NFC Usage in Windows* Store Apps – a Healthcare App Case Study

Abstract

Modern mobile apps take advantage of a myriad of sensor types available on the platform. NFC is one such feature that is becoming increasingly popular, as it is very versatile and allows several types of use cases. In this article we will look at how a sample healthcare app uses NFC to enhance the user experience and enable new usage models. By the end of this article, you will learn how to add NFC usage to Windows* Store apps. Specifically, we will cover how to do protocol activation, how to automatically open your app when a user taps a custom-programmed NFC tag, and how to use a NFC tag as a check-in/check-out mechanism in a hypothetical patient room.

Contents

Abstract.................................................................................................................................................1
Contents................................................................................................................................................1
Overview ................................................................................................................................................2
NFC and protocol activation in Windows Store Apps............................................................................2
A Healthcare Line of Business Windows Store App................................................................................3
Adding NFC to a sample healthcare app – a case study.......................................................................4
    Adding Proximity Capability to the app Manifest................................................................................5
    Adding Protocol Activation Extension to the app Manifest.................................................................5
    Handling Protocol activation inside the app .........................................................................................6
    Detecting NFC availability and Subscribing for message(s)..............................................................7
    Reading and parsing NFC tag information.........................................................................................8
Summary................................................................................................................................................10
Overview

Near Field Communication (NFC) enables short-range wireless connectivity with data speeds of 106, 212, or 424 kbps and requires the device(s) to be in close proximity (e.g.: less than 4 cm). The connection is quick, simple, and automatic. In addition, the connection requires little configuration on the user’s part, unlike other connectivity technologies like Bluetooth. This renders it extremely convenient for different use cases.

At a higher level, NFC usage can be divided into three use cases: acquiring Information (e.g.: read URI from NFC tag), exchanging information (e.g.: send/receive photo), and connecting devices (e.g.: tap device to configure Bluetooth or other connection configuration). These three categories together can enable a plethora of NFC use cases. For an in-depth discussion on NFC technology, please refer to this article:


NFC usage is getting more popular by the day. Most of the new generation mobile hardware supports NFC. In this article, we will discuss how NFC enables new user experiences in a sample healthcare app. We will focus on how to use NFC tags for automatically activating (opening) our sample healthcare app and how to use the information embedded in the tag to identify and take additional steps.

NFC and Protocol Activation in Windows Store Apps

Windows Store apps can use NFC functionality via the Windows.Networking.Proximity namespace. The proximity namespace classes allow a device to discover another device nearby, and publish/subscribe messages between them[?].

The reference for the Windows Store apps proximity namespace is given below.


Before we can use the proximity APIs, we first need to declare the ‘proximity’ capability in the app manifest, which allows Windows to enforce security and user permissions for an app. Additionally, the app must be running in the foreground for it to be able to use proximity related APIs. In the
In the following sections, we will walk through these steps as part of a healthcare sample case study. For a detailed reference on NFC in Windows Store apps, please refer to the following article.


We will also take advantage of another feature in Windows Store apps—protocol activation. Protocol activation allows us to register our app to be activated for a particular URI scheme. We can even define our own custom URI scheme that our app registers for. This URI can be fed to the device from anywhere—including a tap of NFC tag, which we will use in our case study. Please refer to the reference below for more details on protocol activation in Windows Store apps.


A Healthcare Windows Store App

As seen in several other articles in this forum, we will build the case study around a healthcare Line of Business Windows Store app. We will extend it with the capability to do protocol activation and use NFC tags to implement a sample patient room check-in/check-out mechanism.

Some of the previous articles include:

- Porting App Life Cycle, Settings and Search Features from iOS* to Windows* 8
- Porting the User Interface from an iOS* Basic App to Windows* 8
- Porting Advanced User Interface from iOS* to Windows* 8
- Porting iOS* Custom Charts and Graphs to Windows* 8 UI

The application allows the user to login to the system, view the list of patients (Figure 1), and access patient medical records, profiles, doctor’s notes, lab test results, and vital graphs.
Figure 1: The “Patients” page of the Healthcare Line of Business app provides a list of all patients. Selecting an individual patient provides access to the patient’s medical records.

