Mobile Computing

Android NFC

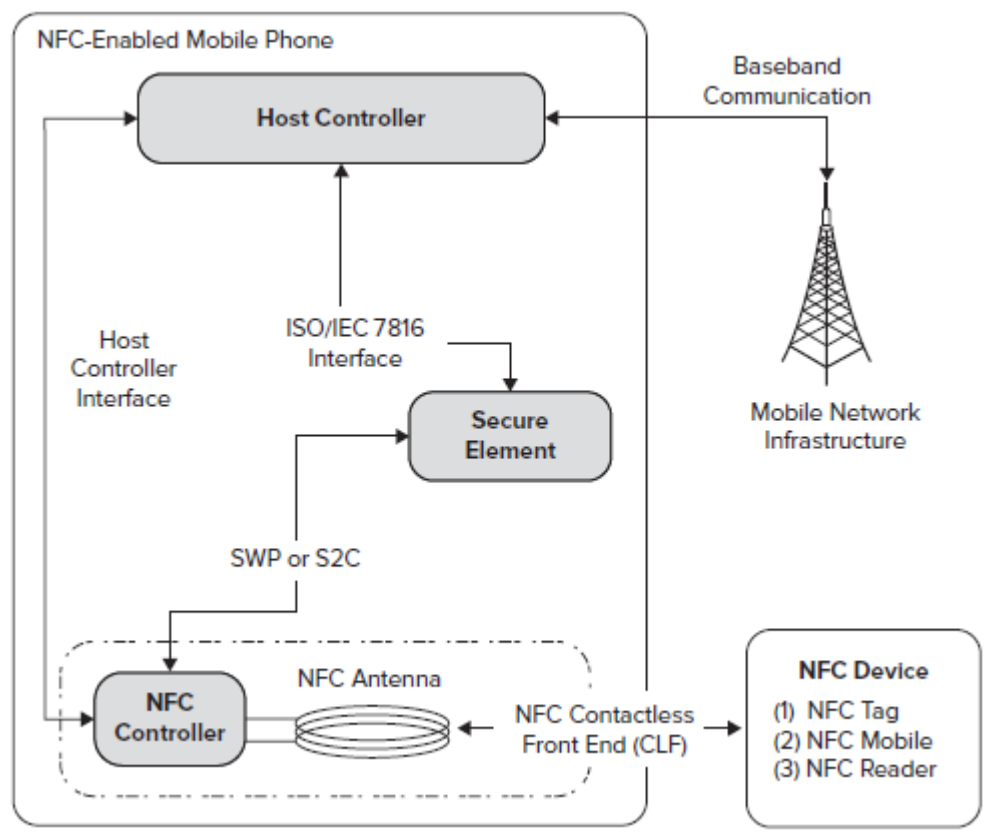
What is NFC

- Global standard for short-range wireless communications
  - NFC – Near Field Communication
  - Driven by NFC Forum
    - https://www.nfc-forum.org
  - Enables simple and safe 2-way interactions in close proximity
    - Allows data exchange
    - Real range usually < 1 cm
    - NFC tags can contain data and be passive (no power)
    - Read and write operations on certain tags
    - Device interaction
    - Card emulation with secure element (or soft emulation)

NFC Modes and Android

- NFC can work in three modes
  - Read and write tags
  - Peer-to-Peer
  - Card emulation
- Android support
  - Read and write tags (good) (since API 10)
  - P2P (beam/limited) (since API 14)
  - Card Emulation (real/emulated SE) (since API 19)
- Core Android Classes
  - NFCManager (system service)
  - NFCAdapter
  - Tag Technology classes
  - HostApduService

NFC Hardware Architecture



- The NFC controller can be operated from the mobile host and work with any of the three operating modes
- The hardware can include a secure element which is similar to a smartcard running JavaCard OS (it may be the phone SIM)
- The secure element can contain applications (applets) accessed from the mobile host
- The secure element has also a direct connection to the NFC controller

