FORUM NOKIA

Nokia 6131 NFC SDK: Programmer's Guide

Version 1.1; July 3, 2007

NFC

NOKIA

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

Change history

March 27, 2007	Version 1.0	Initial document release
July 3, 2007	Version 1.1	Updated for Nokia 6131 NFC SDK 1.1

1 About this document

This document describes how to develop MIDlets with Nokia 6131 NFC SDK 1.1 that use the Contactless Communication API (JSR-257).

1.1 Target audience

The target audience of this document is mobile Java™ developers, whose target system has Near Field Communication (NFC) cards and the communication with these cards is handled trough the Contactless Communication API (JSR-257).

1.2 Typographical conventions

The following typographical conventions are used in this document:

Convention	Explanation
Bold	**Bold** is used to indicate windows, views, pages and their elements, menu items, and button names.
Italic	*Italics* are used when referring to manuals. Italics are also used for key terms and emphasis.
`Courier`	`Courier` is used to indicate parameters, file names, processes, commands, directories, and source code text.
`<variable data>`	Variable data is written between the < and > characters. For example: `http://<host>:<port>/directory`

2 Introduction

This is the Programmer's Guide for the Contactless Communication API (JSR-257), which is part of the Software Development Kit for Nokia 6131 NFC (Nokia 6131 NFC SDK). The Contactless Communication API aims to provide access to the information stored on various contactless targets. The API functionalities are divided into five packages. The API contains mandatory and optional packages, which all are included in the API implementation. However, some functionalities are left unimplemented. This document also defines some Nokia implementation-specific additional interfaces and classes that do not belong to the standard JSR-257.

Near Field Communication (NFC) is a short-range radio frequency technology that evolved from a combination of contactless radio frequency identification (RFID) and interconnection technologies. Operating over a distance of only a few centimeters, it allows users to read (and write) small amounts of data from tags and to communicate with other devices by a simple touch. When touching a tag, the NFC device reads the data stored on the tag and initiates the appropriate action after the user's confirmation. For example, it can open a Web page, call a favorite number, or send an SMS message. Small items such as Web links can also be shared by touching another NFC device.

Applications based on the Contactless Communication API can be developed and tested in a simulated environment. The Nokia 6131 NFC SDK offers a full-blown development environment with NFC tag and NFC smart card simulation, and communication to external card readers and devices supporting JSR-257 as shown in Figure 1.

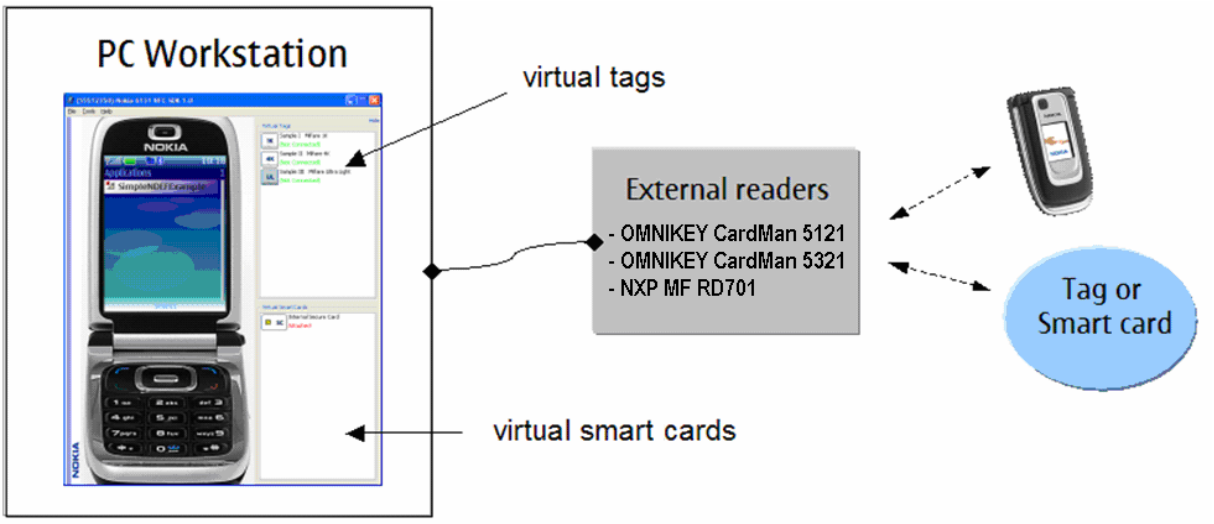

Figure 1: Nokia 6131 NFC SDK development environment

Readers of this guide should have previous experience in developing MIDlets and be familiar with the external card reader products. They should also have a basic knowledge of secure card technologies. Before you start, refer to the Nokia 6131 NFC SDK: User's Guide [1] for more information on how to get started with the Nokia 6131 NFC SDK. For information about the Nokia 6131 NFC device (for example, NFC antenna area), see the User's Guide of the device [2] or refer to the Nokia 6131 NFC device details [12].

3 Using the API

Applications can determine the version of the Contactless Communication API present in the device by checking the system property `microedition.contactless.version`. If the API is present, the returned value is string "`1.0`"; otherwise the returned value is `null`.

Basically the use of this API consists of two phases: discovering contactless targets and data exchange with those targets. There are different types of contactless targets. In the API, each contactless target type is located in its own API package. This chapter describes how to discover contactless targets and exchange data.

3.1 Discovering a contactless target

The starting point of using the API is to discover a contactless target, so that an application can communicate with it. The API package `javax.microedition.contactless` contains the classes and interfaces needed in the contactless target discovery. Table 1 shortly describes classes and interfaces in the package.

