How to Acquire and Publish Content from / to NFC Tags and Proximity Peers

This article explains how to use Windows (Phone) 8 Proximity APIs to interact with Near Field Communication (NFC) tags and devices

Introduction

The Proximity framework in the Windows 8 Platform is about enabling “Tap and Do” actions. One of the main Near Field Communication (NFC) scenarios involves acquiring content from NFC tags and other peer devices. To talk back, you can also publish messages to other peers, or even write NFC tags.

This article gives an overview of the possibilities as well as snippets how to get started with developing proximity apps. It’s based on the slides of the Windows 8 Platform NFC Development lecture, but adds all the needed background information.

Acquiring Content from NFC Tags and Peers

Content can be acquired in a multitude of forms, for different scenarios. The simplest possible case is a tag that contains a link to your company website. A more involving type is to offer a visitor more information by tapping, e.g., for a piece of art in a museum or to buy a concert ticket by tapping the poster. For a higher impact advertising solution, it’s also possible to create custom app extensions that can be enabled by tapping; for example to unlock a bonus item in a game by touching a tag contained in a magazine, or a plush toy in the form of an additional bird.

Let’s take a look at acquiring content from the developer’s perspective. You mainly need to remember two classes: `ProximityDevice` and `ProximityMessage`. The first will create the connection to the hardware, detect devices as they come in proximity range and allow you to publish & subscribe to messages. Those messages are then delivered by the `ProximityMessage` class. Before you can use those classes in your app however, you need to activate the “Proximity” capability in your manifest file.

Subscribing to URIs

The most straightforward use case for acquiring content is subscribing for URIs. First, activate the proximity device – usually, it’s fine to simply get the default. We’re using the variable name `_device` here – create the according instance variable in your class. Then, choose to subscribe for the `WindowsUri` message type and specify your callback handler method. The return value is a numeric subscription ID, which you can use to unsubscribe again at a later stage, as well as to find out which subscription the message is coming from in case you use multiple subscriptions with the same handler method.

```
_device = ProximityDevice.GetDefault();
_subscribedMessageId = _device.SubscribeForMessage("WindowsUri",
MessageReceivedHandler);
```

Through the second parameter in the handler method (message), your code will get the message that it received – in case of the `WindowsUri` type, it’s directly the URI, no matter how it’s actually stored on the NFC tag. You only need to convert the Unicode-encoded data of the Proximity message to a string / URI:

```
private void MessageReceivedHandler(ProximityDevice sender, ProximityMessage message)
{
    var msgArray = message.Data.ToArray();
    var url = Encoding.Unicode.GetString(msgArray, 0, msgArray.Length);
    Debug.WriteLine("URI: " + url);
}
```

When reading an NFC tag, it will most likely contain a standardized NDEF message (more on that later), not a `WindowsUri` type. Luckily, when subscribing for `WindowsUri`, your app will get informed about the URI contained on the tag, no matter if it’s formatted as a URI NDEF record or a Smart Poster. So you can ready every standard tag with the code above and don’t need to worry about its format. However, you’ll miss additional information stored in Smart Posters, like the title text.

 (C) Copyright Nokia 2014. All rights reserved.

To unsubscribe in case you don't need the message subscription anymore, just call the following method with the URI you got when subscribing (which we also stored in a member variable):

```
_device.StopSubscribingForMessage(_subscribedMessageId);
```

Spread the Word – Publish & Write

Instead of just receiving content, your app might also want to inform other peers or write information back to tags. Publishing to other peers can also be used for pushing content to another instance of your app running on another device, through using a custom URI scheme that your app is registered for (a custom URI scheme is for example skype:username?call instead of http://www.skype.com/).

Publishing to Peers

Sending out a URI to peers is even easier than subscribing for messages, and can be done with just one line of code (after acquiring the default proximity device, as shown in the previous section):

```
_publishingMessageId = _device.PublishUriMessage(new Uri("http://nfcinteractor.com"));
```

Note that this call will just send the URI over to peer devices, but will not write the URI to tags. Communication to other devices is encapsulated in the standardized SNEP protocol, so most other NFC devices and phones are able to receive and understand the message; they don't have to be Windows-based.

