**Introduction**

This article is part of a series intended to act as an introduction to the world of NFC as it applies to BlackBerry smart phones such as the 9900 and is aimed at developers who wish to take advantage of this exciting new technology. Readers of this article should have some pre-existing knowledge of the fundamental architecture of NFC systems and be familiar with the BlackBerry WebWorks SDK. Familiarity with Java is required for those parts of the article which deal with the sample Web Works extension that illustrates how to add NFC NDEF capabilities to a WebWorks application.

Importantly, unless you already have good knowledge of the subjects covered, it’s recommended that you read the earlier articles in this series. Links to these articles can be found at the end of this one.

This article will look specifically at an example application that emulates a virtual NFC target. In effect allowing a BlackBerry java application to appear to a reader as a smart card and, building on previous articles, enabling a second BlackBerry device to read the virtual tag emulated on the original device.

**The Authors**

This article was co-authored by **Martin Woolley** and **John Murray** both of whom work in the **RIM Developer Relations** team. Both Martin and John specialise in NFC applications development (amongst other things).

**About this article**

In the previous articles we’ve shown how to read and write smart tags using the Java APIs available to java applications and also demonstrating how a WebWorks application could be extended to allow smart tags by using JavaScript extensions written in Java. However, we’ve only been dealing with smart tags up to now and it’s time to delve a little bit deeper into the APIs that are available to NFC applications.

To do this we’re going to examine, in a little more detail, some of the lower level APIs that underpin smart tags and understand how to exploit them. Specifically:

* Understand how to interact with a smart card using the ISO 14443-4 interface.
* Introduce the concept of Application Protocol Data Units (APDUs)
* Understand how to write a java application that will emulate a virtual smart card using the ISO 14443-4 interface
* Have some fun by using a second BlackBerry device to interact with another that is emulating a virtual smart card and exchange some data between the two using NFC.

We’ve already started to use some terminology such as ISO 14443-4 and APDU so first thing to do is describe what these are so that we can exploit them.

**Understanding the Standards**

Figure 1 demonstrates the relationship between two closely related sets of standards. ISO/IEC 7816 describes the characteristic of smart cards that use physical contacts. If you’ve ever looked at the contacts on your chip-and-PIN credit or debit card or at the contacts on your smart phone SIM card then this is the set of standards that define this interaction.

ISO/IES 14443, on the other hand, defines how “Contactless” cards interact, this means radio and in our context this means NFC radio.

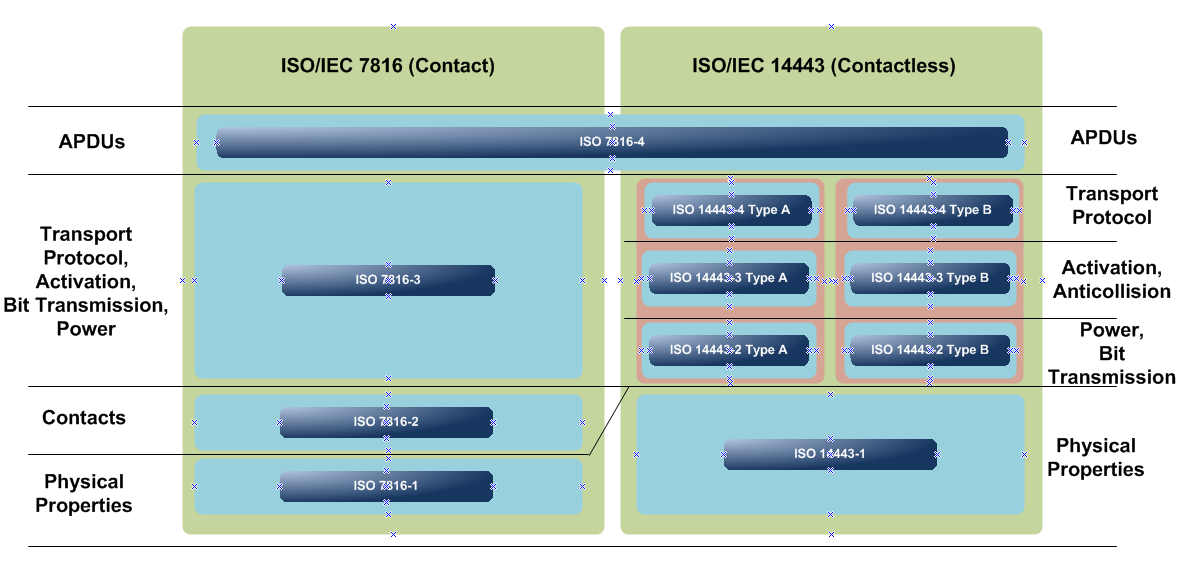


Figure 1 ISO IEC Contact and Contactless Standards

The important thing to take away from this is that the way that you communicate with such smart cards from an application is through a common interface at the very top level of this diagram called ISO 7816-4. If you drop below this layer things begin to become technology specific.