Class / Interface	Description
`TagConnection`	Interface for all RFID tag and smart card-related connections.
`TargetListener`	Interface that provides a mechanism for discovering contactless targets.
`TransactionListener`	Interface for security element activity notifications in card emulation mode.
`DiscoveryManager`	Class that offers a mechanism for contactless target discovery.
`TargetProperties`	Class that collects properties that are common to the supported contactless targets.
`TargetType`	Class that collects the contactless target types supported by the API.
`ContactlessException`	Exception that is thrown when an unsupported operation is attempted.

Table 1: Classes and interfaces in the javax.microedition.contactless package

In your application, you can discover only target types that are supported by the device. That is why you should first ensure that the target type you are aiming to discover is supported. You can retrieve information about supported target types by calling the `DiscoveryManager.getSupportedTargetTypes()` method, as shown in the following example:

```
TargetType[] targets = DiscoveryManager.getSupportedTargetTypes();
```

The return value contains the list of target types supported by this API implementation. The possible values are:

- `TargetType.ISO14443_CARD` for ISO 14443-4 compliant smart cards accessed using APDU commands.
- `TargetType.NDEF_TAG` for a tag that contains NFC Forum formatted data.
- `TargetType.RFID_TAG` for general RFID tags.

Now, when you know the supported target types, you can set your application to listen for those target types. For example, the following code snippet shows how you can discover tags supporting NFC Forum formatted data.

```
DiscoveryManager dm = DiscoveryManager.getInstance();
try {
  dm.addTargetListener(this, TargetType.NDEF_TAG);
}
catch (ContactlessException ce) {
  // handle exception
}
```

Note that only one `TargetListener` for each `TargetType` is allowed. Setting a listener for a target type that already has a registered `TargetListener` will cause an `IllegalStateException` to be thrown. The code snippet below demonstrates `TargetListener` implementation by showing how to open the connection to the first found target by using `NDEFTagConnection`. Note that the `getUrl()` method on `TargetProperties` returns `null` on connections other than NDEF ones.

If the detected target contains NDEF formatted data, the first element in `TargetProperties` contains the NDEF target.

```
public void targetDetected(TargetProperties[] prop) {
  // Select the first found target
  TargetProperties target = prop[0];

  // Get URL to open the NDEF connection
  String url = target.getUrl();
  try {
    // Open NDEFTagConnection to the target
    NDEFTagConnection conn = (NDEFTagConnection)Connector.open(url);
    // use the opened NDEFTagConnection.
  }
  catch (IOException ioe) {
    // Handle exception
  }
}
```

Note: There must not be more than one `TargetListener` for each `TargetType`. The restriction for one `TargetListener` is needed because the hardware handles only one connection to the physical target at a time. The registration of `TargetListener` should fail if there is an existing listener in other Java virtual machines or in the native platform. Therefore, it would be a good idea to unregister the `TargetListener` in your MIDlet's `destroyApp(boolean unconditional)` method. Otherwise, other applications might not be capable of registering target listeners.

Connections to different contactless targets are designed on top of the Generic Connection Framework (GCF). Each different target type defines a new protocol to the GCF. In practice, a connection opening to a discovered contactless target can be made using the `open` method of `javax.microedition.io.Connector`.

Note: Transaction events through `TransactionListener` are not supported in the Nokia 6131 NFC SDK. Adding a listener with the `addTransactionListener` method of `DiscoveryManager` will cause a `ContactlessException`.

3.2 Exchanging NDEF formatted data

The `NDEFTagConnection` interface defines the basic functionality for exchanging NFC Forum formatted data with contactless targets. The actual data is stored in the `NDEFMessage` object containing the data in `NDEFRecords`. Those classes and the `NDEFTagConnection` interface are located in the `javax.microedition.contactless.ndef` package (see Table 2).

Class / Interface	Description
`NDEFRecordListener`	Interface that provides a mechanism for NDEF records discovery from contactless targets.
`NDEFTagConnection`	Interface for exchanging NFC Forum formatted data.
`NDEFMessage`	Class representing an NDEF message.
`NFEFRecord`	Class representing an NDEF record.
`NDERecordType`	Class that encapsulates the type of an NDEF record.

Table 2: Classes and interfaces in the javax.microedition.contactless.ndef package

The `NDEFTagConnection` interface provides methods for reading and writing data. The `readNDEF()` method reads NFC Forum formatted data from the target and the `writeNDEF(NDEFMessage message)` method writes NFC Forum formatted data in the NDEF message to the target.

The `NDEFRecordListener` provides a mechanism for the application to be notified when NDEF records are discovered from the contactless targets. The notification is requested based on the NDEF record type on the target. Through this notification the application has a read-only access to the data on the target, but it has no possibility to open a connection to it. Applications based on JSR-257 must implement this interface to receive the notification. The `NDEFRecordListener` interface contains one method that is called when registration rules are filled:

```
void recordDetected(NDEFMessage ndefMessage)
```

You can add `NDEFRecordListener` to a certain `NDEFRecortType` by using DiscoveryManager's method `addNDEFRecordListener(NDEFRecordListener listener, NDEFRecordType recordType)`. A registered listener can be removed by using the method `removeNDEFRecordListener(NDEFRecordListener listener, NDEFRecordType recordType)`. There can be multiple applications registered to receive notifications about different NDEF records. Only one application can register a listener for a certain NDEF record type.

3.3 Using JSR-257 API extensions

The JSR-257 Contactless Communication API contains specific API extensions that provide interfaces for communicating and accessing certain types of targets.