Specifically, your Windows device will actually send two NDEF messages, each containing one record:

1. A standardized URI NDEF record. This one can be read by any other device and contains just the URI you specify in your code.
2. An NDEF record of type `windows.com/UriSchemeToAppHint` (type name format: Absolute URI), which contains the scheme, platform and app ID. This type is obviously specific to Windows, and helps other Windows devices finding an app that can handle a custom URI scheme. Read more about custom URIs and launching apps.

Writing Tags

If you'd rather like to write the message to a tag, you need to call a slightly different method. Most of the work before that call is to convert the URI string to a byte array in the mandated little endian UTF-16 encoding.

```
var dataWriter = new Windows.Storage.Streams.DataWriter {
    UnicodeEncoding = Windows.Storage.Streams.UnicodeEncoding.Utf16LE };
dataWriter.WriteString("http://nfcinteractor.com [This link is external to TechNet Wiki.
It will open in a new window.] ");
var dataBuffer = dataWriter.DetachBuffer();
_device.PublishBinaryMessage("WindowsUri:WriteTag", dataBuffer);
```

In contrast to the `PublishUriMessage()` method, which sends two messages to peers, the `PublishBinaryMessage()` with the type `WindowsUri:WriteTag` only writes a single standardized NDEF URI message to the tag, and doesn't add a Windows specific message. Keep in mind that the size of a typical NFC tag is limited to very few bytes, so every single character matters and a second message might not even fit on the tag.

Tag Size

If you want to find out the writable tag size before attempting to store your information on it, you can subscribe to a special type: `WriteableTag`.

```
_device.SubscribeForMessage("WriteableTag", MessageReceivedHandler);
```

Whenever the `ProximityDevice` will discover a tag that is suitable for writing information, your handler will be called, and the

 (C) Copyright Nokia 2014. All rights reserved.

`ProximityMessage` contains the writable tag size in bytes:

```
var tagSize = BitConverter.ToInt32(message.Data.ToArray(), 0);
Debug.WriteLine("Writeable tag size: " + tagSize);
```

While we're at it: you can also find out some information about the proximity hardware – most useful will be the transfer rate; the driver specifications mandate a minimum of 16kB/sec. Available as well is the maximum size of a message that you can publish (>= 10kB).

```
var bps = _device.BitsPerSecond; // >= 16kB/s
var mmb = _device.MaxMessageBytes; // >= 10kB
```

NDEF Handling

We've seen NDEF messages and records before; so what is NDEF? The NFC Forum standardized the NFC Data Exchange Format, which essentially defines how contents have to be formatted in the world of NFC.

Each NDEF record has a header, which contains – amongst other bits of information – the type of the record. For example the URI record has the name "U". The URI record definition then also specifies how the payload of the record has to look like; for example, the URI itself needs to be stored in UTF-8 format. To save some precious space on a tag, common URI protocols and prefixes are shortened. This allows encoding the long and expensive "http://www." into a single byte in the header.

The Smart Poster can then be seen as a meta record, which wraps a URI with some other bits of information; most commonly one or more title texts, in various languages.

An NDEF message can then be thought of as a box that contains several individual NDEF records, to also let the reader know when to stop reading the tag.

Subscribing to the WindowsUri type will read URI records and Smart Posters – and give you only the URI. What if you'd like to read the remaining information of a Smart Poster, or subscribe to a totally different NDEF record? This is the corresponding subscription call:

```
_subscribedMessageId = _device.SubscribeForMessage("NDEF", MessageReceivedHandler);
```

Your handler will receive call-backs for all discovered NDEF messages, no matter which type. The parameter will then contain the raw payload of the message. It's up to you to figure out the contents of the message; e.g., if it's a simple URI, you need to convert the shortened byte back to the full "http://www.", as defined in the NFC Forum specifications. That's a lot of work and requires much low-level code, working with bits and bytes. Even parsing a simple URI record is already more than one page of source code.

