files and application project related resources are packaged into an APK: android package, which is an archive file with an ".apk" suffix. The APK generated file self contains all the contents of an android application. The resulting APK file can be deployed to an Android device by either the ADB tool in developing step or by the google android store once the final application is published in release mode. The APK package is used by the Android powered devices to install the application with a unique user and group ID. Each application file is private to this generated user and other applications cannot access this file.

When launched by the operating system, the application runs, in its own process, upon its own Dalvik virtual machine and is totally isolated from other running applications. In this way, each application has access only to the components that it requires to do its work. However, applications can share data between themselves either by giving the same Linux user ID to the application which needs to share data like a file, or by requesting permission to access device data such as camera's data, SMS data, and so on. Data request is done via an Android component that handles the sharing of the data as a service or a content provider. The permissions are granted at the installation time by the android permission system. Permission can be automatically granted, rejected or asked to the user. By this way the android system implements the principle of least privilege.

1.3.2 Application components

Android applications are written using application components. Application Android strategy is to be able to share application resources among all application installed on an Android device. In this way application can publish its resources so they can be used by other applications. Android system allows to automatically instantiate the Java object owner of the desired component. This approach implies in a first hand that application simply request to Android system to start the desired component within the application that contain it, and in the second hand Android applications don't have an single entry point but rather get components that the system can instantiate on demand.

Android components can be categorized in 4 main families:

- Activities
- Services
- Broadcast receivers
- Content providers

The 3 last families are shortly described in this application note as they are note deeply used to demonstrate the NFC implementation use case.

Those components are asynchronously activated by messages called intents. On intent, Android system finds the right component to respond to it and instantiates it if necessary by a request from a content resolver method call.

1.3.3 Intents and Intent filters

Intent overview

Intent is a messaging object used to request an action from a component to other one and then to facilitate communication between components. Intents are used on Android System to:

- Start an activity
- Start a service
- Deliver a broadcast

Structure of intent implicitly defines the two intent types:

- Explicit intent: specifies the component to start by name. Typically use to start a component from the same application as the component caller.
- Implicit intent: declares a general action to perform without giving a component to start. Caller doesn't know the name of the capable component. Android system is then in charge to find the appropriate component able to answer the caller request. Android system looks for intent listed in the component's intent filters declares in the manifest file. If several components are identified, Android System displays a selectable list to let the final user to decide which component to use.

Intent filters

Intent filters are xml structured definition stored with the component declaration in the android manifest file (detailed later on in this application note). This structure is attached to a component definition and declares the intent(s) the component is able to receive to perform action. By this way, user makes possible for other application to start an activity component with a defined intent. Likewise, if component activity declaration doesn’t have an intent filter declaration the activity is only start from its application owner.

Activities

Activity is a component belonging to an application and described in the android manifest of the application it belongs to. The activity may provide a (partial) screen with which the user can interact with. Application is usually composed by several activities that are loosely bound to each other. One activity is declared as the main activity which launched on the application start. This main property is declared in the application manifest.xml file.

Each activity from the started application can start other activity, which present a new screen to the user. Every time a new activity is started it is put in a stack and the previous one put in stopped state while the Android system keeps the activity in a stack abiding to the basic “last in, first out” stack mechanism also called back stack. When the current activity is finished (action ended, back button pressed) it is popped from the back stack and destroy, the activity on the top of the back stack is then resumed.

Activities life cycle

Activity life cycle is handled with callback methods that the system calls when the activity transition between various states. Each activity developer implements must inherit for Activity class (or an existing subclass of it).

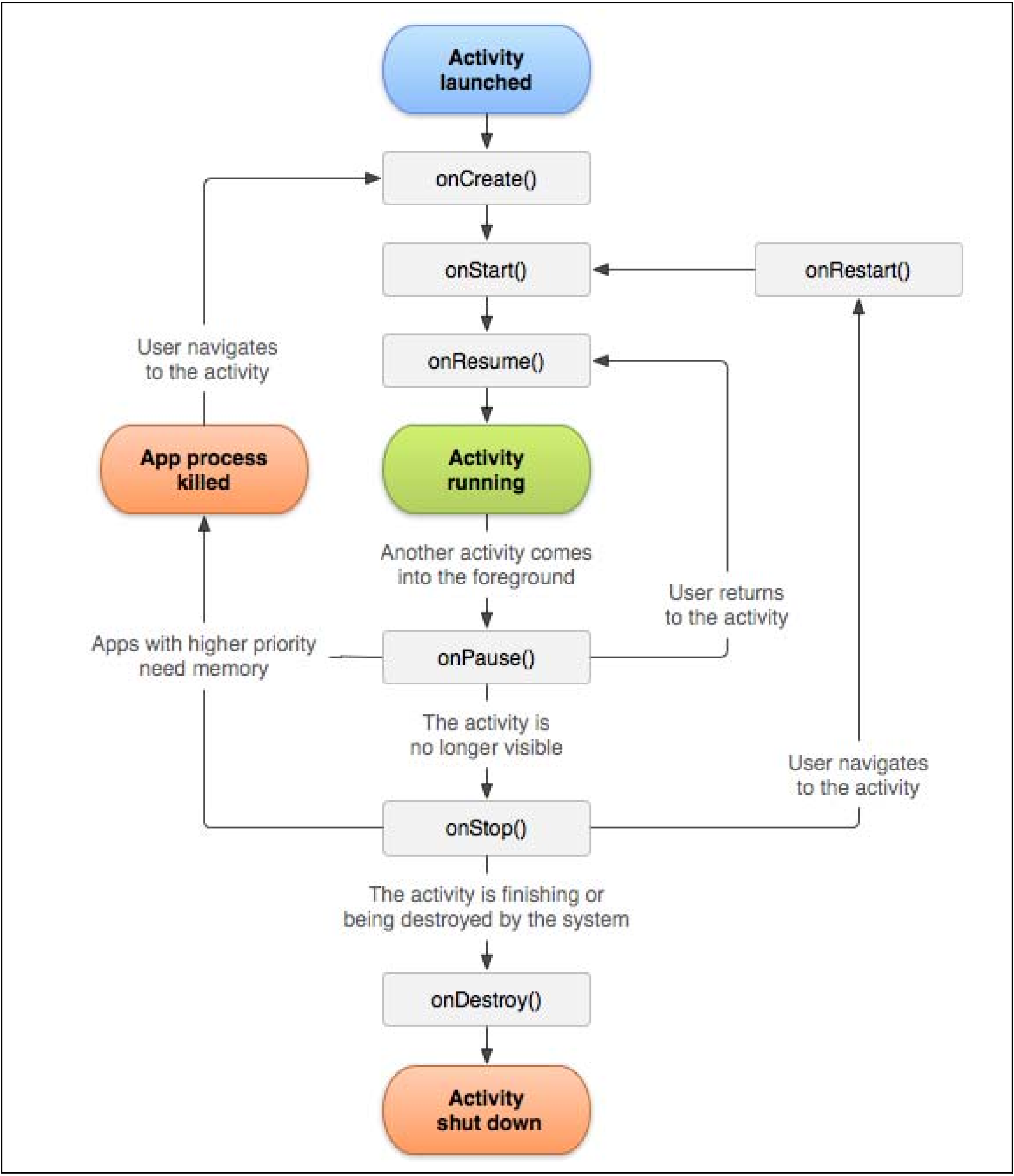
Figure 3. Activity Java skeleton

```
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // The activity is being created.
    }
    @Override
    protected void onStart() {
        super.onStart();
        // The activity is about to become visible.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // The activity has become visible (it is now "resumed").
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Another activity is taking focus (this activity is about to be
"paused").
    }
    @Override
    protected void onStop() {
        super.onStop();
        // The activity is no longer visible (it is now "stopped")
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // The activity is about to be destroyed.
    }
}
```