The Nokia 6131 NFC SDK supports all these extensions, but some of them are provided only as stub implementations. With the stub implementation, developers are able to create and compile MIDlets that use these API extensions using the Nokia 6131 NFC SDK, but they need to test and run those on a real Nokia 6131 NFC device.

Table 3 lists all the JSR-257 API extensions and the level of support for each of them in the Nokia 6131 NFC SDK.

Extension	Description
com.nokia.nfc.nxp.simpletag	Provides an interface for accessing a SimpleTag. Full implementation included in the Nokia 6131 NFC SDK.
com.nokia.nfc.nxp.mfstd	Provides an interface for accessing a Mifare Standard card. Full implementation included in the Nokia 6131 NFC SDK.
com.innovision.rf	Provides an interface for accessing Innovision Jewel tags. Supported as a stub implementation in the Nokia 6131 NFC SDK.
com.nokia.nfc.nxp.desfire	Provides a connection interface and utility classes to be used when accessing Mifare DESFire cards. Supported as a stub implementation in the Nokia 6131 NFC SDK.
com.nokia.nfc.p2p	Provides an interface for communicating with NFCIP-1 devices. Supported as a stub implementation in the Nokia 6131 NFC SDK.
com.sony.felica	Provides an interface for accessing a NFC Forum Type 3 tag. Supported as a stub implementation in the Nokia 6131 NFC SDK.
com.nokia.nfc	Contains the NFCException interface that extends ContactlessException in order to provide access to the actual cause of an error. Supported as a stub implementation in the Nokia 6131 NFC SDK.

Table 3: JSR-257 extensions provided in the Nokia 6131 NFC SDK

3.3.1 SimpleTagConnection

In addition to connection interfaces provided by the JSR-257 API, the Nokia 6131 NFC SDK also contains the `SimpleTagConnection` interface, which is recommended to be used when exchanging data with Mifare Ultralight / Type 2 tag. The `SimpleTagConnection` interface is located within the `com.nokia.nfc.nxp.simpletag` package. The interface provides a full set of methods to handle the specific tag type, for example, reading and writing with logical or physical indexing, and locking and programming of the one-time programmable bits.

Class / Interface	Description
SimpleTagConnection	Interface that provides all the needed mechanism to communicate with non-DEF formatted data on Mifare Ultralight / Type 2 tag.

Table 4: SimpleTagConnection interface in the com.nokia.nfc.nxp.simpletag package

The following code snippet gives an example of how to create the `SimpleTagConnection` when `targetDetected` of `TargetListener` is called:

```
public void targetDetected(TargetProperties[] properties)
...
try {
TargetProperties target = properties[0]; // always at least one target
SimpleTagConnection conn = (SimpleTagConnection)
Connector.open(target.getUrl(SimpleTagConnection.class));
} catch (Exception e) {}
...
```

A `SimpleTagConnection` can be read and written in either physical or logical mode. In physical mode you can read and write to any block of the tag. The size of each block is defined in the constant `SimpleTagConnection.BLOCK_SIZE`. In logical mode you can read and write bytes in the user area of the tag.

To read and write blocks in physical mode, use the functions `read()` and `write()`. You cannot write partial blocks; the data is automatically padded with zeroes if its length is not a multiple of block size. You can get the number of available blocks with the function `getBlockAmount()`.

To read and write bytes in logical mode, use the functions `readLogical()` and `writeLogical()`. You can only write to the user area. Offset zero refers to the first byte of the user area. You can get the size of the user area with the function `getUserDataSize()`.

3.3.2 MFStandardConnection

The `MFStandardConnection` interface provides a starting point for accessing the Mifare Standard 1k and 4k Tags. The interface is located within the `com.nokia.nfc.nxp.mfstd` package. Table 5 summarizes all the interfaces and classes in the package.

Class / Interface	Description
`MFStandardConnection`	Interface representing the connection to a Mifare Standard 1k or 4k Tag.
`MFApplication`	Interface that provides access to a Mifare Standard Tag's application.
`MFApplicationDirectory`	Provides access to Mifare Standard Tag's application directory (MAD).
`MFBlock`	Base interface for handling Mifare Standard Tag blocks.
`MFDataArea`	Provides simple read/write functionality to data area(s) of a Mifare Standard Tag.
`MFManufacturerBlock`	Interface representing a Mifare Standard manufacturer block.
`MFSector`	Provides access to sectors within Mifare Standard Tags.
`MFSectorTrailer`	Provides access to a sector trailer.
`MFAccessBits`	Class for handling Mifare Standard Tag's access bits.
`MFKey`	Class representing Mifare Standard Tag authentication key.
`MFKey.KeyA`	Class representing Mifare Standard Tag authentication key type A.
`MFKey.KeyB`	Class representing Mifare Standard Tag authentication key type B.
`MFTrailerContents`	Describes the contents of a sector trailer, including authentication keys and access bits.
`MFValue`	Class representing the contents of a Mifare Standard Tag's value block.
`MFStandardException`	Thrown to indicate an error specific to the Mifare Standard Tag communication, that is, if authentication fails or if the method's conditions are not met.

Table 5: Classes and interfaces in the com.nokia.nfc.nxp.mfstd package

Figure 2 shows how the classes and interfaces relate to each other. The user can write directly to the tag by using the `MFStandardConnection.write()` method, but it is more advisable to use the special classes the API provides for accessing the tag. If you want to modify, for example, a block, you

can use the `MFStandardConnection.getBlock()` method to get a reference to an `MFBlock` object and then use `MFBlock.write()` to modify its content.