Adding NFC to a Sample Healthcare App – a Case Study

In this sample app, we can use NFC tags for uniquely identifying patient or lab room(s) that the logged-in user (healthcare provider or doctor) visits. When the user is about to enter the room, he/she can tap on the NFC tag at the entrance. If our sample app is not in the foreground, Windows will automatically activate it (via protocol activation), thereby bringing it to the foreground. We could additionally navigate to the appropriate screen inside the app depending on the reason for app activation (in this case NFC). Next, the app can read the room number embedded inside the NFC tag and log the timestamp and room details for check-in. When the user is about to leave the room, he/she will tap again and the app will log the check-out details. All results will be summarized on the user’s home screen.

Before we start adding these features to our sample app, first we need to modify the app manifest.
Adding Proximity Capability to the app Manifest

Double clicking on the *Package.appxmanifest* in your Visual Studio® 2012 project should bring up a manifest UI allowing you to tweak the settings. Figure 2 shows the Proximity capability enabled in the manifest, which lets us use the NFC feature.

![Figure 2: App manifest showing the Proximity package capability (captured from Visual Studio® 2012)](image)

Our project should now be ready to access the NFC feature.

Adding Protocol Activation Extension to the app Manifest

We also want our sample healthcare app to be activated (open the app, bringing it to the foreground) whenever we tap a NFC tag. This is very useful for enhancing the user experience since
the NFC-based proximity namespace classes only work when our app is in the foreground. To achieve this feature, we will need to enable protocol activation.

In the app manifest window, click on “Declarations.” Here we can add a new “protocol” declaration for our sample app. We can define our own custom URI scheme; in our sample app we use ‘prapp’ as the URI that we register for. You can choose the custom URI depending on your app requirements. Please refer to screen shot in Figure 3.

![Figure 3: Protocol declaration for sample app (captured from Visual Studio* 2012)](image)

Our sample app should now be ready for NFC and protocol activation features.

**Handling Protocol activation inside the app**

Our sample app will get invoked automatically every time a URI with the ‘prapp’ scheme is triggered on the device. In our case study, we custom program a NFC tag with a URI scheme of the format ‘prapp://rm=????’ where “????” is any patient or lab room number. When a user taps on this NFC tag,
Windows automatically reads in the URI, notices the ‘prapp’ URI scheme, and triggers activation for our sample app. The activation ‘kind’ is ‘Protocol’. To handle the app activation, we need to override ‘OnActivated’ method in our Application class. Please refer to the code listing below.

```csharp
// protocol activation
protected async override void OnActivated(IActivatedEventArgs args)
{
    if (args.Kind == ActivationKind.Protocol)
    {
        ProtocolActivatedEventArgs protocolArgs = args as ProtocolActivatedEventArgs;
        await InitMainPage(args);

        var frame = Window.Current.Content as Frame;
        var loggedinuser = SessionSettingsViewModel.SessionSettings.Loginuser;
        if (loggedinuser == null || !loggedinuser.LoginState)
        {
            frame.Navigate((typeof(Login));
            return;
        }
    }
    if (frame.CurrentSourcePageType != typeof(UserPage))
    frame.Navigate((typeof(UserPage));
```

Figure 4: Handling the app activation for custom URI protocol scheme that we registered for ++

Our sample app gets activated only when our specified URI scheme is triggered. Inside the ‘OnActivated’ method we also check for the kind of activation. If it is ‘Protocol,’ we can proceed to redirect the user to the appropriate UI screen.

In this case study, we first check if the user is already logged into the app. If the user is not logged in, we redirect the user to the login page. If the user is already logged in, we redirect the user to his/her home screen where he/she can see the NFC room information summarized. Depending on the app requirements, additional checks and verifications can be performed at this step.