Figure 4 is an activity skeleton to show which callback method set has to be implemented to handle the fundamental lifecycles methods. Taken together, those callbacks define the entire life cycle of an activity managed by the android system. The same figure shows that developer can manage three different nested loops in the activity life cycle

- Entire lifetime: Bound within the *onCreate()* and *onDestroy()* calls, activity should setup or release resources on *onCreate()* and *onDestroy()* system call respectively.
- Visible lifetime: Bound within *onStart()* and *onStop()* calls, user can see the activity on-screen and the required resources by the activity have to be maintains.
- Foreground lifetime: Bound within the *onResume()* and *onPause()* calls, user can interact with the activity's screen.

Figure 4. Activity life cycle


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Activity screen

Activity can directly managed is own graphic interface by implementing graphical object in a static way. To provide more flexibility, android propose a dedicated graphic object called Fragment. Fragment represents a behavior or a portion of graphic user interface in an activity. Multiple fragments can be combined in a single activity to facilitate reuse, screen adaption. Fragment is always embedded in an activity and follows the activity lifecycle.

Broadcast receiver

Broadcast receiver is a component in charge to respond to the system-wide broadcast announcement like a screen turn off message, a low battery message and so on.

Services

Services are application components which perform long-running operation in background without providing any user interface. Service runs while the application is still alive independently the user switches to other activity (from the same application) to other application.

Content providers

Due to security system used by Android system, Content Providers is the component class in charge to provide mechanisms for defining data security. They are standard interface to ensure data sharing between components which not run in the same process. Application can manipulate its own set of data and store it in file system, SQLite data base or external storage like SD. Other application can access and modify this set of data by querying it through the content provided implemented by the data owner application.

Content provider is a part of an application and is in charge to centralized data access from other foreign components. It can also provide its own UI for working with the data it provides. To get deeper understanding on content provider, reader can refer itself to literature from web and more specifically on ‘calendar provider’ and ‘contact provider’ from Android developer web site.

Application Manifest

Associated to every android application, developer must join an xml application descriptor file call AndroidManifest.xml. This file located in the application root directory gives a Java package name for the application used as a unique identifier for the application. Manifest file lists the permission the application requires to do its job and the minimum android API level the application requires to be fully functional. It also describes components (ie. Services, activity, broadcast receiver, content provider) belonging to the application. Each component declaration gets a java class name which implements the component, a set of properties like a list of intent messages handled by the component.

Structure of the manifest file

Manifest file is a hierarchy file as show in General structure manifest figure where child hierarchy must be followed. Element at the same level is generally not ordered.

As manifest file is a key file in android application conception it’s recommended the reader refers to the android manifest documentation.

Figure 5. General structure manifest

```
<?xml version="1.0" encoding="utf-8"?>

<manifest>

    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />

    <application>

        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </activity-alias>

        <service>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <receiver>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </receiver>

        <provider>
            <grant-uri-permission />
            <meta-data />
            <path-permission />
        </provider>

        <uses-library />

    </application>

</manifest>
```

2 How to create an Android Application for ST dual memories

2.1 Development environment

In order to implement an android application, developer has to setup his own development environment to be able to edit the source code, build it with the right tool chain, and flash the application to the embedded targeted system (i.e. the mobile phone running the android operating system)

2.1.1 Prerequisites

Operating system computer from where the application is generated must belong to the following families:

- Windows XP (32 bits), Windows Vista (32 to 64 bits), Windows 7 (32 to 64 bits)
- Mac OS X 10.5.8 or later (restricted to x86 solutions)
- Ubuntu (8.04 or later) Linux (with GNU C lib 2.7 or later)

Development environment:

- Eclipse 3.6.2 (Helios) or greater with included Eclipse JDT plugin
- Java Development Kit 6
- Android SDK: Provides to the android application developers the Android API libraries and tools necessary to build, test, trace and debug applications. Here is the list of the main repositories in android SDK:
  - SDK Tools: contains tools for debugging and testing.
  - SDK Platforms tools: contains platform-dependent tools
  - Documentation: offline copy of the android platforms APIs downloaded on developer's station.
  - SDK Platforms: store the SDK platform (for each version of android) – SDK platforms can be downloaded on demand by the developer.
  - System Images: system images (such as for ARM, x86,..) derivate by each android version to let developer to test his application on a specific android system using an emulator.
  - Source code for android SDK: source code of each android version (to be downloaded on demand by the developer according to the android version he's working with.
  - Samples for SDK: collection of samples provided by Google.
  - Google APIs: SDK add-on that provides both a platform and a system image.
- Android development tools (ADT) plugin (recommendation to fasten developments steps): This plugin has been designed to give a powerful, integrated environment in which to build android applications by extending the capabilities of Eclipse (IDE).

It's highly recommended to use eclipse with ADT installed as this solution is the fastest way to get started. This solution eases the task to setup a new application project, to build the final application in debug or release mode. During the pure development activity developers are also supported by various integration tools and enhanced XML editor.

For more information about how to install and use eclipse or ADT according to its own development station, reader can refer to Android developer web site.

2.2 NFC Android API

From Android point of view, NFC is a set of short-range wireless technologies requiring a distance of 4 cm or less to initiate a connection. User can exchange small payload of data between NFC capable android phone and an external solution (in our cast ST-M24SR or ST-M24LR products). Many of the android APIs are based around the NFC Forum standard called NDEF (NFC Data Exchange Format). To get further information on NDEF specification reader as to refer itself NDEF specification publicly accessible from NFC Forum web site.

Android API is then able to handle 3 modes of operation:

- Reader/writer mode – mode treated in this application note
- P2P mode – not currently addressed in this application
- Card emulation mode allowing the NFC device to act as an NFC Card and be addressed by an external NFC reader (not treat in the current application note).

2.2.1 Android.nfc Package

This package which appears with the API 9 and enhanced by the next API revision gives the access to the Near Field Communication functionality on NFC capable Android. This package helps developer to easily read or write NDEF messages on an external NDEF compliant NFC Tag.