The messages that are defined by this standard and are passed between an application and a smart card are called APDUs (Application Protocol Data Units. If you’re familiar with communication protocol Standards then you ought to begin to see a parallel here. In fact they use the same terminology of the ISO 7-layer model.

The next important point to be aware of is that you can interact with smart cards can at a number of different levels within this model. We’ve already seen how to interact with a smart tag at the NDEF level where we deal with messages that contain various structures of text and URIs to convey information when the tag is read.

It’s possible to interact with smart cards at a number of levels but in this article we’re going to concentrate on interacting as an ISO 14443-4 target. This is effectively the “next layer down” as shown in Figure 2.

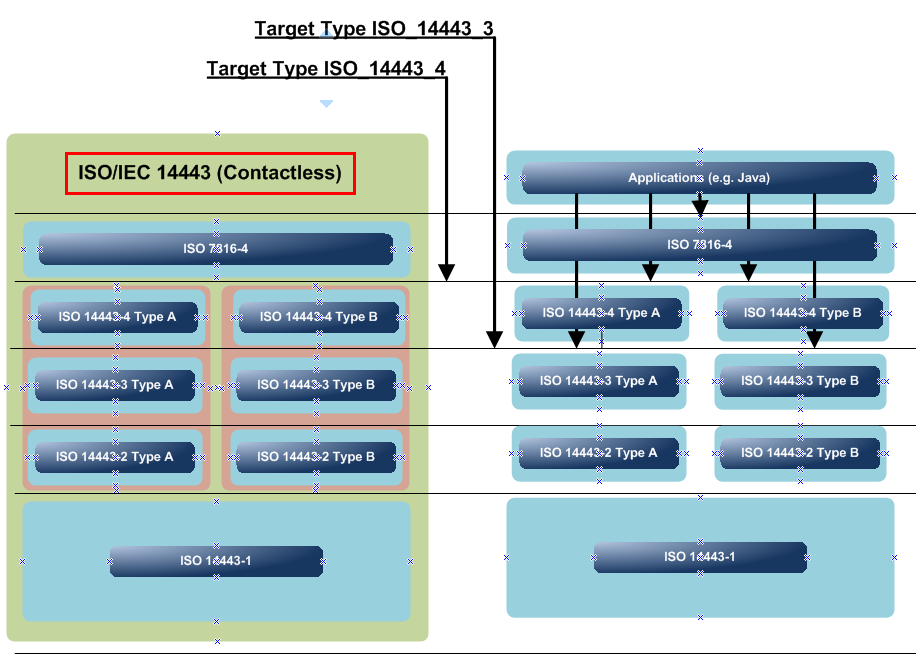


Figure 2 Target Types Defined

We’re not going to look at the lower level of ISO 14443-3. So, let’s just summarise what we’re going to do. We will:

* Write a java application that will make the BlackBerry device appear as an ISO 14443-4 virtual smart card (or, target) to an NFC reader device. This allows us to have much more control of the data that can be received and sent by the device.
* Also write a java application on another BlackBerry that will act as an NFC reader/writer to recognise the virtual smart card that is being emulated by the first BlackBerry.
* Exchange data between the BlackBerry device emulating the virtual smart card and the BlackBerry device acting as an NFC reader/writer

In fact we’ll be rather clever and incorporate these two features into a single application that can take on either role. Figure 3 shows the model of the application interaction that we are going to develop.

One important point to be aware of when using virtual emulation is that the virtual card will only be “visible” to an external reader through NFC when the application that is performing the emulation is in the foreground! A soon as the application gets placed in the background then emulation is stopped!