NDEF Library for Proximity APIs

To save you from wasting your time with implementing NDEF message parsing according to technical specification white papers, you can download and use the open source NDEF Library.

Written in C#, it can convert the raw byte array that you get from the Proximity APIs into a NDEF message. Specialized classes for each record type will allow you to simply extract the information present in the record, without needing to know how exactly it's encoded in the raw data.

Additionally, the library allows creating NDEF messages, which can then easily be written to a tag or published to another peer.

The library is released under the LGPL license, which is one of the most permissive open source licenses. It allows using the library in closed source, commercial applications as well. The library is partly based on the Connectivity module of the Qt Mobility library, which is also licensed under LGPL.

To read and parse a Smart Poster from the raw byte array you get from the handler callback of the proximity APIs, the following code snippet is enough. Of course, you can also extract more information from the `NdefSpRecord`, like additional titles, the size of the linked content, etc. – depending on what is actually defined on the tag.

 (C) Copyright Nokia 2014. All rights reserved.

```
// Parse raw byte array to NDEF message
var msg = NdefMessage.FromByteArray(rawMsg);

foreach (NdefRecord record in msg)
{
    // Go through each record, check if it's a Smart Poster
    if (record.CheckSpecializedType() == typeof (NdefSpRecord))
    {
        // Convert and extract Smart Poster info
        var spRecord = new NdefSpRecord(record);
        Debug.WriteLine("URI: " + spRecord.Uri);
        Debug.WriteLine("Titles: " + spRecord.TitleCount());
        Debug.WriteLine("1. Title: " + spRecord.Titles[0]);
        Debug.WriteLine("Action set: " + spRecord.ActionInUse());
    }
}
```

If on the other hand you'd like to write a Smart Poster to a tag, the following piece of code first creates the corresponding record and adds a single title to it. Then, it encapsulates the record in a NDEF message and converts it to a byte array. This is then published with the Proximity APIs using the type `NDEF:WriteTag`, which allows writing raw NDEF messages.

Note: Be careful with the raw byte arrays – if your NDEF message contains errors, the APIs might refuse writing them to the tag, and not necessarily inform you about the issue. Therefore, it's recommended to use the NDEF Library, which always creates standard-compliant messages.

```
// Initialize Smart Poster record with URI, Action + 1 Title
var spRecord = new NdefSpRecord {
    Uri = "http://nfcinteractor.com",
    NfcAction = NdefSpActRecord.NfcActionType.DoAction };

spRecord.AddTitle(new NdefTextRecord {
    Text = "Nokia", LanguageCode = "en" });

// Add record to NDEF message
var msg = new NdefMessage { spRecord };

// Publish NDEF message to a tag / phone
// AsBuffer(): add -> using System.Runtime.InteropServices.WindowsRuntime;
_device.PublishBinaryMessage("NDEF:WriteTag", msg.ToByteArray().AsBuffer());
```

References

This article is based on the corresponding presentation by Andreas Jakl and provides background explanations that you wouldn't get when only browsing through the slides. The slides can be viewed and downloaded here: http://slidesha.re/UJAefK

To find out more about Proximity APIs, check out the API documentation.

For some cases also helpful is the specification for Proximity driver implementers – you should not need to know it, but it might help with some background knowledge if you want to go to lower levels or need details about encodings or limitations.

The NDEF Library for Proximity APIs can be downloaded for free from CodePlex: https://ndef.codeplex.com/

http://developer.nokia.com/community/wiki/How_to_Acquire_and_Publish_Content_from_/_to_NFC_Tags_and_Proximity_Peers (C) Copyright Nokia 2014. All rights reserved.

http://developer.nokia.com/community/wiki/How_to_Acquire_and_Publish_Content_from_/_to_NFC_Tags_and_Proximity_Peers (C) Copyright Nokia 2014. All rights reserved.