Main class of the Android.nfc package

- *NfcAdapter*

  This class maps the local android phone's NFC adapter. NfcAdapter is then the entry point to perform NFC operations. Developer has to call the *getDefaultAdapter()* to retrieve the default NFC adapter.
- *NfcManager*

  This class offers the NFC high level manager service used to obtain the instance of the NfcAdapter. The instance can be retrieved by the call of *getSystemService(String)* with *NFC_SERVICE* string given in parameter.
- *NdefMessage* and *NdefRecord*

  NdefMessage class represents the NDEF data message object implementation while *NdefRecord* class represents the NDEF records carrying the NDEF message. Those classes can be mapped on the NDEF NFC Forum specification. To construct a NDEF message developer has to use the *NdefMessage([byte])* call from a binary data that is parsed by android API on *NdefMessage* creation. If the developer expects to use typed data he may call *NdefMessage(NderRecord*, *NdefRecords*,….). *NdefRecord* object is created according to type of record message developer wants to store in NDefMessage (ie: RTD_TEXT, RTD_URI, RTD_SMART_POSTER, ..). RTD (Record Type Definition) are specified by the NFC Forum and can be retrieving from their web site.
- *NfcEvent*

  Object used to wraps information associated with an NFC event usually included in callbacks from *NfcAdapter. NfcEvent* must be preferred to parameters in the

NfcAdapter's callback in order to keep backward compatibility as fields and parameter may be added in the next API releases.

- Tag

  This class implements the Tag that has been discovered (when a Tag is presented to the android phone's NFC field). This object is immutable as it represents the state of the Tag at the time it has been discovered. This object is used to retrieve the tag ID (*getID()* call) and the properties of the Tag like the supported technologies (*getTecList()* call).

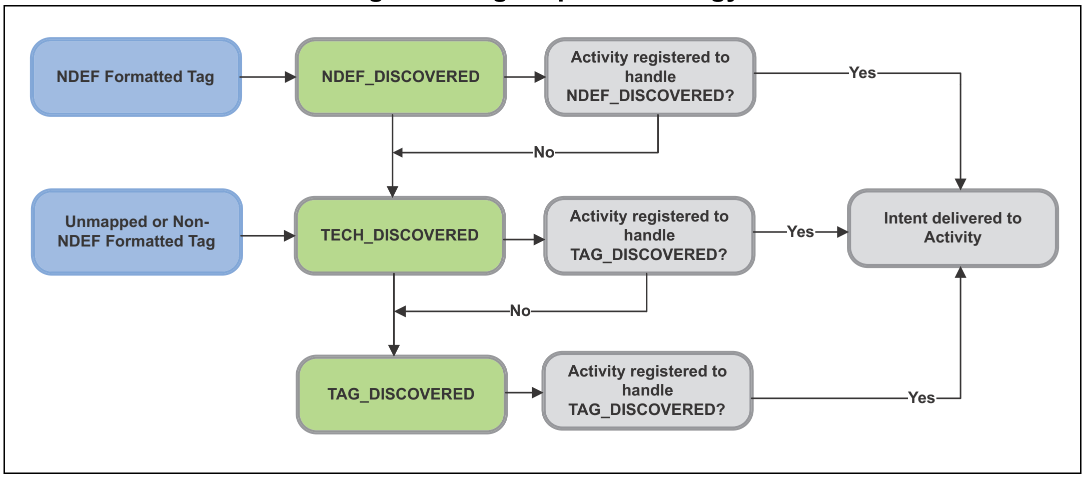
A new tag object is created every time a tag is discovered even if it is the same physical tag. Developer must then ensure to always use the latest tag discovered to perform specific action like TagTechnology interface manipulation.

When a tag is discovered, android system creates a tag object, store it in an intent message and requests for a startActivity with the created intent in parameter.

Depending on the tag characteristics and on the installed applications a four stage dispatch solution is used to select the most appropriate activity to handle the current detected tag (see Figure 6 Tag Dispatch Startegy). Android system executes each stage in order and complete dispatch as soon as a single matching activity is found.

- Stage 1 – Foreground Activity dispatch: a foreground activity which has called *NfcAdapter.enableForegroundDispatch(…)* is given the priority. This call is generally done in the *onResume()* activity state callback. To release this priority developer has to call *NfcAdapter.disableForegroundDispatch(Activity)* generally in *Pause()* activity state callback.
- Stage 2 – NDEF data dispatch: If the discovered tag contains at least an NDEF message with a first Record with URI, SmartPoster or MimeData type Android system call a *startActivity()* with ACTION_NDEF_DISCOVERED intent. Android System will then look for a component ACTION_NDEF_DISCOVERED handling capable. If most that one component is find, Android System requests to user to select the application/component to launch to treat the pending intent. If none component is register to handle this kind of intent dispatch move to stage 3.
- Stage 3 – Tag Technology dispatch: Android System call a *startActivity()* with ACTION_TECH_DISCOVERED and look for a component capable to handle current tagTechnology. If any component is register to handle at least one technology supported by the tag Android System goes to Stage 4.
- Stage 4 – Fall-back dispatch: Android System call a *startActivity()* with ACTION_TAG_DISCOVERED and look for a component capable to handle current intent.

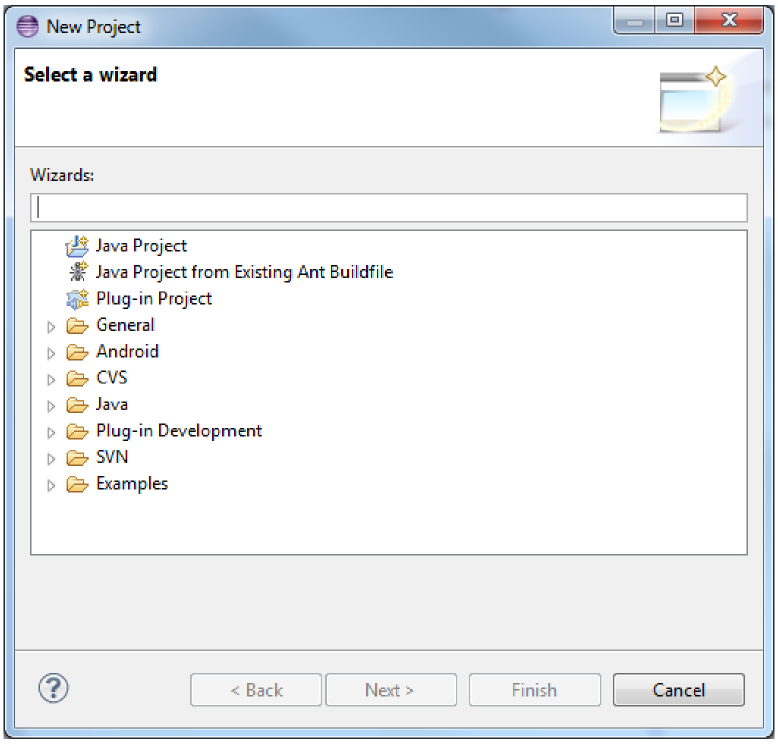
Figure 6. Tag Dispatch Strategy


NDEF Formatted Tag
NDEF_DISCOVERED
Activity registered to handle NDEF_DISCOVERED?
Yes
No
Unmapped or Non-NDEF Formatted Tag
TECH_DISCOVERED
Activity registered to handle TAG_DISCOVERED?
Yes
Intent delivered to Activity
No
TAG_DISCOVERED
Activity registered to handle TAG_DISCOVERED?
Yes