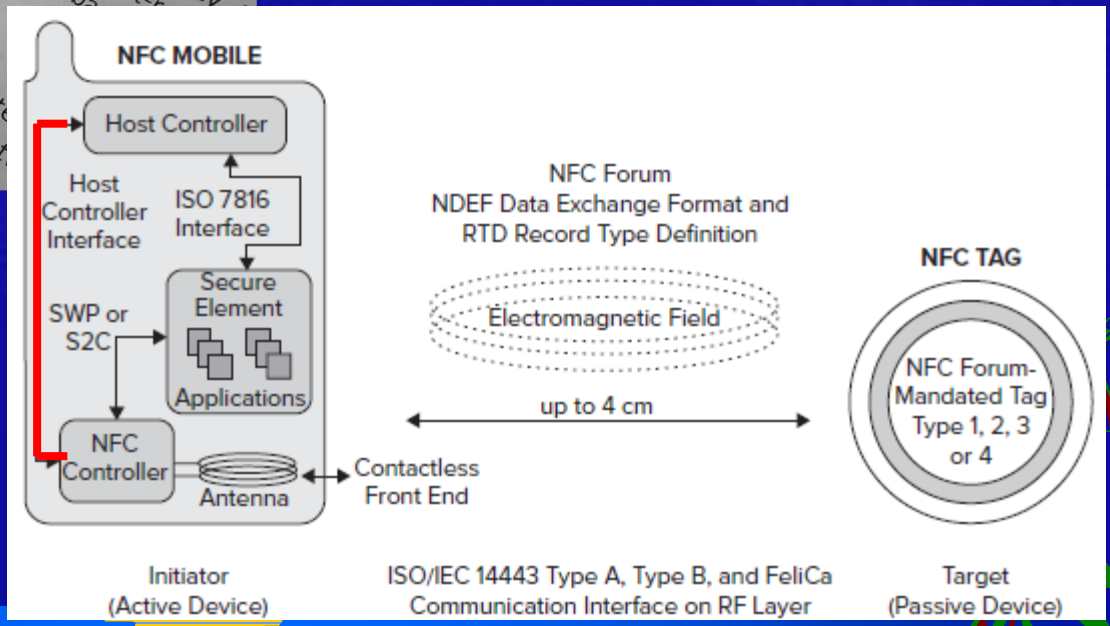
In Android there is only support for the connection between the host and NFC controller, essential for the read/write and P2P modes of operation, and also to emulated SE's.

With an extension (SEEK for Open SimAlliance Mobile API) implemented by some mobile makers, it is possible to use the connection between the host and the secure element.

Read/Write Mode

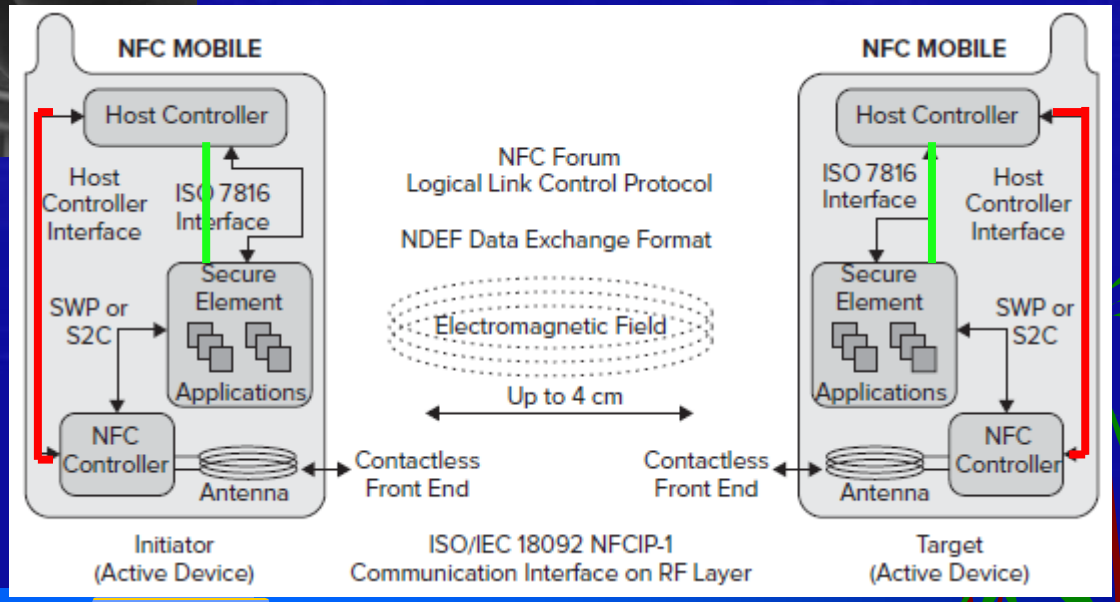
Reading information from tags

Writing information to tags accepting it (write once / read/write tags)