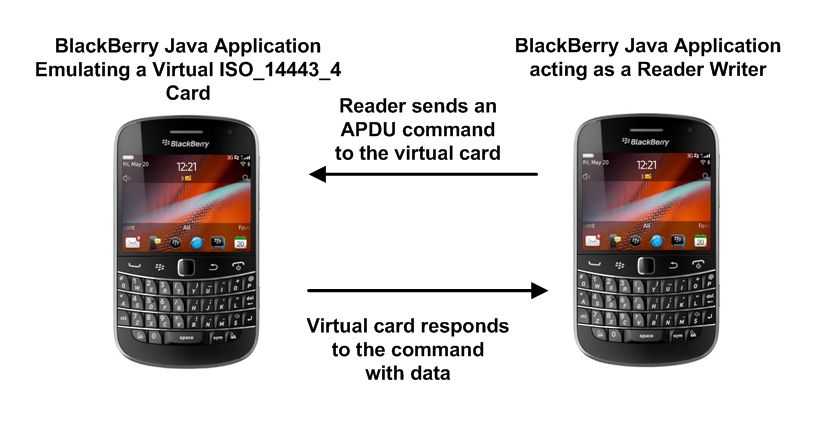


Figure 3 Application Interaction Model

It’s probably worth noting that this example of enabling two BlackBerry devices to communicate via NFC is for illustrative purposes and is not the most efficient way of doing this. Efficient Peer-to-Peer communication using NFC is a capability enabled by the LLCP and SNEP standards from the NFC Forum and these will be available in BlackBerry version 7.1.

Let’s start looking at the example application in more detail!

**Emulating a Virtual Card**

Starting emulation of a virtual target is really quite a simple task. Figure 4 shows the basics. First of all you need to create a new instance of **VirtualISO14443Part4TypeATarget()** and simply call its **startEmulation()** method. The name **VirtualISO14443Part4TypeATarget()** gives it away a bit – here we are requesting in emulating an ISO 14443-4 target that uses the Type A protocol as shown in Figure 2.

|  |
| --- |
| **protected** **void** startEmulationOfISO14443ATarget() {  **try** {  \_virtualISO14443ATarget = **new** VirtualISO14443Part4TypeATarget(  **new** NfcVirtTargListener(**this**),   *"12131415",*   **null**);  \_virtualISO14443ATarget.startEmulation();  } **catch**(NFCException e) {  e.printStackTrace();  } } |

Figure 4 Code fragment showing how to start emulation

The **VirtualISO14443Part4TypeATarget()** constructor can take three parameters. The first is an instance of a class that implements the **VirtualISO14443Part4TargetCallback** interface which is where the processing of events presented to the virtual card takes place. The second parameter becomes the identifier that uniquely identifies the virtual card whilst the third can be used to present **“historical bytes”** to the reader when the virtual card is detected. This is information that can be used to further characterise the capabilities of the virtual card – we don’t use them in this example.

|  |
| --- |
| **protected** **void** stopEmulationOfISO14443ATarget() {  **try** {  \_virtualISO14443ATarget.stopEmulation();  } **catch**(NFCException e) {  e.printStackTrace();  }  } |

Figure 5 Code fragment showing how to stop emulation

Of course you need to be able to stop virtual card emulation as well and Figure 5 shows how this is done.

Let’s take a closer look at the class **NfcVirtTargListener()** that we use to implement the interface VirtualISO14443Part4TargetCallback and where the main processing of events takes place. A fragment of this is shown in the figures below.

|  |
| --- |
| **package** nfc.sample.virtual.target;  **import** net.rim.device.api.io.nfc.emulation.VirtualISO14443Part4TargetCallback;  **import** net.rim.device.api.ui.UiApplication;  **import** net.rim.device.api.util.Arrays;  **public** **class** NfcVirtTargListener **implements** VirtualISO14443Part4TargetCallback {  **private** NfcVirtTargScreen \_screen;  **public** NfcVirtTargListener(NfcVirtTargScreen nfcVirtTargScreen) {  **this**();  **this**.\_screen = nfcVirtTargScreen;  }  **public** NfcVirtTargListener() {  **super**();  }  ... |

Figure 6 Implementing the VirtualISO14443Part4TargetCallback

Notice that this interface is agnostic as to whether Type A, or Type B protocol was used over the radio interface.