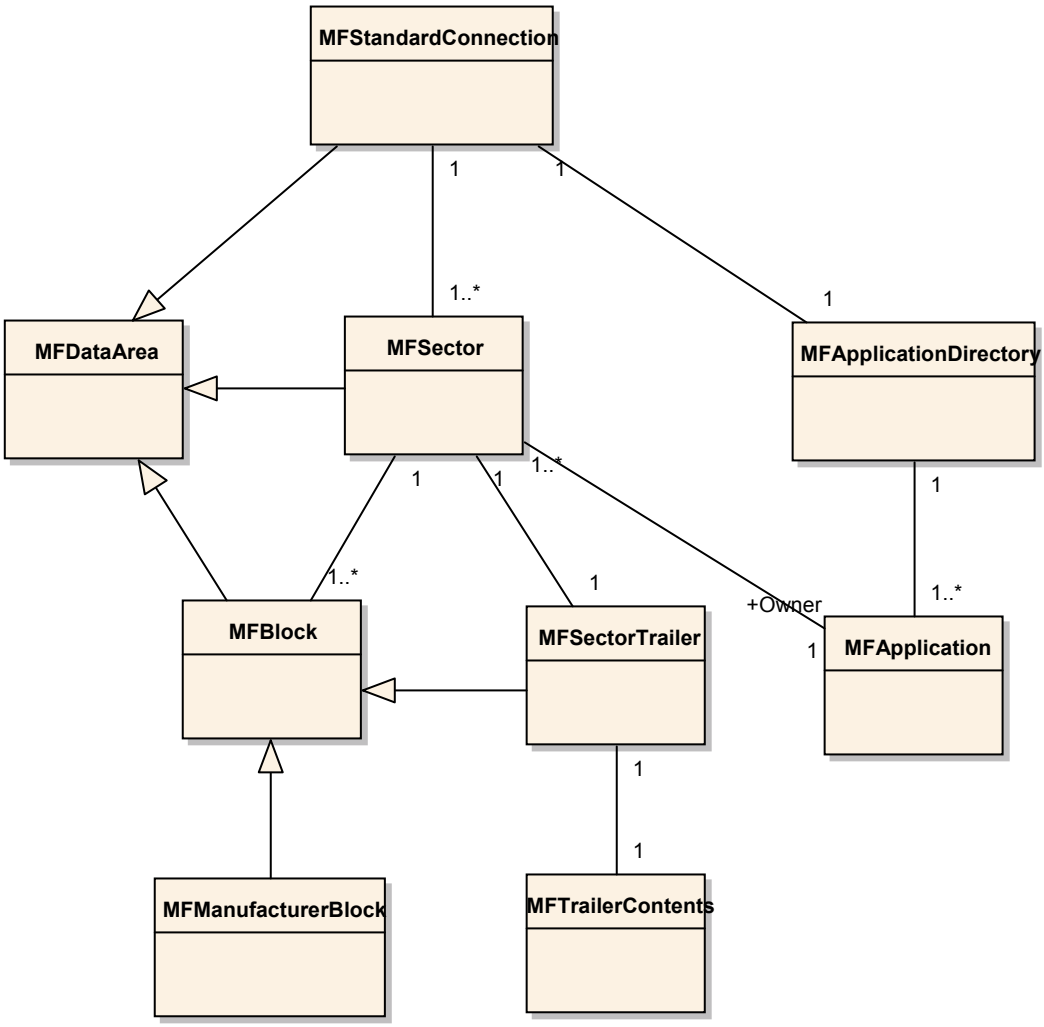

Figure 2: Class diagram of Mifare Standard API components

It is helpful to know the internal memory structure of the Mifare Standard Tags to be able to use the above-mentioned interfaces effectively. The Mifare Standard Tags use keys to provide a high level of security. Consequently their memory is organized in a different way than in the Mifare Ultralight Tags. Figure 3 illustrates the memory layout of a Mifare Standard 4k Tag.

Sector	Block	Byte number within a block 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Description
0	0																	Manufacturer Data
	1																	Data
	2																	Data
	3	Key A						Access Bits				Key B						Sector Trailer 0
1	0																	Data
	1																	Data
	2																	Data
	3	Key A						Access Bits				Key B						Sector Trailer 1
	..																	..
	..																	..
	..																	..
30	0																	Data
	1																	Data
	2																	Data
	3	Key A						Access Bits				Key B						Sector Trailer 30
31	0																	Data
	1																	Data
	2																	Data
	3	Key A						Access Bits				Key B						Sector Trailer 31
32	0																	Data
	1																	Data
	..																	Data
	..																	Data
	14																	Data
	15	Key A						Access Bits				Key B						Sector Trailer 32
	..																	..
	..																	..
	..																	..
39	0																	Data
	1																	Data
	..																	Data
	..																	Data
	14																	Data
	15	Key A						Access Bits				Key B						Sector Trailer 39

Figure 3: The memory layout of a Mifare Standard 4k Tag

The Mifare 4k Tag has a total of 40 sectors. The first 32 sectors contain 4 blocks while the last 8 are bigger and have 16 blocks each. The length of a block is always 16 bytes. The 1k tag has otherwise similar memory layout but it has only 16 sectors, which each contain 4 blocks.