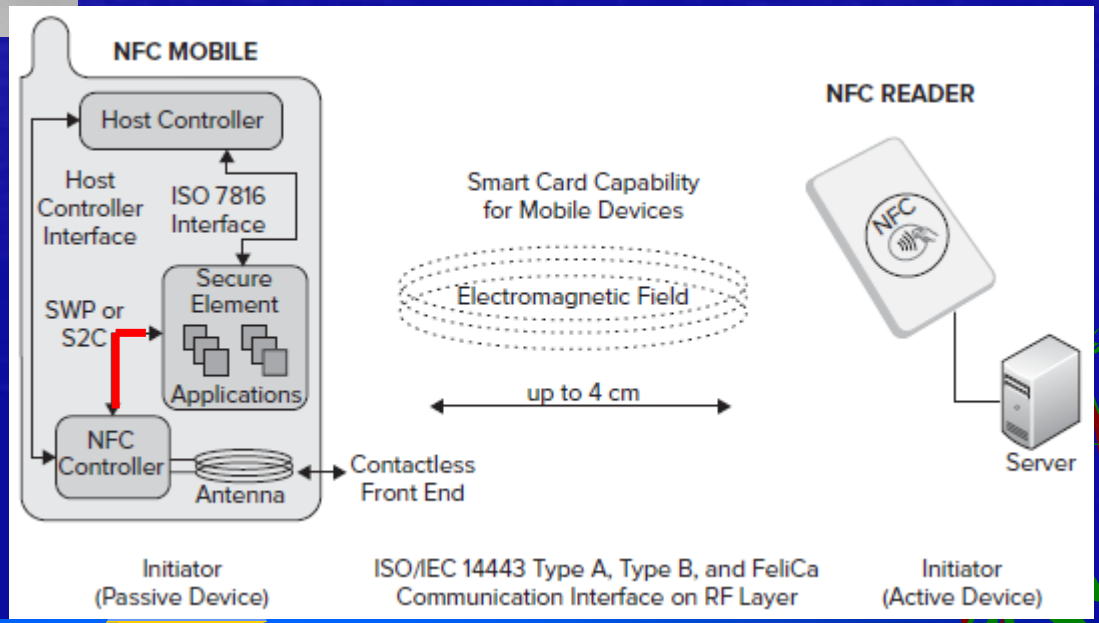
Peer-to-Peer Mode

Communication between NFC devices



Card Emulation Mode

The remote reader talks directly to the secure element applet using NFC as a channel

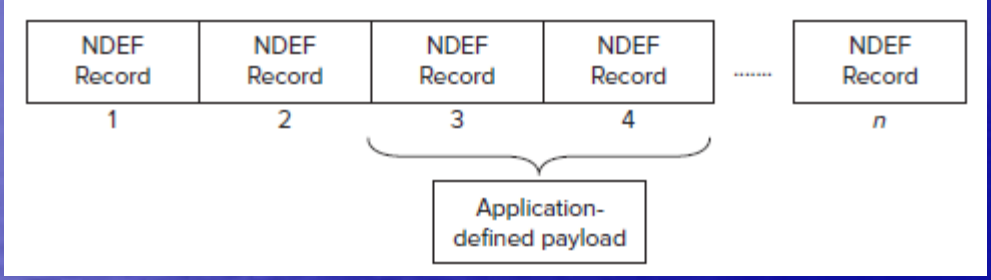


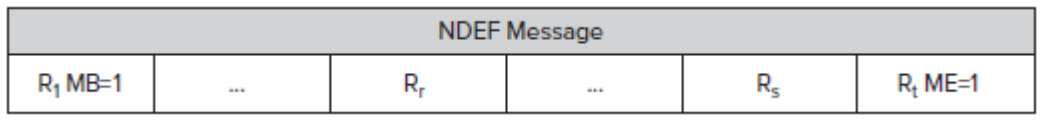
NFC Messages

Messages between a device and a tag or between devices follow a standard format:

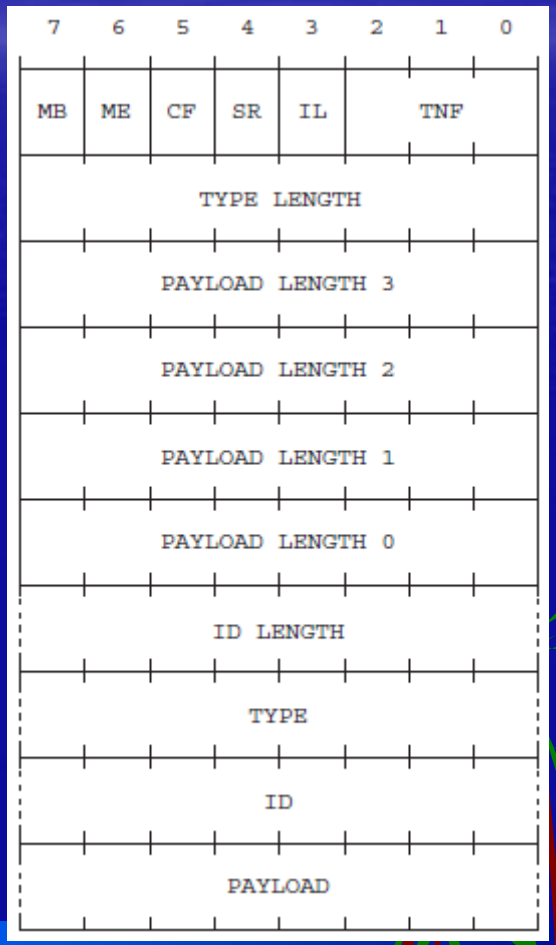
NDEF – NFC Data Exchange Format

NDEF Message





NDEF Record



short tnf, byte[] type, byte[] id, byte[] payload

Android NdefMessage

Android supports NFC Messages through the NdefMessage class.
It has a constructor accepting an array of NdefRecord objects, each one representing a NDEF record.

When obtaining an NDEF message represented by the NdefMessage object we can get the records contained in it through the getRecords() method.

The NdefRecord constructor accepts a TNF field (pre-defined constant), type, id, and payload data (NdefRecord(tnf: Short, type: ByteArray, id: ByteArray, payload: ByteArray)).
The type, id and payload are conditioned by the TNF field. For tags, TNF values of TNF_ABSOLUTE_URI or TNF_WELL_KNOWN are usually used, with corresponding values of type, id, and payload. For device to device messaging the TNF value is usually TNF_MIME_MEDIA with a custom media type in type, an id of 0 bytes, and a payload.
The NdefRecord class has get methods for the TNF, type, id, and payload constituents.
It has also static NdefRecord methods for building some kinds of records, like createMime().

Example building a mime message

```
val mimeType = "application/my.mime.type“       // a custom mime type
val payload = "This is a TNF_MIME_MEDIA"
val mimeRecord = NdefRecord.createMime(mimeType,
                                       payload.getBytes(Charsets.UTF_8))
val newMessage = NdefMessage(arrayOf(mimeRecord))
```

Requirements to Use NFC in Android

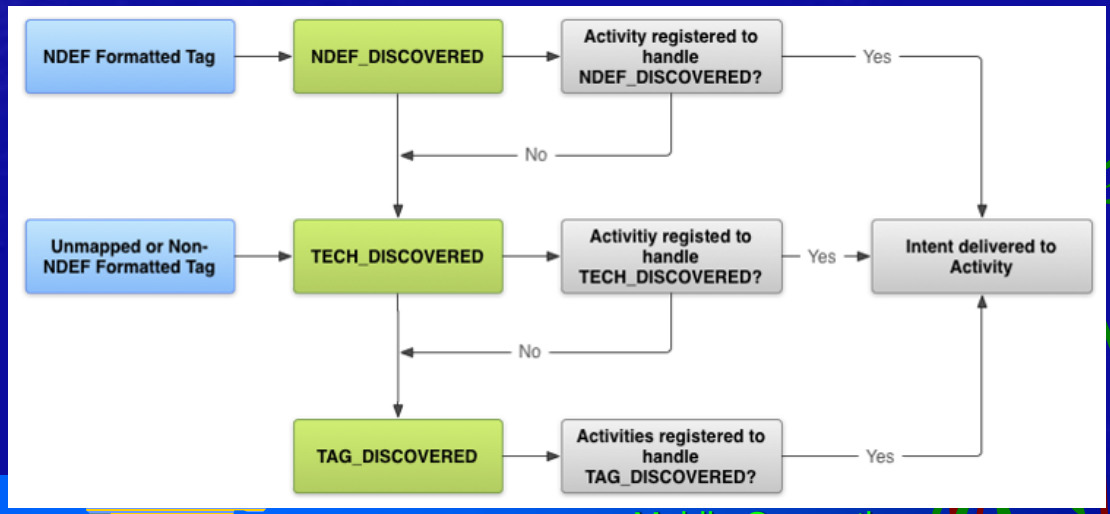
The hardware must be present and enabled in the settings.