The first method of interest that we may want to implement is **onVirtualTargetEvent()**. This is called when events that reflect a change in state of the virtual card take place such as being activated by a reader. The details can be seen in Figure 7. In the full code of our example will do nothing more than log the event as having occurred.

|  |
| --- |
| **...**  **public** **void** onVirtualTargetEvent(**int** targetEvent) {  String displayableTargetEvent;  **switch** (targetEvent) {  **case** VirtualISO14443Part4TargetCallback.*DEACTIVATED*:  // do something with this event  **break**;  **case** VirtualISO14443Part4TargetCallback.*EMULATION\_STOPPED*:  // do something with this event  **break**;  **case** VirtualISO14443Part4TargetCallback.*SELECTED*:  // do something with this event  **break**;  **default**:  // do something else  **break**;  }  } ... |

Figure 7 The method “onVirtualTargetEvent()”

Detecting changes in state of the virtual card are fine, but the interesting part is the handling of commands that have been sent to the virtual card from a card reader/writer when the BlackBerry device enters into its radio field. Commands are presented via the **processCommand()** method and a very simple implementation is shown in the code fragment in Figure 8. The source code of the example that accompanies this article is essentially the same but is instrumented with code to display events on the BlackBerry device’s screen.

|  |
| --- |
| ...  **public** **byte**[] processCommand(**byte**[] command) {  String responseText = \_screen.getTextToSend();  **byte** [] responseTextBytes = responseText.getBytes();  **byte** [] responseBytes = **new** **byte** [responseTextBytes.length   + CommonADPUs.*goodResponseRapdu*.length];  **if** (Arrays.*equals*(command, CommonADPUs.*simpleISO14443Capdu*)) {  System.*arraycopy*(responseTextBytes, 0,   responseBytes, 0,   responseTextBytes.length);  System.*arraycopy*(CommonADPUs.*goodResponseRapdu*, 0,   responseBytes, responseTextBytes.length,   CommonADPUs.*goodResponseRapdu*.length);  **return** responseBytes;  } **else** **if** (Arrays.*equals*(command,  CommonADPUs.*ndefTagApplicationSelectCapdu*) ||  Arrays.*equals*(command,  CommonADPUs.*altNdefTagApplicationSelectCapdu*)) {     **return** CommonADPUs.*noNdefSupportRapdu*;   } **else** {  **return** CommonADPUs.*insNotSupportedRapdu*;  }   } } |

Figure 8 How to process a command presented to the virtual card

The basic processing that takes place is that commands arrive as **byte []** and these have to be parsed to determine how to respond. The example has been set up so that we can effectively roll our own simple commands and responses provided they adhere to the structure of commands and responses as defined in the **ISO 7186-4** specification which defines exchanges at the top of the stack as shown in Figure 1. That is they must take the form of **APDU**s. APDUs can be tedious to parse but for the sake of this example we will make use of only three simple ones that will demonstrate their use. These are shown in Figure 9.

The messages that flow between a reader and a smart card at the APDU level are in the form of commands and responses. The reader will send a command APDU (C-APDU) to the card and the card will answer with a response APDU (R-APDU).

C-APDUs have a well defined format. An example of a simple one is demonstrated by the **byte [] simpleISO14443Capdu** in Figure 9. It has a number of fields:

* A **Class** field (**CLA**) – this contains various flags that control the interpretation of the subsequent instruction (**INS**) itself. In the example in Figure 9 it identifies the **INS** as a proprietary one rather than one of the standard ones defined in the **ISO 7816-4** specification.
* An **Instruction** (**INS**) field – this contains the specific operation that is to be performed by the card. Examples might be “**READ**”, “**SELECT**”, etc. In this case I’ve arbitrarily chosen my own instruction, **0x37** in hexadecimal notation, since this is a sample application. A real application would use the standard instruction sets for the card in question.
* Parameter **P1** and **P2** fields – these would carry any additional qualification information specific to the instruction (**INS**). In this example they are not used
* An optional **Command Length** (**L(c)**) field – if the command carries optional data L(c) would indicate the length if the data. L(c) may not be present if there is no data to be carried by the command. In this example I’ve added a payload of **0x99** just to demonstrate the use of the L(c) field which has a value **0x01**.
* An optional **Response Length Expected** field (**L(e)**) – if present this identifies the upper bound on the length of the expected response. In this example I’ve coded a value of **0x00** for L(e) which, confusingly, indicates a response of up to 256 (decimal) bytes may be expected.