To access a Mifare Standard Tag the first step is to set your application to listen for the tag or connect directly to the internal tag of the Nokia 6131 NFC device. The code below demonstrates how to listen for a tag and is the same as with NDEF tags but the target type is set to `TargetType.RFID_TAG`. Note that the class you register to `DiscoveryManager` should implement the `TargetListener` interface.

```
DiscoveryManager dm = DiscoveryManager.getInstance();
try
{
  dm.addTargetListener(this, TargetType.RFID_TAG);
}
catch (ContactlessException ce)
{
  // handle exception
}
```

When a tag is attached to a reader the `targetDetected` method is called by `DiscoveryManager`. We would now like to find out whether the new target can be accessed through `MFStandardConnection`.

The following code snippet goes through all the targets and returns a connection to the first target that supports `MFStandardConnection`, or `null` if there were no suitable targets. Note that one target can support multiple connection types. Mifare Standard Tags, for example, support both `MFStandardConnection` and `NDEFTagConnection`.

```
public MFStandardConnection getMFStandardConnection
    (TargetProperties[] tProp)
{
    for (int j = 0; j < tProp.length; j++)
    {
       Class[] connections = tProp[j].getConnectionNames();
       if(connections != null)
       {
          for (int i = 0; i < connections.length; i++)
          {
             if (connections[i].getName().equals(
                "com.nokia.nfc.nxp.mfstd.MFStandardConnection"))
             {
                try
                {
                   return (MFStandardConnection) Connector.open(tProp[j]
                      .getUrl(connections[i]));
                }
                catch (Exception e)
                {
                   // Handle exception
                }
             }
          }
       }
    }
    return null;
}
```

In the following example the `targetDetected` method uses the connection to find out how many blocks there are on the tag. The `MFBlock` interface is used to write a byte array to the beginning of block index 6. The blocks in this case are indexed from the beginning of the tag, so the block index 6 refers to block number 2 of sector number 1.

```
public void targetDetected (TargetProperties [] tProp)
{
    MFStandardConnection conn = getMFStandardConnection(tProp);
    if(conn == null)
       return;
    System.out.println("The tag has "+conn.getBlockCount()+" blocks.");
    MFBlock block = conn.getBlock(6);
```

```
    byte[] k = {(byte)0xff,(byte)0xff,(byte)0xff,(byte)0xff,
       (byte)0xff,(byte)0xff};
    MFKey.KeyA key = new MFKey.KeyA(k);
    byte[] data = {(byte)0x1, (byte)0x2, (byte)0x3};
    try
    {
       block.write(key, data, 0);
    }
    catch (Exception e)
    {
       // Handle exception
    }
    try
    {
       conn.close();
    }
    catch (IOException e)
    {
       // Handle exception
    }
}
```

The authentication key in the example above is set to its default value. If you want to know more about the Mifare Standard Tags, refer to *MF1 IC S50 Functional Specification* [13] and *MF1 IC S70 Functional Specification* [14]. The Nokia 6131 NFC JSR-257 Implementation document [15] provides more detailed information about the Mifare Standard API as well.

3.3.3 NFCIPConnection

NFCIP allows two devices to communicate with each other when they are within close range. The communication follows a simple request-response protocol.

The NFCIPConnection interface is located in the `com.nokia.nfc.p2p` package. A MIDlet can function either as an initiator or a target. An initiator establishes a connection, sends some data, and receives a response. A target establishes a connection, receives some data, and sends a response.

When establishing the connection, the URL specifies whether the MIDlet will be the initiator or the target. You have to use one of the predefined URLs:

```
String INITIATOR_URL = "nfc:rf;type=nfcip;mode=initiator";
String TARGET_URL = "nfc:rf;type=nfcip;mode=target";
```

The connection is opened in the usual way with `Connector`. The call to `open()` will block until a suitable peer is found. You cannot use `DiscoveryManager`/`TargetListener` to wait for a connection of this type. The connection is opened as follows:

```
NFCIPConnection conn = (NFCIPConnection) Connector.open(url);
```

When in initiator mode, the MIDlet should send and then receive data as follows:

```
byte[] message = ...
conn.send(message);
byte[] response = conn.receive();
```

When in target mode, the MIDlet should receive and then send data as follows:

```
byte[] message = conn.receive();
byte[] response = ...;
conn.send(response);
```

Finally, the connection should be closed:

```
conn.close();
```

3.4 Exchanging data with external smart cards

The use of smart cards improves a transaction's convenience and security. Smart cards also provide tamper-proof storage of user and account identity. The Contactless Communication API allows communication with external smart cards by providing a discovery mechanism for them. The actual communication with the ISO14443-4 and ISO 7816-4 compliant smart cards is done using APDU (Application Protocol Data Unit) commands. An APDU command contains either a command message or a response message sent from the interface device to the card or the other way around. A device that contains RFID hardware can emulate a contactless smart card to an external reader device.

There are also some smart cards based on ISO 14443-4 that do not require ISO 7816-4 communication. For these products some other means than APDUs are used to communicate with the card. With those cards, the ISO 7816-4 APDU is replaced by some other (typically proprietary) command format.

In the JSR-257 API, the `ISO14443Connection` interface provides access to the ISO 14443-4 compliant contactless smart card. This interface is located in the API package `javax.microedition.contactless.sc` (see Table 6).