2.3 Create a new project with eclipse

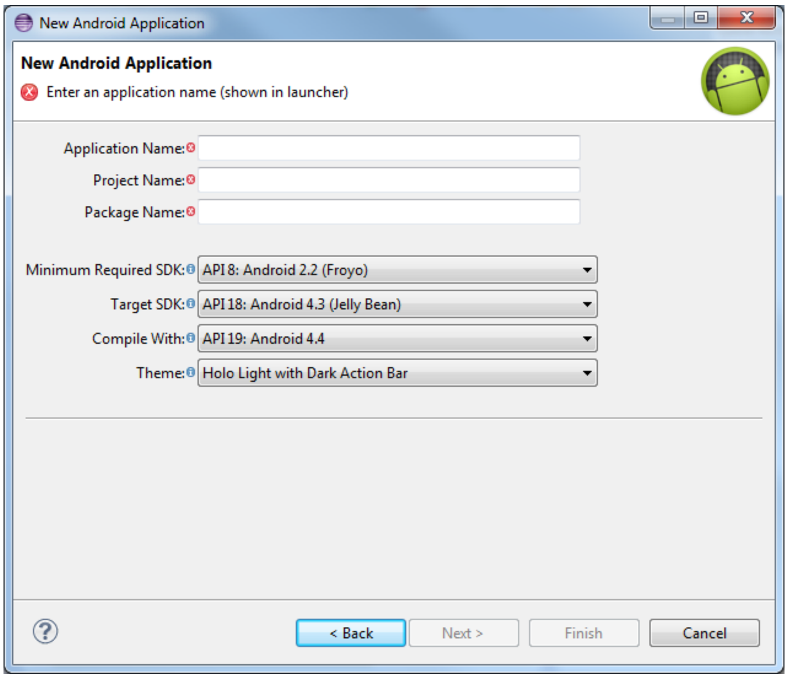
Once the development environment is installed (eclipse with ADT and Android SDK) developer has to create his project by starting Eclipse IDE and select File/New/Project to open the suitable create project wizard (Figure 7 Start Project Wizard

Figure 7. Start Project Wizard



Then select Android/Android Application project and click on Next button to open the "New Android Application" wizard (*Figure 8*). In the text edit widget ApplicationName enter myFirstNFCApp. "Project Name" and "Package name" text edits are automatically filled.

Figure 8. New Android Application Wizard


New Android Application

New Android Application

Enter an application name (shown in launcher)

Application Name:

Project Name:

Package Name:

Minimum Required SDK: API 8: Android 2.2 (Froyo)

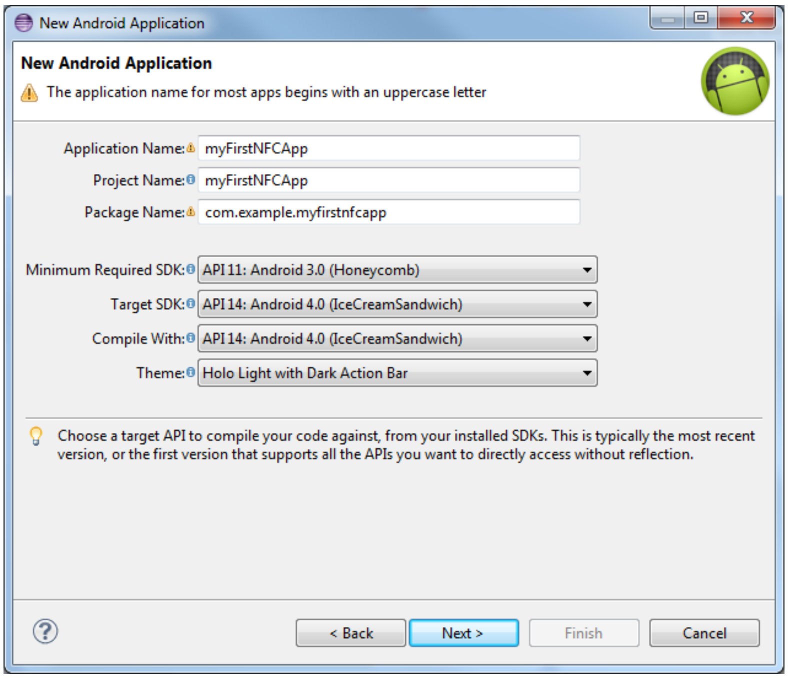
Target SDK: API 18: Android 4.3 (Jelly Bean)

Compile With: API 19: Android 4.4

Theme: Holo Light with Dark Action Bar

Set the minimum Required SDK to API 11, Target SDK to API 14, and compile with to at least API14, then press Next several times until the project is created (*Figure 9*). Ensure that the Android API level14 SDK is installed on the developing station. Once the project is created, build it to ensure that whole project creation settings are valid. If the build is successful at this step, developer can start/debug myfirstnfcapp application on USB connected android mobile phone to ensure that the application is correctly built.

Figure 9. New Android Application Settings


New Android Application

New Android Application

The application name for most apps begins with an uppercase letter

Application Name: myFirstNFCApp

Project Name: myFirstNFCApp

Package Name: com.example.myfirstnfcapp

Minimum Required SDK: API 11: Android 3.0 (Honeycomb)

Target SDK: API 14: Android 4.0 (IceCreamSandwich)

Compile With: API 14: Android 4.0 (IceCreamSandwich)

Theme: Holo Light with Dark Action Bar

Choose a target API to compile your code against, from your installed SDKs. This is typically the most recent version, or the first version that supports all the APIs you want to directly access without reflection.

2.4 Setup Android Manifest file

Once the project creation first step is completed, to use NFC functionality, developer has to configure properly the android application settings in the AndroidManifest.xml.

2.4.1 NFC permission

In order to use NFC capability, application must request the NFC permission. To do so, this request must be specified in the manifest file by the following declaration:

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

2.4.2 NFC feature

Specify that the application uses NFC by adding the following declaration:

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

In this way, the application is known as not functional if the NFC feature is not available on the android device.