You really need to read the ISO 7816-4 specification if you want to understand how to encode all the permutations of length and data fields.

If we now revisit Figure 8 we can see that the **byte []** that’s received is checked against the command that we are prepared to handle. If it matches, a response is built and returned, whilst in all other cases a specific rejection response is sent.

Let’s look at response now; or rather R-APDU formats. AN R-APDU is really very simple. It consists of:

* An optional **Response Data** field -- this represents any data that the command expects to be returned. The absence of an **L(e)** field in the original command indicates that no response data is expected and this field is missing in the response.
* Two **Status Word** (**SW1** and **SW2**) fields – these are always present and reflect information that indicates the success or failure of the original command. The standard identifies the particular (**SW1**, **SW2**) response of (**0x90**, **0x00**) as indicating **SUCCESS**. For convenience the **byte [] CommonADPUs.goodResponseRapdu** in Figure 9 is set up this way.  
    
  You will notice that there is another R-APDU defined in this code fragment: **byte [] CommonADPUs.noNdefSupportRapdu**. The specific value (**0x6A**, **0x82**) of (**SW1**, **SW2**) is defined by the NFC Forum Type 4 Tag specification as indicating that: “**NDEF is not supported by this card**”.

So, why do we want to use this? Well, you’ll see when you run the sample code that accompanies this article that the BlackBerry device that is acting as the reader (Figure 3) will attempt to probe the virtual card that is being emulated to try and see if it is actually an NDEF smart tag. It does this by sending a series of commands defined by the NFC Forum Type-4 Tag specification to figure this out. The C-APDU command that we check for in this case is

* **byte [] CommonADPUs.ndefTagApplicationSelectCapdu**

You can examine the sample code that comes with this example to see the exact form of this C-APDU.

It’s not the intent of this article to examine how to interact with an NDEF smart tag at the NDEF level using low level commands so we simply respond with **byte [] CommonADPUs.noNdefSupportRapdu** as a simple way of responding to the reader if the C-APDU matches one of the NFC Forum Type 4 commands.

Any other C-APDU that we do not understand or do not want to handle is rejected with the **byte [] CommonADPUs.insNotSupportedRapdu** response for (**SW1**, **SW2**) of (**0x6D**, **0x00**) which is a standard form for the “Instruction Not Supported” response.

|  |
| --- |
| **package** nfc.sample.virtual.target;  **public** **interface** CommonADPUs {  **public** **static** **final** **byte**[] *simpleISO14443Capdu* = **new** **byte**[] {   (**byte**) 0xA0 // CLA - proprietary command type  , (**byte**) 0x37 // INS - arbitrary instruction ( ‘cos it’s proprietary )  , (**byte**) 0x00 // P1 - not used  , (**byte**) 0x00 // P2 - not used  , (**byte**) 0x01 // L(c) - length of 1 byte   , (**byte**) 0x99 // command payload (1 byte just for fun )  , (**byte**) 0x00 // L(e) ( up to 256 expected in response )  };  **public** **static** **final** **byte**[] *goodResponseRapdu* = **new** **byte**[] {   (**byte**) 0x90 // SW1 ...  , (**byte**) 0x00 // SW2 ... All goodness! Deep joy!  };  **public** **static** **final** **byte**[] *noNdefSupportRapdu* = **new** **byte**[] {   (**byte**) 0x6A // NDEF Tag Application not found  , (**byte**) 0x82 // ... " ...  };  **public** **static** **final** **byte**[] *insNotSupportedRapdu* = **new** **byte**[] {  (**byte**) 0x6D // SW1 ...  , (**byte**) 0x00 // SW2 ... INS Not Supported  }; ... ... ... } |

Figure 9 Some simple APDUs

So, with this new information at hand we can take another look at Figure 8 and now understand the logic. All commands received are rejected with either the reason: “NDEF not supported”, or “INS not supported”; and the single command that we’re interested in (our own proprietary one) is responded to positively by taking the contents of a field on the screen of the BlackBerry device and constructing an R-APDU with the field contents as the response data.

The net result is that the reader will receive this data in response to the command that is has issued.