Class / Interface	Description
`ISO14443Connection`	Interface that provides access to the ISO 14443-4 compliant contactless smart card

Table 6: ISO14443Connection interface in the javax.microedition.contactless.sc package

After the application has received a new event through the `TargetListener` interface, you can check whether the connection type is ISO-14443-4 compliant. The following code snippet demonstrates briefly how to open an ISO1443-4 connection and exchange data:

```
TargetProperties target = …

 …

Class[] classes = target.getConnectionNames();
byte[] commands = … // APDU commands

for(int i = 0; i<classes.length;i++)
{
   if (classes[i].equals(
   Class.forName("javax.microedition.contactless.sc.ISO14443Connection")
   ))
   {
     String url = target.getUrl(classes[i]);
     // Open connection to external smart card
     ISO14443Connection smc = (ISO14443Connection)Connector.open(url);
     byte[] response = smc.exchangeData(commands);
   }
}
```

In the example above, APDU commands are "coded" in a byte array and sent through the `ISO14443Connection` interface by using the `exchangeData` method. Basically, the APDU command structure consists of two parts: a mandatory header and a conditional body part. The header consists of 4 bytes (CLA, INS, P1, and P2). The body part consists of L_c, data and L_e fields. The number of bytes present in the data field of the command APDU is denoted by L_c. The maximum

number of bytes expected in the data field of the response APDU is denoted by L_e. When the L_e field contains only zeros, the maximum number of available data bytes is requested [10]. Figure 4 illustrates the structure of the APDU commands. For detailed information about forming the command, see the specifications of the smart card you are using.

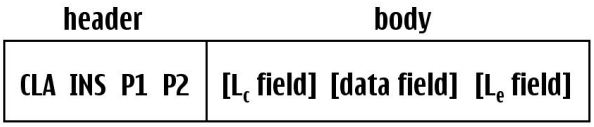

Figure 4: APDU command structure [10]

3.5 Connecting to internal card/tag of the Nokia 6131 NFC device

The Nokia 6131 NFC device has an internal tag and an internal secure card. `DiscoveryManager` does not report these to `TargetListener` objects since the cards are always connected. To connect to the tag or the secure card, you should read the URL from system properties and call `Connector.open()` with the returned URL. The internal tag URL is stored in property `internal.mf.url` and the internal secure card in property `internal.se.url`.

For example, connecting to the internal secure card can be done as follows:

```
ISO14443Connection connection = (ISO14443Connection)
Connector.open(System.getProperty("internal.se.url"));
```

4 Using branding configuration

Branding configuration is used for configuring branding elements, such as image, sound, or video. The branding elements are associated with the smart card application ID. When the Nokia 6131 NFC is being used as a contactless card, the external card reader (for example, ticket reader or point-of-sale device) requests for a specific application ID in order to complete a transaction with the Nokia 6131 NFC. Based on the application ID, the Nokia 6131 NFC is able to show/play the related branding elements.

Note: The emulator does not support showing or playing the branding elements.

The branding elements can be one of the following:

- combination of an image + sound
- video clip

Branding elements are supported on both displays. The following combinations are supported (MD = main display, SD = secondary display):

- MD video/swf
- MD video/swf + audio
- MD image + audio
- SD image + audio
- MD video/swf + SD image
- MD video/swf + SD image + audio
- MD image + SD image + audio

The specifications for the branding elements are:

- Image: JPEG, GIF, or PNG. Resolution 240x250 (for main display) and 128x160 (for secondary display).
- Audio: AAC, MIDI, or MP3. Duration of the sound clip should be 2–5 seconds.
- Video: 3GP. Resolution 176x144 or 128x96 (video supported on main display only).

The `BrandingConfiguration` class from package `com.nokia.nfc.dbapi` is used to configure the images and sounds for different smart card applications. To set the configuration for a smart card application, you need to know its application ID. To start configuring an application, you create a `BrandingConfiguration` object:

```
String appId = "0x1a2b3c4d5e6f"; // example application ID
BrandingConfiguration bc = new BrandingConfiguration(appId);
```

The next step is to load the image and audio files and set them to the `BrandingConfiguration` object. Your application needs to load the file into memory, but does not need to understand the format. To set the image of the main display you would do the following. The function `readFile()` in this example is assumed to return the content of a file.

```
byte[] mainDisplayContent = readFile("example.png");
bc.setElement(BrandingConfiguration.MAIN_DISPLAY, mainDisplayContent);
```

To set the image on a secondary display, use the constant `MINI_DISPLAY` and to set the sound file, use the constant `AUDIO`. Note that only the combinations listed above are valid. When all the branding elements have been set, the call to `save()` will save the data in the device memory:

```
bc.save();
```

5 Using the MIDP 2.0 PushRegistry

Applications can be started after a wake-up event to improve the usability and to minimize memory consumption in the device. This functionality is provided by the MIDP 2.0 PushRegistry.

Note: Testing MIDlet launch using PushRegistry is not supported in the emulator.

5.1 NDEF record push

Registration for startup is based on the record type name and format of the NDEF record. There can be one application for each record type name and format pair registered for startup at a time. If data in the target contains more than one NDEF record, the startup is based on the record type name and format of the first matching record in the data. Table 7 lists URL coding rules in BNF format.

Element	Coding
<ndef url>	::= "ndef:"<record_type_format>?name=<record_type_string>
<record_type_format>	::= "rtd" | "external_rtd" | "mime" | "uri"
<record_type_string>	::= String of US_ASCII characters, the fully qualified name of the record type

Table 7: NDEF record push URL in BNF format

6 Working with NDEF records

NFC Data Exchange Format (NDEF) is a lightweight binary message format. It encapsulates one or more NDEF records, each of which has payload data. The size of one payload data is limited to 2^{32}-1 bytes but records can be chained together to support larger payloads. However, the physical limits of user data in tags are:

- Mifare Ultralight: 48 bytes
- Mifare Standard 1k: 720 bytes
- Mifare Standard 4k: 2384 bytes