**Detecting NFC availability and subscribing for message(s)**

Using the Windows Runtime proximity namespace classes, we can detect the presence of NFC capability on the device. If NFC capability is present, we can subscribe for specific types of messages. We can use the `ProximityDevice.GetDefault` static method for the default instance of NFC on the device.
In our case study, we subscribe for messages of type WindowsUri. Please refer to the article linked below for different message types supported.


Below is a sample code snippet from the case study.

```csharp
protected override void OnWindowCreated(WindowCreatedEventArgs args)
{
    var proximityDev = ProximityDevice.GetDefault();
    if (proximityDev != null)
    {
        nfcmsgsubid = ProximityDevice.GetDefault().SubscribeForMessage("WindowsUri", MessageReceivedHandler);
        System.Diagnostics.Debug.WriteLine("prxDev found and registered");
    }
}

private void MessageReceivedHandler(ProximityDevice sender, ProximityMessage message)
{
    PRAppUtil.ShowMsg("URI: " + message.DataAsString);
}
```

Figure 5: Sample code snippet for NFC check and subscribe message ++

Additional steps can be performed in the Message handler, depending on your app requirements.

**Reading and parsing NFC tag information**

Since we encoded the patient or lab room information as part of our custom URI (on NFC tag) itself, we can parse the URI to decode the room number and log the check-in details. When our sample app gets activated, the full absolute URI that was obtained from the NFC tag is passed on to the OnActivated method call via the Uri.AbsoluteUri property of the args.

The code snippet below parses the room information and updates the user view model with check-in and check-out details.

```csharp
protected async override void OnActivated(IActivatedEventArgs args)
```
if (args.Kind == ActivationKind.Protocol)
{
    ProtocolActivatedEventArgs protocolArgs = args as ProtocolActivatedEventArgs;
    await InitMainPage(args);

    var frame = Window.Current.Content as Frame;
    var logginuser = SessionSettingsViewModel.SessionSettings.Loginuser;
    if (logginuser == null || !logginuser.LoginState)
    {
        frame.Navigate(typeof(Login));
        return;
    }
    if (frame.CurrentSourcePageType != typeof(UserPage))
    {
        frame.Navigate(typeof(UserPage));
    }

    int rm = Convert.ToInt32(protocolArgs.Uri.AbsoluteUri.Split('=')[1]);
    if (ru.rm != 0 && ru.rm != rm) ru.rm = 0;
    if (ru.rm == 0)
    {
        ru.rm = rm;
        ru.cin = DateTime.Now;
    }
    else
    {
        ru.cout = DateTime.Now;
        RoomUsageViewModel.AddRoomUsage(ru.rm, ru.cin, ru.cout);
        ru.rm = 0;
    }
}

Figure 6: Sample code for parsing the NFC URI for room information and updating user view model

In our case study, we have a separate user home page where all the room check-in/check-out information is displayed. It automatically gets updated every time the user taps the room identifier NFC tag with our custom URI scheme. Please refer to the screen shot in Figure 7 for additional details.
Summary

NFC enables several types of use cases. In this article, we discussed how a sample healthcare app incorporates NFC into its workflow to provide an enhanced user experience and enable new types of usage models. We discussed how you can easily add protocol activation and simple NFC usage to your Windows Store apps.

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH
PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED
WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES
RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY
PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR
INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A
SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must
not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.”
Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or
incompatibilities arising from future changes to them. The information here is subject to change without
notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may
cause the product to deviate from published specifications. Current characterized errata are available on
request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing
your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel
literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel
microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific
computer systems, components, software, operations, and functions. Any change to any of those factors may
cause the results to vary. You should consult other information and performance tests to assist you in fully
evaluating your contemplated purchases, including the performance of that product when combined with
other products.

Any software source code reprinted in this document is furnished under a software license and may only be
used or copied in accordance with the terms of that license.

Intel, the Intel logo are trademarks of Intel Corporation in the US and/or other countries.

Copyright © 2013 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

Optimization Notice
This sample source code is released under the Intel OBL Sample Source Code License (MS-LPL Compatible), Microsoft Limited Public License, and Visual Studio* 2012 License.