So, how could we test the behaviour of this application? Well, if you have access to a contactless card reader connected to your PC then you could make use of a tool called **GPShell**. GPShell is a tool that be downloaded from SourceForge (<http://sourceforge.net/projects/globalplatform/files/> ) and is commonly used to perform operations on smart cards that conform to the **Global Platform** set of standards. In this simple example we can use it to send our simple proprietary APDU to the BlackBerry device and examine the response.

To perform this test the application has been started on the BlackBerry device to emulate a virtual target and the device placed on the contactless card reader connected to a PC. For example, in Figure 10 you can see a BlackBerry 9900 device running this application placed on a contactless NFC reader and connected to a PC using a USB cable.



Figure 10 Using a BlackBerry 9900 with a contactless reader

The GPShell command line tool is used to send our own proprietary APDU ( *0xA037000001990* ) to the device as shown in Figure 1.

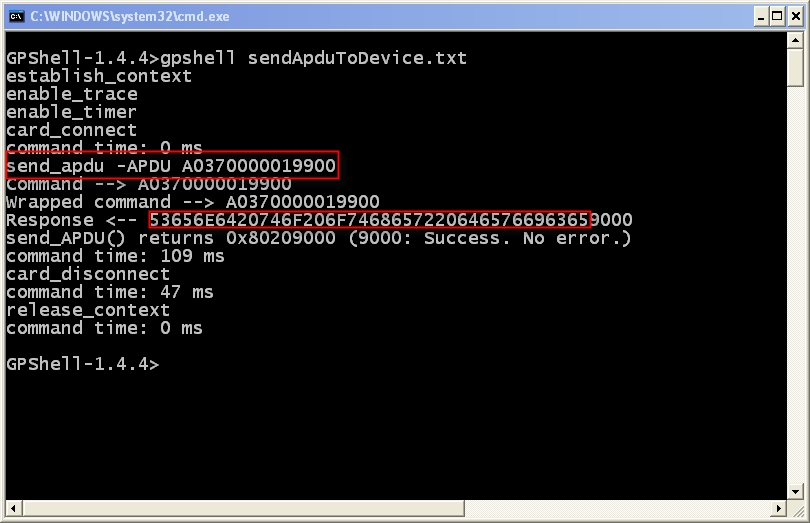


Figure 11 Using GPShell to test the application

The response payload that comes back from the device is 0x53656E...6365 along with the SW1/SW2 status of 0x9000 indicating successful execution of the command.

If you examine the response payload (0x53656E...6365) you’ll see that it represents the string: *“Send to other device”* which is the text that the application on the BlackBerry device sent in response to our command.

So, what we’ve done is verify that the BlackBerry device is, in fact, correctly emulating a virtual target and responding as expected to APDUs that are sent to it.

On that note, let’s now take a look at the other side of this example application. Let’s see how to write the reader side of the application which will interact with the virtual card in the other BlackBerry device in the same way that we’ve just tested with a real contactless reader and the GPShell tool.

**Interacting with the Virtual Card**

Now, we’ve actually seen how to do this before. We use exactly the same technique that was used in the previous article called **“Reading and Writing NFC Smart Tags”**. The model is that we register an interest in being notified of targets (smart card, tag, or a device emulating a card or tag) entering the radio field of the BlackBerry device. In this case we’re interested in detecting the presence of the other BlackBerry device that is emulating an ISO-14443-4 target.

Figure 10 demonstrates the same technique that we have used in the earlier article of creating a class that implements the **DetectionListener** interface.

|  |
| --- |
| **package** nfc.sample.virtual.target;  **import** java.io.IOException;  **import** java.util.Enumeration;  **import** javax.microedition.io.Connector;  **import** net.rim.device.api.io.nfc.readerwriter.DetectionListener;  **import** net.rim.device.api.io.nfc.readerwriter.ISO14443Part4Connection;  **import** net.rim.device.api.io.nfc.readerwriter.Target;  **import** net.rim.device.api.ui.UiApplication;  **public** **class** NfcVirtDetectionListener **implements** DetectionListener {  **private** NfcVirtTargScreen \_screen;  **public** NfcVirtDetectionListener(NfcVirtTargScreen screen) {  **this**();  **this**.\_screen = screen;  }  **public** NfcVirtDetectionListener() {  **super**();  }  ... |

Figure Implementing the DetectionListener

This means that the **onTargetDetected()** method will be called when the other BlackBerry device enters the radio field of the this device. Figure 11 shows some of the basic checking that we can do at this point. In particular it demonstrates how to obtain the values of the various properties that are associated with the target that’s just been detected.