2.4.3 Intent filtering settings

As seen in the application overview section, component can be declared to be NFC intent handler capable. To do so, an intent filter must be declared. Actually the application identified by com.example.myfirstnfcapp in the manifest file is composed by a single component which is an activity. This activity known as the main activity already has an intent filter declaration:

```
<intent-filter>
       <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

This intent filter is declared specifically for main activity component to let Android System to know which activity component to start on user request.

As the application get a single activity and as NFC intent can be handled in this main activity we may declares a new intent filter to handle NDEF message when TAG is discovered.

In case activity component has to manage every kind of NDEF message we may add the following intent filter declaration (solution that we keep for our source code example):

```
<intent-filter>
      <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
</intent-filter>
```

Complementary intent-filter

Android system also lets developer to refine the kind of intent he wants to act on. Keeping the NDEF_DISCOVERED message, developer may want to only manage with a MIME type of text/plain:

```
<intent-filter>
      <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
      <category android:name="android.intent.category.DEFAULT"/>
      <data android:mimeType="text/plain" />
</intent-filter>
```

or wants to act on certain URI in the form of http://www.st.com:

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:scheme="http"
              android:host="www.st.com"/>
</intent-filter>
```

In case developer wants the activity component been activated on *ACTION_TECH_DISCOVERED intent*, he must write a XML file to be store in res/xml folder. Name of the file is what the developer wish (ex: nfc_tech_filter.xml). This file lists the technology supported by the component within a tech-list set as follows:

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <tech-list>
        <tech>android.nfc.tech.IsoDep</tech>
        <tech>android.nfc.tech.NfcA</tech>
        <tech>android.nfc.tech.NfcB</tech>
        <tech>android.nfc.tech.NfcF</tech>
    </tech-list>
</resources>
```

Once the XML NFC filter resources is created, the intent-filter on ACTION_TECH_DISCOVERED is:

```
<intent-filter>
    <action android:name="android.nfc.action.TECH_DISCOVERED"/>
</intent-filter>

<meta-data android:name="android.nfc.action.TECH_DISCOVERED"
    android:resource="@xml/nfc_tech_filter" />
```

The component activity will be then started by the Android system once a tag is discovered with IsoDep, NfcA, NfcB or Nfcf capabilities. (Note that if the detected TAG gets a NDEF message in and if a component is registered to treat NDEF intent the current application will never receive the *ACTION_TECH_DISCOVERED intent.*)

In case developer expects his component is instantiate on ACTION_TAG_DISCOVERED intent he must simply declare the following intent filter in the AndroidManifest.xml file:

```
<intent-filter>
    <action android:name="android.nfc.action.TAG_DISCOVERED"/>
</intent-filter>
```

2.5 Android NFC Activities implementation

As seen in previous sections, application is a set of components from different kind of family (ie. activities, services,..). Components may be started from intents sent by the Android System or from the application itself. In this case, the component known as the main one is started by the application upon user's start application request.

Once the activities has been declares with the associated intent filters in the androidManifest.xml file activity has to be implemented by a corresponding Java Class. In the studied case, developer has to implement the lifecycle activity callback, the NFC intent handling functionalities, and finally the NFC message treatment.

Here is the provided implementation of the MainActvity automatically generated during project creation, referenced in the AndroidManifest.xml file that developer has to upgrade to support NFC feature.

```
package com.example.myfirstnfcapp;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class MainActivity extends Activity {

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

      @Override
      public boolean onCreateOptionsMenu(Menu menu) {
            // Inflate the menu; this adds items to the action bar if it is present.
            getMenuInflater().inflate(R.menu.main, menu);
            return true;
      }

}
```

2.5.1 Import NFC packages

In order to support NFC functionality, developer imports NFC android packages first. With Eclipse IDE it can be automatically done while developer uses new objects.

```
import android.nfc.NdefMessage;
import android.nfc.NfcAdapter;
```

2.5.2 NFC Adapter and foreground Dispatch

Involving with NFC sub system requires retrieving the reference of the NFC adapter. In *onCreate()* callback developer grabs a reference to the NFC adapter:

Add a new class member to the activity class. This mAdapter reference allows to listening the tag being scanned:

```
NfcAdapter mAdapter;
```

Retrieve the default NFC adapter on component creation:

```
mAdapter = NfcAdapter.getDefaultAdapter(this);
```

In order to declare the application to grab NFC intents and to answer to all NFC intent while the activity is running (see activity lifecycle stage1) NFC dispatch capabilities has to be called in *onResume()* and disabled in *onpause()* callbacks.

```
@Override
public void onResume() {
        super.onResume();
        PendingIntent intent = PendingIntent.getActivity(this, 0, new Intent(this,
getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);
        NfcAdapter.getDefaultAdapter(this).enableForegroundDispatch(this,intent, null,
null);
}


@Override
protected void onPause() {
        super.onPause();
        if (NfcAdapter.getDefaultAdapter(this) != null)
              NfcAdapter.getDefaultAdapter(this).disableForegroundDispatch(this);
}
```

With the previous *enableForegroundDispatch()*, activity is declared as a grabber of whole NFC intents while the activity is in foreground state. It's possible to add filter to only grab specific NFC intents like ACTION_TAG_DISCOVERED ones. In *onResume()* call back, it may then possible to declare a filter and configure the default NfcAdapter with:

```
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
                              new Intent(this,
                              getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);

IntentFilter tagtoHandle = new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
IntentFilter[] filters = new IntentFilter[] { tagtoHandle };

mAdapter.enableForegroundDispatch(this, pendingIntent, filters, null);
```

2.5.3 Intents and NFC objects treatment

If the MainActivity has been started by the Android System on NFC intent, the activity has to retrieve the intent and treat it.

This can be done on activity creation (*onCreate()* callback call) by the following implementation:

```
Intent intent = getIntent();
                    String action = intent.getAction();
                    // Manage Intent in case the intent activate the application (automated
launch)
                    // check AndroidManifest.xml
                    if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(action)) {

                            process(intent); // call private method to treat the intent

                            // storing NDEF Data

       }
```

If a new NFC tag is detected during the MainActivity life (ie.:Activity is already in a started state) and as activity is in launched mode set to "singleTop" ,the new intent is sent to the activity which is in a foreground state and the onNewIntent(Intent) callback is called by the system. Then onNewIntent(intent) callback can be overridden as following:

```
@Override
public void onNewIntent(Intent intent) {
      String action = intent.getAction();
      if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(action)) {
            process(intent);
            }
      }
```

At this stage the activity is setup to start on NDEF intent from the System, to grab NDEF intent while it still alive. The last action the activity must perform is to treat the received NDEF message according the activity is designed to. Received NDEF intent carrying all the Tag information parsed by low HW/SW layers.

Below is an example on how to handle NDEF message recorded by the Android NFC stack. Developer may upgrade this method to fit his own implementation needs:

```
private void process(Intent intent)
        {

                Parcelable[] parcelabled =
intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);

                if (parcelabled != null)
                {
                // notify user that a NDEF tag is discovered and processed.
                Toast.makeText(getApplicationContext(), "NDEF TAG DETECTED",
Toast.LENGTH_LONG).show();
                //retrieving the whole NDEF message detected from the current tag discovered
from the NFC field
                NdefMessage[] ndefmsg = new NdefMessage[parcelabled.length];
                        for (int i = 0; i < parcelabled.length; i++) {
                                ndefmsg[i] = (NdefMessage) parcelabled[i];
                        }
                // log records from first NDEF message
                for (int i=0; i<ndefmsg[0].getRecords().length;i++)
                {
                        // log the NDEF record TNF value / type  and payload
                        Log.d("NDEF Intent process",ndefmsg[0].getRecords()[i].toString());
                }