An NDEF record is constructed from three different factors: payload type, optional payload identifier, and payload data. In the JSR-257 API, the `javax.microedition.contactless.ndef.NDEFRecord` class represents NFC Forum's NDEF record. The `NDEFRecord` class allows you to construct NDEF records in two ways. You can either formulate a new NDEF record from a byte array or, in addition to a byte array, you can also define a specific record type to your record. The constructors you can use are:

```
NDEFRecord(byte[] data, int offset)
NDEFRecord(NDEFRecordType recordType, byte[] id, byte[] payload)
```

In the application, the `NDEFRecordType` class can be used to construct the appropriate payload type. For example:

```
new NDEFRecordType(NDEFRecordType.URI,
"http://schemas.xmlsoap.org/soap/envelope/");

new NDEFRecordType(NDEFRecordType.NFC_FORUM_RTD,
"urn:nfc:wkt:Sp"); // see RTD Smart Poster

new NDEFRecordType(NDEFRecordType.NFC_FORUM_RTD,
"urn:nfc:wkt:T"); // see RTD Text

new NDEFRecordType(NDEFRecordType.NFC_FORUM_RTD,
"urn:nfc:wkt:U"); // see RTD URI
```

The payload identifier may be given in the form of an absolute or relative URI, and it allows user applications to use a URI-based linking mechanism. However, NDEF does not define any particular linking mechanism or format. An application that repackages NDEF records should ensure that links are working.

Payload data on an NDEF Record (`NDEFRecord`) depends heavily on the payload type (`NDEFRecordType`). Table 8 presents the payload types available in the NDEF specification.

Name	NDEFRecordType constant	PushRegistry string	Example value
Empty	EMPTY	-	-
NFC Forum well-known type	NFC_FORUM_RTD	ndef:rtd?name=<NAME>	"T","U", and "Sp"
Media-type as defined in RFC 2046	MIME	ndef:mime?name=<MIME TYPE>	"image/jpeg","text/uri-list"

Name	NDEFRecordType constant	PushRegistry string	Example value
Absolute URI as defined in RFC 3986	URI	ndef:uri?name=<ABSOLUTE URI>	"http://schemas.xmlsoap.org/soap/envelope/"
NFC Forum external type	EXTERNAL_RTD	ndef:external_rtd?name=<NAME>	N/A
Unknown	UNKNOWN	-	-
Unchanged	-	-	-
Reserved	-	-	-

Table 8: Different payload types

There are a few well-known record type definitions (RTD) [7] available already. These are Text [8], URI [9], and Smart Poster [5]. The Text record type defines the record type for plain text data. The URI record type describes a record to be used with the NFC Data Exchange Format (NDEF) to retrieve a URI stored in a NFC-compliant tag or to transport a URI from one NFC device to another. The Smart Poster record type is used to put URLs, SMS messages, or phone numbers on an NFC Forum tag.

As an example, Section 6.1, "RTD Text record," describes the content of the RTD text record. A Well-Known Type defined by the NFC Forum is a URN [11] with the namespace identifier (NID) "`nfc`" [7]. The Namespace Specific String of the NFC Well-Known Type URN is prefixed with "`wkt:`". Thus, the NFC Forum Well-Known formed URN should start with the "`urn:nfc:wkt:`" prefix. For example, the Well-Known Type "`urn:nfc:wkt:T`" for text records would be encoded as "`T`".

Note: The MIDP 2.0 PushRegistry can be used to launch the appropriate user application with a specific payload type (see Section 5.1, "NDEF record push").

6.1 RTD Text record

The "Text" record contains freeform plain text. It can be used to describe, for example, a service or the contents of the tag. The Text record can be used as a sole record in an NDEF message. However, the Text record should be used in conjunction with other records to provide explanatory text. Table 9 and Table 10 define the structure of the Text record content.

Offset (bytes)	Length (bytes)	Content
0	1	Status byte. See Table 10.
1	n	ISO/IANA language code. Examples: "fi", "en-US", "fr-CA", "jp". The encoding is US-ASCII.
n+1	m	The actual text. Encoding is either UTF-8 or UTF-16, depending on the status bit.

Table 9: Text contents [8]

Table 10 describes the status bit encodings.

Bit number (0 is LSB)	Content
7	0: The text is encoded in UTF-8. 1: The text is encoded in UTF16.
6	Reserved (MUST be set to zero).
5..0	The length of the IANA language code.

Table 10: The status bit encodings [8]

6.2 Hello World example with NDEF

Now we can take advantage of the Text record and write a simple `NDEFRecord` containing the message "Hello World" as shown in the following code snippet:

```
NDEFTagConnection connection;
…
NDEFRecord hello = new Text("Hello world!");
NDEFMessage message = new NDEFMessage(new NDEFRecord[]{hello});
connection.writeNDEF(message);
```

The example above used a custom wrapper class, which converts a Text-based message to byte array.

```
class Text extends NDEFRecord {
    public Text(String content) { this(content, ""); }
    public Text(String content, String lang) {
      super(new NDEFRecordType(NDEFRecordType.NFC_FORUM_RTD,
"urn:nfc:wkt:T"),
      new byte[0], convert(content, lang));
    }
    private static byte[] convert(String text, String lang) {
     byte[] textBits, langBits;
      try {
        textBits = text.getBytes("utf-8");
        langBits = lang.getBytes("US-ASCII");
      } catch (UnsupportedEncodingException e) {
        throw new IllegalArgumentException("Cannot convert.
"+e.getMessage());
      }
      if (langBits.length > 0x3f)
        throw new IllegalArgumentException("Language tag too long.");
      byte[] ret = new byte[1+langBits.length + textBits.length];
    // set status bit
      ret[0] = (byte) langBits.length;
      // copy lang bytes
      for (int i=0; i<langBits.length; i++)
        ret[1 + i] = langBits[i];
      // copy text bytes
      for (int i=0; i<textBits.length; i++)
        ret[1 + langBits.length + i ] = textBits[i];
      return ret;
    }
 }
```

7 Example MIDlets

The Nokia 6131 NFC SDK contains six example MIDlets in the SDK's `examples` folder. These MIDlets demonstrate the use of JSR-257. See the release notes of the example MIDlets for a description of how to run them on the SDK.