|  |
| --- |
| ...  **public** **void** onTargetDetected(Target target) {  **if**( target.isType(Target.*ISO\_14443\_3*) ||   target.isType(Target.*ISO\_14443\_4*) ||   target.isType(Target.*NDEF\_TAG*) ) {  Enumeration targetProperties = target.getProperties();  **if**(targetProperties != **null**) {  **for** (Enumeration e = targetProperties; e.hasMoreElements();) {  String targetPropertyName = (String) e.nextElement();  String targetPropertyValue =  target.getProperty(targetPropertyName);  **if**(!targetPropertyName.equals("SerialNumber")) {  // do something with targetPropertyName  // ... and targetPropertyValue  } **else** {  // handle serial number differently  **byte**[] serialNumber = targetPropertyValue.getBytes();  }  }  }  } ... |

Figure Detection of a suitable target

The really interesting part is show in Figure 12. Here we ensure that the target just detected is of ISO 14443-4 type. We only want to send APDUs to the other BlackBerry device if the card type it’s emulating is capable of receiving APDUs. Of course we’ve constructed the example in this way but we should verify this in any case.

Now that we know the target is of the correct type we obtain a URI that references this target and open an **ISO14443Part4Connection** to it. Sending a C-APDU and receiving an R-APDU in response is a synchronous operation and achieved using the **ISO14443Part4Connection.tranceive()** method.

The C-APDU that we send is the proprietary one that we crafted in Figure 9, namely **CommonAPDUs.simpleISO14443Capdu**. The response from the target is returned in the field **byte[] tranceiveResponse**.

Recall that an R-APDU should always have two status fields (**SW1**, and **SW2**) at the end of the message. So, any response that is less than two byes in length will be ignored in this simple example.

For any other R-APDU the fields (**SW1**, and **SW2**) should be checked. In our simple example any status other than (**0x90**, **0x00**) – that is, the “*Happy Path”* defined as part of the ISO 7816-4 standard – would indicate an error condition.

A status of (**0x90**, **0x00**) indicates that the payload of the R-APDU can be used and it ought to be the message that was sent from the other BlackBerry device that is emulating a smart card!

Parsing such messages really means manipulating **byte []** objects as shown below.

|  |
| --- |
| ...  **if**(target.isType(Target.*ISO\_14443\_4*)) {  String targetURI = target.getUri(Target.*ISO\_14443\_4*);  ISO14443Part4Connection targetISO14443Part4Connection = **null**;  **byte[] tranceiveResponse** = **null**;  **try** {  targetISO14443Part4Connection =   (ISO14443Part4Connection)Connector.*open*(targetURI);  tranceiveResponse =  targetISO14443Part4Connection.transceive(  CommonADPUs.*simpleISO14443Capdu*);  **if**(tranceiveResponse.length > 2) {  **byte**[] statusCode = **new** **byte**[2];  **byte**[] responseMessage =   **new** **byte**[tranceiveResponse.length - statusCode.length];    System.*arraycopy*(tranceiveResponse, 0,   responseMessage, 0,   responseMessage.length);  System.*arraycopy*(tranceiveResponse, responseMessage.length,   statusCode, 0,   statusCode.length);  // do something with the message we’ve just received  // from the other BlackBerry device: responseMessage  // is the “statusCode” is 0x9000  } **else** { /\* ignore \*/ }  } **catch**(IOException e) {  e.printStackTrace();  }  } **else** { /\* ignore \*/ }  }  } |

Figure Sending the APDU command and receiving a response

The example code that is associated with this article was used to construct the fragments in the preceding sections. Some logic such as some error handling and logging has not been displayed in order to make the overall logic easier to grasp and understand.

**Running the Sample Application**

If you run the sample code in two BlackBerry devices and follow the instructions, bringing them together then they will exchange data that has been entered into a field on the screen of the device that is emulating a virtual card and this will be displayed on the screen of the other device that is acting as the reader.

In fact very much like this! Figure 13 shows the screen of the BlackBerry that is emulating a virtual tag after the two BlackBerry devices have successfully exchanged data and then separated. The messages relating to **“Deactivated”** and **“Emulation stopped”** are because the devices have separated from one another and are no longer in NFC radio range.