                }
                return;
        }
```

2.5.4 Final AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myfirstnfcapp"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="11"
        android:targetSdkVersion="19" />
    <uses-permission android:name="android.permission.NFC" />

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


    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.myfirstnfcapp.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
                   <intent-filter>
                   <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
                   </intent-filter>

        </activity>
    </application>

</manifest>
```

2.5.5 Final MainActivity class file

```
package com.example.myfirstnfcapp;

import java.io.IOException;

import android.app.Activity;
import android.app.PendingIntent;
import android.content.Intent;
import android.nfc.NdefMessage;
import android.nfc.NfcAdapter;
import android.os.Bundle;
import android.os.Parcelable;
import android.util.Log;
import android.view.Menu;
import android.widget.Toast;

public class MainActivity extends Activity {


      NfcAdapter mAdapter;

      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            mAdapter = NfcAdapter.getDefaultAdapter(this);

            Intent intent = getIntent();
        String action = intent.getAction();
        // Manage Intent in case the intent activate the application (automated launch)
        // check AndroidManifest.xml
        if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(action)) {
                  process(intent); // call private method to treat the intent
//storing NDEF Data
}



      }


      @Override
      public void onNewIntent(Intent intent) {
            String action = intent.getAction();
            if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(action)) {
                  process(intent);
                  }
            }
private void process(Intent intent)
        {

                Parcelable[] parcelabled =
intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);

                if (parcelabled != null)
                {
                // notify user that a NDEF tag is discovered and processed.
                Toast.makeText(getApplicationContext(), "NDEF TAG DETECTED",
Toast.LENGTH_LONG).show();
                //retrieving the whole NDEF message detected from the current tag discover
from the NFC field
                NdefMessage[] ndefmsg = new NdefMessage[parcelabled.length];
                        for (int i = 0; i < parcelabled.length; i++) {
                                ndefmsg[i] = (NdefMessage) parcelabled[i];
                        }
                // log records from first NDEF message
                for (int i=0; i<ndefmsg[0].getRecords().length;i++)
                {
                        // log the NDEF record TNF value / type  and payload
                        Log.d("NDEF Intent process",ndefmsg[0].getRecords()[i].toString());
                }

                }
                return;
        }

        @Override
        public boolean onCreateOptionsMenu(Menu menu) {
                // Inflate the menu; this adds items to the action bar if it is present.
                getMenuInflater().inflate(R.menu.main, menu);
                return true;
        }

        @Override
        public void onResume() {
        super.onResume();
        PendingIntent intent = PendingIntent.getActivity(this, 0, new Intent(this,
getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);
        NfcAdapter.getDefaultAdapter(this).enableForegroundDispatch(this,intent, null,
null);
        }

        @Override
        protected void onPause() {
        super.onPause();
        if (NfcAdapter.getDefaultAdapter(this) != null)
        NfcAdapter.getDefaultAdapter(this).disableForegroundDispatch(this);
        }


}
```

3 How to use ST NFC package to implement HW enhanced features

Customer may want to use enhanced feature provided by ST-M24SR or ST-M24LR. As these features are not NDEF dedicated our out of NFC Forum scope or not yet provided by Android API, user may address discovered NFC Tag through Android NFC technology API and with dedicated protocol defined by ST (or customer in case of transceiver addressing use case).

These study examples are extracted from dedicated ST product android application. These applications are downloadable from ST web site as Android application itself or as Android Application source code.

3.1 Enhanced M24SR feature use case – CCFile addressing

NFC android Application developer may want to read the CC file stored in each NFC Type 4 Tag. Android API doesn't provide a high level interface to address this kind of file. Developer has to address specific nfc.technology provided by android to send 7816-4 commands to access this specific file.

Once a tag is discovered from the NFC field, intent is grabbed by the current foreground activity. With this intent, a rawTag which is the software representation of the state of the physical tag detected by the NFC field is extractable by the following call:

```
Tag rawTag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
```

If the Tag type is identified as a NFC Forum Tag Type 4 by the *decodeTagType()* method, physical tag in the NFC field gets a CCFile. Developer has to request it through a sequence of commands to be able to read it. These commands are listed in the ST-M24SR datasheet downloadable from the ST web site.

```
private static NfcTagTypes decodeTagType (Tag pTag) {
		NfcTagTypes lType = NfcTagTypes.NFC_TAG_TYPE_UNKNOWN;
		List<String> lTechList = Arrays.asList(pTag.getTechList());
		String nfcTechPrefixStr = "android.nfc.tech.";
		// Try the Ndef technology
		Ndef lNdefTag = Ndef.get(pTag);
		if (lNdefTag != null) {
			if (lNdefTag.getType().equals(Ndef.NFC_FORUM_TYPE_1)) {
				lType = NfcTagTypes.NFC_TAG_TYPE_1;
			} else if (lNdefTag.getType().equals(Ndef.NFC_FORUM_TYPE_2)) {
				lType = NfcTagTypes.NFC_TAG_TYPE_2;
			} else if (lNdefTag.getType().equals(Ndef.NFC_FORUM_TYPE_3)) {
				lType = NfcTagTypes.NFC_TAG_TYPE_3;
			} else if (lNdefTag.getType().equals(Ndef.NFC_FORUM_TYPE_4)) {
				if (lTechList.contains(nfcTechPrefixStr+"NfcA")) {
					lType = NfcTagTypes.NFC_TAG_TYPE_4A;
				} else if (lTechList.contains(nfcTechPrefixStr+"NfcB")) {
					lType = NfcTagTypes.NFC_TAG_TYPE_4B;
				}
			}
		} else {
			// Try the IsoDep technology
			IsoDep lIsoDepTag = IsoDep.get(pTag);
			if (lIsoDepTag != null) {
				if (lTechList.contains(nfcTechPrefixStr+"NfcA")) {
					lType = NfcTagTypes.NFC_TAG_TYPE_4A;
				} else if (lTechList.contains(nfcTechPrefixStr+"NfcB")) {
					lType = NfcTagTypes.NFC_TAG_TYPE_4B;
				}
			} else {
				// Try the underlying technologies
				if (lTechList.contains(nfcTechPrefixStr+"NfcA")) {
					lType = NfcTagTypes.NFC_TAG_TYPE_A;
				} else if (lTechList.contains(nfcTechPrefixStr+"NfcB")) {
					lType = NfcTagTypes.NFC_TAG_TYPE_B;
				} else if (lTechList.contains(nfcTechPrefixStr+"NfcF")) {
					lType = NfcTagTypes.NFC_TAG_TYPE_F;
				} else if (lTechList.contains(nfcTechPrefixStr+"NfcV")) {
					lType = NfcTagTypes.NFC_TAG_TYPE_V;
				}
			}
		}

		return lType;
	}
```

At this stage physical tag is supposed to be available in the NFC field. The anti-collision, RATS, PPS sequences are already done and the Tag is accessible in read or write mode.