The application manifest should contain the necessary permission and uses feature declaration. Also, the minimum SDK version should be at least 10.

```
<uses-permission android:name="android.permission.NFC" />
<uses-sdk android:minSdkVersion="10"/>
<uses-feature android:name="android.hardware.nfc" android:required="true" />
```

When an NFC message is received from a tag or other device, an intent is generated depending on the message format and activities registered for handling the intent action.

Intent generation strategy when an NFC message is received



These intents carry other information on extra fields In the extra bundle:

. EXTRA_TAG – mandatory (is a Tag object)
. EXTRA_NDEF_MESSAGES – An array of NdefMessage for NDEF_DISCOVERED intents
. EXTRA_ID (opcional) – The low-level ID of the tag

Declaring Intent Filters for NFC

As seen, receiving NFC messages triggers the emission of an intent with one of the actions:

- action.NDEF_DISCOVERED
- action.TECH_DISCOVERED
- action.TAG_DISCOVERED

For an activity to respond to these actions it must have a corresponding intent filter in the manifest. Here are some examples:

Intent Filter for TNF_WELL_KNOWN with RTD_URI

(To explicitly respond to a tag with a URL containing: http://nfclab.com)

```
<activity>
   ...
   <intent-filter>
      <action android:name="android.nfc.action.NDEF_DISCOVERED" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:scheme="http“  android:host="nfclab.com“  android:pathPrefix="" />
   </intent-filter>
```

Intent Filter for TNF_MIME_MEDIA

(Responds to a NDEF message containing the mime type application/my.mime.type)

```
<activity>
   ...
   <intent-filter>
      <action android:name="android.nfc.action.NDEF_DISCOVERED" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="application/my.mime.type" />
   </intent-filter>
```

NFC Tag Writing

In order to write to a writable tag, it must be discovered first (sending a message to the device). When the activity processes the received message, it can write a new message to the tag, getting the corresponding object from the intent, like:

```
if (NfcAdapter.ACTION_NDEF_DISCOVERED == intent.getAction()) {
    val detectedTag : Tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG)
       // PREPARE THE NDEF MESSAGE TO WRITE  ....
    writeNdefMessageToTag(newMessage, detectedTag)
}
```

```
fun writeNdefMessageToTag(message: NdefMessage, detectedTag: Tag): Boolean  {
   val size = message.toByteArray().length
   try {
      val ndef = Ndef.get(detectedTag)
      if (ndef != null) {
         ndef.connect()
         // verify writable condition and size capacity …
         ndef.writeNdefMessage(message)
         ndef.close()
         return true
      }
      else {
         val ndefF = NdefFormatable.get(detectedTag)
         if (ndefF != null) {
```

```
            try {
               ndefF.connect()
               ndefF.format(message)
               ndefF.close()
               return true
            } catch (…
         else {
            return false
```

```
            } catch ( …
               return false
            }
```

Receiving NFC Messages

When a new NFC message is received by a device an intent is generated.
This intent will try to start an activity instance that matches it in his intent filter, even if that activity is already running.
To avoid the creation of new activity instances we can specify the launchMode property of the activity (in the manifest) to "singleTop".
In this case, when a new intent is delivered to this activity, instead of creating a new instance, the onNewIntent() callback is called. Because the arrival of new intents to an an activity always pauses it, you can count that the onResume() callback is also called.