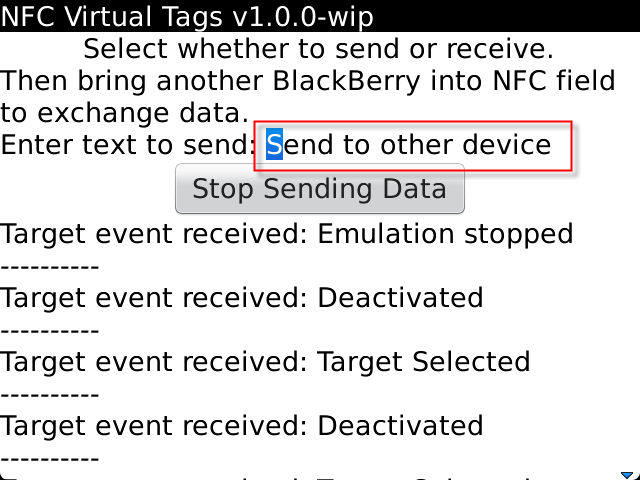


Figure BlackBerry emulating a virtual tag

So can see that the text “Send to other device” has been entered into a text field on the screen of the device performing the virtual emulation whilst in Figure 14 the same text has been read on the other device and displayed on the screen along with a successful (0x90 0x00) APDU response code!

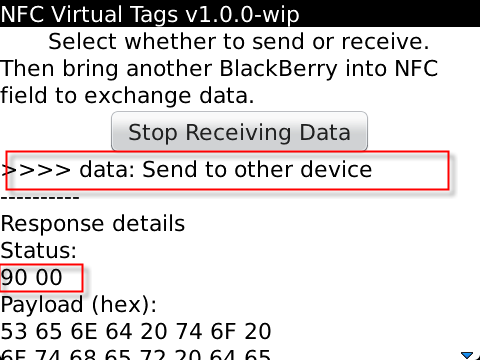


Figure BlackBerry reading the tag presented by the other device

**Summary**

Hopefully this article has given you some insight into how you can use Virtual Target Emulation as part of your BlackBerry application. The APIs are available today so download the latest BlackBerry®﻿﻿ JDK from

<http://us.blackberry.com/developers/blackberry7/>

BlackBerry Java APIs are documented here if you'd like to browse further:

<http://www.blackberry.com/developers/docs/7.0.0api/>

The NFC Forum Home Page can be found here: [http://www.nfc-forum.org](http://www.nfc-forum.org/)

# Other BlackBerry Developer NFC Articles

NFC Primer for Developers: <http://supportforums.blackberry.com/t5/Java-Development/NFC-Primer-for-Developers/ta-p/1334857>

Reading and Writing NFC Smart Tags: <http://supportforums.blackberry.com/t5/Java-Development/Reading-and-Writing-NFC-Smart-Tags/ta-p/1379453>

NDEF Tag Reading from Web Works Applications: <http://supportforums.blackberry.com/t5/Java-Development/NDEF-Tag-Reading-from-Web-Works-Applications/ta-p/1430431>

# Glossary of NFC Terms

* **APDU** - Application Protocol Data Unit. A message type which forms part of a protocol and which may be exchanged between peers as either a request or a response message. Applications on a BlackBerry smart phone may communicate with applets in an SE using the ISO7816-4 APDUs for example.
* **CLF** - Contactless Front-end. Part of the NFC Controller. Communicates via RF with an NFC reader.
* **HCI** - Hardware Controller Interface. Amongst other things, this component of the NFC system architecture redirects radio communication to appropriate SE.
* **ISO7816-4** - the specification which defines the protocol which allows applications to communicate with applets installed in an SE or smart card.
* **NDEF** - NFC Data Exchange Format
* **NFC** - Near Field Communications
* **PCD** – Proximity Coupling Device (also known as “card reader”)
* **PICC** – Proximity Integrated Circuit Card
* **SE** - Secure Element. A hardware component which can host and act as an execution environment for small software applications known, in this context, as "applets".
* **SIM** - Subscriber Identity Module. In 2G modules this used to be synonymous with the SIM \*card\* i.e. the small smart card inserted in the handset. In 3G networks, the SIM is a software module installed on the UICC.
* **SWP** - Single Wire Protocol.
* **UICC** - Universal Integrated Circuit Card - the smart card used in mobile terminals in GSM and UMTS networks