To select the TAG and set it in an accessible mode Android Application sends to the tag an Application To Select command defined in 7816-4 specification. To do so, the Tag rawTag can be mapped to an IsoDep android object in order to send a synchronous ATS command to the physical tag. The defined command is sent to the tag by using the

IsoDep.tranceive(byte[]) command. Application then waits for the corresponding answer (answer codes are listed and defined in both 7816-4 specification and in M24SR datasheet).

```
private static final byte[] NdefSelectAppliFrame = new byte[] { (byte) 0x00, (byte) 0xA4,
                                                 (byte) 0x04, (byte) 0x00, (byte) 0x07,
                                                 (byte) 0xD2, (byte) 0x76, (byte) 0x00,
(byte) 0x00,
                                                 (byte) 0x85, (byte) 0x01, (byte) 0x01};

    // request ATS
    public int requestATS()
    {
        byte[] transcieveAnswer = new byte[] { (byte) 0x01 };
        if (isoDepCurrentTag == null)
            isoDepCurrentTag = IsoDep.get(this.currentTag);
        if (isoDepCurrentTag == null)
        {
            return 0;
        }
        int cpt = 0;

        while (( transcieveAnswer[0] == 1 || transcieveAnswer[0] == (byte)0xAA) &&
cpt <= 1)
        {
            try {
                if (!isoDepCurrentTag.isConnected())
                {
                    isoDepCurrentTag.connect();
                }
                    //isoDepCurrentTag.setTimeout(20);
                transcieveAnswer =
isoDepCurrentTag.transceive(NdefSelectAppliFrame);

                if (transcieveAnswer[0] == (byte) 0x90 && transcieveAnswer[1] ==
(byte) 0x00)
                    {
                        return 1;
                    }
                    else
                    {
                        cpt++;
                        return 0;
                    }
            } catch (TagLostException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                throw new RuntimeException("fail", e);
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                throw new RuntimeException("fail", e);
            }
        }
            return 0;
    } // End of ATS request
```

If the *transceiveAnswer* is equal to 0x9000 then the current physical tag is in application select mode and the tag file structure can be addressed.

In the same way activity can send the specific command to address the CCfile. CCFile is known to get a standardized ID: 0xE103. To get the CCFile, application sends the following message sequence:

- Select the file with the ID 0xE103
- Read the file size of the CCFile
- Read the binary of size previously request which is corresponding to the file.

```
private static final byte[] CCSelect = new byte[] {(byte) 0x00, (byte) 0xA4,

                                (byte) 0x00, (byte) 0x0C, (byte) 0x02,

                                (byte) 0xE1, (byte) 0x03 };

public int requestCCSelect()
    {
        byte[] transcieveAnswer = new byte[] { (byte) 0x01 };
        if (isoDepCurrentTag == null)
            isoDepCurrentTag = IsoDep.get(this.currentTag);
        int cpt = 0;

        while (( transcieveAnswer[0] == 1 || transcieveAnswer[0] == (byte)0xAA) &&
cpt <= 1)
        {
            try {
                if (!isoDepCurrentTag.isConnected())
                {
                    isoDepCurrentTag.connect();
                    isoDepCurrentTag.setTimeout(20);
                }
                transcieveAnswer = isoDepCurrentTag.transceive(CCSelect);

                if (transcieveAnswer[0] == (byte) 0x90 && transcieveAnswer[1] ==
(byte) 0x00)
                    {
                        return 1;
                    }
                    else
                    {
                        cpt++;
                        return 0;
                    }
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                throw new RuntimeException("fail", e);
            }
        }
            return 0;
    } // End of requestCCSelect
```

```
private static final byte[] CCReadLength = new byte [] {(byte) 0x00, (byte) 0xB0,
                                                          (byte) 0x00, (byte) 0x00, (byte)
0x02};


public int requestCCReadLength()
      {
            byte[] transcieveAnswer = new byte[] { (byte) 0x01 };
            if (isoDepCurrentTag == null)
                  isoDepCurrentTag = IsoDep.get(this.currentTag);
            int cpt = 0;

            int CCLength = 0;

            while (( transcieveAnswer[0] == 1 || transcieveAnswer[0] == (byte)0xAA) &&
cpt <= 1)
            {
                  try {
                        if (!isoDepCurrentTag.isConnected())
                        {
                              isoDepCurrentTag.connect();
                              isoDepCurrentTag.setTimeout(20);
                        }
                        transcieveAnswer = isoDepCurrentTag.transceive(CCReadLength);

                        if (transcieveAnswer[2] == (byte) 0x90 && transcieveAnswer[3] ==
(byte) 0x00)
                              {

                                    CCLength = (int)((transcieveAnswer[0] & 0xFF)<<8) +
(int)(transcieveAnswer[1]&0xFF);
                                    return CCLength;
                              }
                              else
                              {
                                    cpt++;
                                    return 0;
                              }
                  } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                        throw new RuntimeException("fail", e);
                  }
            }
                  return 0;
      } // End of CCReadLength request
```

```
private static final byte[] readBinary = new byte [] {(byte) 0x00, (byte) 0xB0,
                                          (byte) 0x00, (byte) 0x00, (byte) 0x00};

private int requestReadBinary(int size, byte [] buffer)
      {

            byte[] transcieveAnswer = new byte[] { (byte) 0x00 };

            if (isoDepCurrentTag == null)
                  isoDepCurrentTag = IsoDep.get(this.currentTag);
            int cpt = 0;

            byte[] readcmd = new byte[readBinary.length];

            System.arraycopy(readBinary, 0, readcmd, 0, readBinary.length);
            readcmd[4] = (byte) (size & 0xFF);

                  try {
                        if (!isoDepCurrentTag.isConnected())
                        {
                              isoDepCurrentTag.connect();
                              isoDepCurrentTag.setTimeout(20);
                        }

                        transcieveAnswer = isoDepCurrentTag.transceive(readcmd);

                        if (transcieveAnswer[size] == (byte) 0x90 &&
transcieveAnswer[size+1] == (byte) 0x00)
                              {
                                    System.arraycopy(transcieveAnswer, 0, buffer, 0,
size);
                                    return 1;
                              }
                              else
                              {
                                    cpt++;
                                    return 0;
                              }
                  } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                        throw new RuntimeException("fail", e);
                  }
      }
```

At this stage, the whole CC file is stored in the buffer set in requestReadBinary’s parameter. Current activity may then parse the buffer to fill a CCFile structured object:

```
_CCHandler = new stnfccchandler(CCBinary);
```