These examples work with the internal smart card of the Nokia 6131 NFC.

7.1 SimpleNDEFExample

The SimpleNDEFExample MIDlet demonstrates how to read and write NDEF tags using `NDEFTagConnection`. It also uses the Personal Information Management API to add contacts read from the NDEF tag. The MIDlet has a `TargetListener` registered. For example, each time a simulated card is dragged to the emulator, a reading or writing operation is performed to the card. The operation that is performed depends on whether the user has selected to read or write from the main view.

7.2 TicketingExample

The TicketingExample MIDlet demonstrates how to read and write to both an internal and external secure card. With this MIDlet, you can check the balance of your internal card or external card. You can also load money into your internal card from an external card and buy a movie ticket from an external card. The example also demonstrates how to create a simulated smart card class.

7.3 InternalSecureCardExample

InternalSecureCardExample shows how to access and communicate with the internal secure element of the Nokia 6131 device. The example consists of an applet that is running in the secure element and a MIDlet that is running in the device. The MIDlet prompts the user to enter a name, and sends the name to the applet in the secure element, which replies with a greeting that is displayed to the user.

Note that the applet needs to be installed to the secure element prior to running this example. For more information, see Section 5.2 in Nokia 6131 NFC SDK: User's Guide [1].

7.4 MFStandardExample

MFStandardExample shows how to use the Mifare Standard API to write data to the tag and read the previously written data back. An alternative method for accessing the tag by using the `MFApplicationDirectory` interface is demonstrated as well.

7.5 P2PExample

The P2PExample MIDlet demonstrates how to communicate with another Nokia 6131 NFC device using `NFCIPConnection`. With this MIDlet you can send short messages to the peer and also receive them.

The MIDlet can operate in either initiator or target mode. In a peer-to-peer connection, one end should be in initiator mode and the other end in target mode.

7.6 BrandingExample

The BrandingExample MIDlet demonstrates how to use dynamic branding to associate images and sounds with a smart card application using `BrandingConfiguration`.

This example MIDlet allows you to enter an application ID and choose image and audio files that should be associated with the application. You can also use the MIDlet to remove the images and sounds associated with an application.

8 Terms and abbreviations

Term or abbreviation	Meaning
APDU	Application Protocol Data Unit is the protocol used to communicate with smart cards. Defined in the ISO 7816-4 specification.
API	Application programming interface
BNF	Backus-Naur form
Card	In addition to a memory, this contactless communication element also contains a processor.
GCF	Generic Connection Framework
JSR	Java Specification Request
MIDlet	Application that conforms to the MIDP standard.
MIDP	Mobile information device profile
NDEF	NFC Data Exchange Format
NDEF record	Data that is formatted according to the NFC Forum data format specification (NDEF). One record consists of the record type, record identifier, and the actual data of the record. Defined in the NFC Data Exchange Format specification.
NFC	Near Field Communication
NID	Namespace identifier
RFID	Radio frequency identification
RTD	Record type definition
SDK	Software development kit
Tag	Contactless communication element that contains accessible memory.
Target	Any device that supports some form of contactless communication.
URI	Uniform resource identifier
URL	Uniform resource locator

9 References

[1] Nokia 6131 NFC SDK: User's Guide, available on the Forum Nokia Web site and in the Nokia 6131 NFC SDK

[2] Nokia 6131 NFC User's Guide, URL: http://www.nokia.com

[3] JSR-257: Contactless Communication API, URL: http://jcp.org/en/jsr/detail?id=257

[4] ISO/IEC 14443-4:2001

[5] Smart Poster Record Type Definition, SPR 1.1, Technical Specification, URL: http://www.nfc-forum.org

[6] NFC Data Exchange Format (NDEF), NDEF 1.0, Technical Specification, URL: http://www.nfc-forum.org

[7] NFC Record Type Definition (RTD), RTD 1.0, Technical Specification, URL: http://www.nfc-forum.org

[8] Text Record Type Definition, RTD-Text 1.0, Technical Specification, URL: http://www.nfc-forum.org

[9] URI Record Type Definition, RTD-URI 1.0, Technical Specification, URL: http://www.nfc-forum.org

[10] ISO/IEC 7816-4, Information Technology – Identification Cards – Integrated Circuit(s) – Cards with Contacts – part 4: Interindustry commands for interchange.

[11] RFC 2141: URN Syntax, URL: http://www.ietf.org/rfc/rfc2141.txt

[12] Nokia 6131 NFC Device Details, URL: http://www.forum.nokia.com/devices/6131_NFC

[13] MF1 IC S50 Functional Specification, Mifare Standard 1k Tag, URL: http://www.nxp.com/acrobat_download/other/identification/m001052.pdf

[14] MF1 IC S70 Functional Specification, Mifare Standard 4k Tag, URL: http://www.nxp.com/acrobat_download/other/identification/m043531.pdf

[15] Nokia 6131 NFC JSR-257 Implementation, available in the Nokia 6131 NFC SDK

10 Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by rating this resource.