Usually, the code to receive an NFC message uses the two previous callbacks. The intent received in a onNewIntent() callback must be set in the activity, otherwise we only can get the original intent (the one that has started the activity).

```
public override onNewIntent(intent: Intent) {
    setIntent(intent)
}

public override onResume() {
    super.onResume();
    if (NfcAdapter.ACTION_NDEF_DISCOVERED == intent.getAction()) {
        processIntent(intent);
    }
}
```

or whatever action we expect

```
val rawMsgs: Array<Parcelable> = intent.getParcelableArrayExtra(
            NfcAdapter.EXTRA_NDEF_MESSAGES)
// generally only one message sent ...
val msg = rawMsgs[0] as NdefMessage
```

to get messages from the intent ...

Sending Messages in P2P Mode

To send NFC messages using the P2P mode we need access to a NfcAdapter object and possibly implement two interfaces defining callbacks.

The interfaces are defined in the NfcAdapter class and are the:

- OnNdefPushCompleteCallback - defines onNdefPushComplete()
- CreateNdefMessageCallback - defines createNdefMessage()

The first, when registered, is called (in another thread) when an NFC message is delivered. The second is called when the activity is about to send the message. In this way the message can be built, from the activity internal state, only when the user 'touches' another phone.

Another way to define the message to be sent, is calling (usually in the onCreate() callback) the method (from an NfcAdapter object) setNdefPushMessage().
The createNdefMessage() callback, if registered, has priority.

In order to get the NfcAdapter for the device we can go through the NfcManager:

```
val manager = getSystemService(Context.NFC_SERVICE) as NfcManager
val adapter = manager.getDefaultAdapter()
```

or directly:

```
val adapter = NfcAdapter.getDefaultAdapter(this);
```

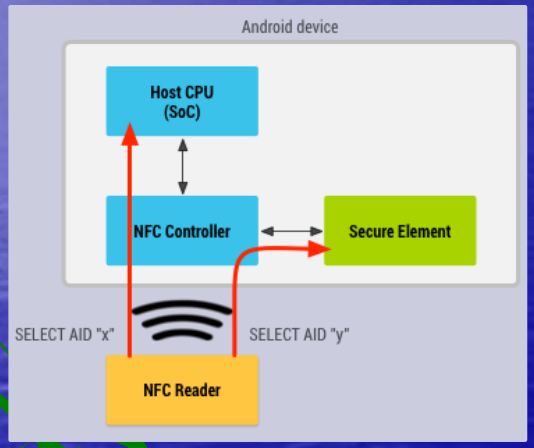
Setting an Activity for Sending

```
override onCreate(savedInstanceState: Bundle?) {
    ...
    val adapter = NfcAdapter.getDefaultAdapter(this)
    if (adapter == null) {
        Toast.makeText(this, "NFC is not available", Toast.LENGTH_LONG).show()
        finish()
        return
    }
    adapter.setOnNdefPushCompleteCallback(this, this);  // callback on sending completion
    adapter.setNdefPushMessageCallback(this, this);     // callback for creating a message
}                                                       // before sending

override createNdefMessage(event: NfcEvent): NdefMessage {
    val text = "Beam me up, Android!\n\nBeam Time: " + System.currentTimeMillis()
    val msg = NdefMessage(NdefRecord( arrayOf(NdefRecord.createMime(
                                        "application/my.mime.type", text.getBytes())))
    return msg
}

override void onNdefPushComplete(event: NfcEvent) {
    // called when the message is delivered, in a different thread than the UI thread
}
```

Example Registering NFC callbacks for sending a message in P2P mode

Card Emulation in a Host Service

- A card reader talks directly to an Android Service emulating a SE in the host processor (not the real SE)
  - The communication is based on an AID (applet ID) and APDU packets implementing applet calls and results
  - The service derives from HostApduService




```
class MyHostApduService : HostApduService() {
   override processCommandApdu(apdu: ByteArray, extras: Bundle): ByteArray {
      ...
   }

   override onDeactivated(reason: Int) {
      ...
   }
}
```

Manifest:

```
<service android:name=".MyHostApduService" android:exported="true"
      android:permission="android.permission.BIND_NFC_SERVICE">
   <intent-filter>
      <action android:name="android.nfc.cardemulation.action.HOST_APDU_SERVICE"/>
   </intent-filter>
   <meta-data android:name="android.nfc.cardemulation.host_apdu_service"
         android:resource="@xml/apduservice"/>
</service>
```

The XML file apduservice.xml (in the xml resources sub-directory) must contain the emulated applet AID, matching the one that will be used by the external reader.
Readers identify the applet to call emiting a special 'Select Aid x' op.