With St object *stnfcchandler* constructor:

```
public stnfcchandler(byte[] buffer)
      {
            mcclength = (short) ((buffer[0]<<8 &0xFF) + (buffer[1]&0xFF));
            mccmappingver = (short)(buffer[2]&0xFF);
            mmaxbytesread = (short) ((((short)buffer[3])<<8 &0xFF) +
(((short)buffer[4])&0xFF));
            mmaxbyteswritten = (short) ((((short)buffer[5])<<8 &0xFF) +
(((short)buffer[6])&0xFF));

            mnbTLVBlocks = 1;

            mtfield = (short)buffer[7];
            mlfield = (short)buffer[8];
            mfieldId = (int)((buffer[9]&0xFF)<<8)+ (int)(buffer[10]&0xFF);
            mndeffilelength = (int)((buffer[11]&0xFF)<<8) + (int)(buffer[12]&0xFF);
            mreadeaccess = (short)buffer[13];
            mwriteaccess = (short)buffer[14];
      }
```

3.2 Enhanced M24LR features Study – Read / Write 15693 Tag

ST M24LR products are ISO/IEC-15693 based product not natively supported by Android System. On some Android Phone, if presented tag is NDEF formatted the Android system may detect the tag and parse file stored inside. If tag is not formatted as a NDEF message, developer can address the memory from the tag in raw addressing mode by sending to the tag specific protocol commands defined by 15693-3 standard. For such an addressing mode, developer implements the relevant android technology tag to handle ST-M24LR products.

Let's show how to implement an activity able to read and write on this kind of tag.

As previous use case, developer retrieves the nfcAdapter and registers the activity to receive NFCV tags events. In the NFC-V use case, intentFilter is declared in order the activity received only the 15693 tag event by using the android NfcV technology (see mfilters and mTechLists).

```
Intent = PendingIntent.getActivity(this, 0,new Intent(this,
()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP), 0);
lter ndef = new IntentFilter(NfcAdapter.ACTION_TECH_DISCOVERED);
            mFilters = new IntentFilter[] {ndef,};
             mTechLists = new String[][] { new String[] {

     android.nfc.tech.NfcV.class.getName() } };


.enableForegroundDispatch(this, mPendingIntent, mFilters, mTechLists);
```

Once the activity is registered to handled *NfcV* tag intent, *OnNewIntent()* callback system is overridden to retrieve the tag's information. Once the referenced tag is parsed from the intent (tagFromIntent Tag object) developer can send protocol command defined to physical tag by the using the android API NfcV (we supposed that the tag is still in NFC field and that the anti-collison process is already done by the embedded NFC stack).

```
@Override
  protected void onNewIntent(Intent intent)
  {
    // TODO Auto-generated method stub
    super.onNewIntent(intent);
    String action = intent.getAction();
    if (NfcAdapter.ACTION_TECH_DISCOVERED.equals(action))
    {
          Tag tagFromIntent = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);

          DataDevice dataDevice = (DataDevice)getApplication();
          dataDevice.setCurrentTag(tagFromIntent);

          byte[] GetSystemInfoAnswer =
CCommand.SendGetSystemInfoCommandCustom(tagFromIntent,(DataDevice)getApplication());

          if(DecodeGetSystemInfoResponse(GetSystemInfoAnswer))
          {
                Intent intentScan = new Intent(this, Scan.class);
                startActivity(intentScan);
          }
          else
          {
                return;
          }
    }
  }
```

```
*****************************************************************/
function send an Get System Info command (0x02 0x2B)
argument myTag is the intent triggered with the TAG_DISCOVERED
*****************************************************************/

tatic byte[] SendGetSystemInfoCommandCustom (Tag myTag, DataDevice ma)

oolean boolDeviceDetected = false;

/ --- 1st Step : Inventory to detect 1 or 2 bytes address ---
yte[] UIDFrame = new byte[] { (byte) 0x26, (byte) 0x01, (byte) 0x00 };
yte[] response = new byte[] { (byte) 0xAA };

..
lfcV nfcvTag = NfcV.get(myTag);
.
 Send Inventory request to the tag
sponse = nfcvTag.transceive(UIDFrame);

Response gets the answer from the physical tag and the UID if successful

 INVENTORY HAS DETECTED DEVICE ... READING GET SYSTEM INFO
te[] GetSystemInfoFrame1bytesAddress = new byte[2];
tSystemInfoFrame1bytesAddress = new byte[] { (byte) 0x02, (byte) 0x2B };


sponse = nfcvTag.transceive(GetSystemInfoFrame1bytesAddress);

parse response to identify Tag from the NFC field
```

Once the tag is identified as 15693 ST product is possible to read block or write block:

Read Example

```
//***********************************************************************/
//* the function send an ReadSingle command (0x0A 0x20) || (0x02 0x20)
//* example : StartAddress {0x00, 0x02}  NbOfBlockToRead : {0x04}
//* the function will return 04 blocks read from address 0002
//* According to the ISO-15693 maximum block read is 32 for the same sector
//***********************************************************************/

	public static byte[] SendReadSingleBlockCommand (Tag myTag, byte[] StartAddress, 
DataDevice ma)
	{
		byte[] response = new byte[] {(byte) 0x0A};
		byte[] ReadSingleBlockFrame;


		ReadSingleBlockFrame = new byte[]{(byte) 0x02, (byte) 0x20, 
StartAddress[1]};

		int errorOccured = 1;
		while(errorOccured != 0)
		{
			try
			{
				NfcV nfcvTag = NfcV.get(myTag);
				nfcvTag.close();
				nfcvTag.connect();
				response = nfcvTag.transceive(ReadSingleBlockFrame);
				if(response[0] == (byte) 0x00 || response[0] == (byte) 0x01)
					errorOccured = 0;
			}
			catch(Exception e)
			{
				…
			}
		}
		return response;
	}
```

Write Example

```
//*********************************************************************/
//* the function send an WriteSingle command (0x0A 0x21) || (0x02 0x21)
//* the argument myTag is the intent triggered with the TAG_DISCOVERED
//* example : StartAddress {0x00, 0x02}  DataToWrite : {0x04 0x14 0xFF 0xB2}
//* the function will write {0x04 0x14 0xFF 0xB2} at the address 0002
//*********************************************************************/
public static byte[] SendWriteSingleBlockCommand (Tag myTag, byte[] StartAddress, byte[]
DataToWrite, DataDevice ma)
{
        byte[] response = new byte[] {(byte) 0xFF};
        byte[] WriteSingleBlockFrame;

        WriteSingleBlockFrame = new byte[]{(byte) 0x02, (byte) 0x21, StartAddress[1],
DataToWrite[0], DataToWrite[1], DataToWrite[2], DataToWrite[3]};

        int errorOccured = 1;
        while(errorOccured != 0)
       {
              try
              {
              NfcV nfcvTag = NfcV.get(myTag);
              nfcvTag.close();
              nfcvTag.connect();
              response = nfcvTag.transceive(WriteSingleBlockFrame);
              if(response[0] == (byte) 0x00 || response[0] == (byte) 0x01)
                     errorOccured = 0;
              }
              catch(Exception e)
                     {… }

              return response;
       }
}
```

4 Revision history

Table 1. Document revision history

Date	Revision	Changes
14-Feb-2014	1	Initial release.