WP8 NFC Tutorial: Voice Messages on NFC Tags

After you have completed this tutorial, you will have created an app that uses speech recognition to record voice messages, stores them on NFC tags, can launch itself from tags via LaunchApp tags and a custom URI scheme, and then speaks the message using speech synthesis!

Introduction

Interested in creating your first NFC app for Windows Phone 8? After following the steps below, you will have an app similar to NearSpeak, which is available in the WP Store and allows to record voice messages using speech recognition, store them on NFC tags, launch the app by tapping the tag and then playing back the voice message using speech synthesis.

This tutorial is based on my NFC session first presented at the Wowzapp Hackathon in Vienna (9. - 11. November 2012). The slides are available at SlideShare, the instructions in this article will help you walk through the tutorial even without attending a live presentation and contains all the details and tricks that would usually only be shown on stage and are not included on slides.

Project Setup

Create a new *Windows Phone App* project, call the app `NearSpeakTutorial` and choose *Windows Phone OS 8.0* as target platform. Our application will not run on Windows Phone 7, as we will use a lot of the new functionality found in the latest version of the OS: NFC, speech synthesis and speech recognition. Note that for developing, you should have a WP8 phone, as the emulator does not currently support simulating NFC.

After Visual Studio has finished creating your project, open the file `Properties/WMAppManifest.xml` from the *Solution Explorer* window. Here, you can define further properties of your app, like the description and author. Navigate to the *Capabilities*-tab and add the following capabilities to your project:

- Proximity (`ID_CAP_PROXIMITY`)
- Microphone (`ID_CAP_MICROPHONE`)
- Speech Recognition (`ID_CAP_SPEECH_RECOGNITION`)
- Networking (`ID_CAP_NETWORKING`)

Next, go to the *Requirements* tab and activate the NFC hardware requirement (`ID_REQ_NFC`). Our app will only work on phones that have NFC hardware, which is not a requirement for WP8 devices. Most phones will feature NFC, but there might be devices coming up without. Of course, if you create an app that uses NFC as an optional feature, you can leave the requirement deactivated and check for NFC support at runtime.

Initialization

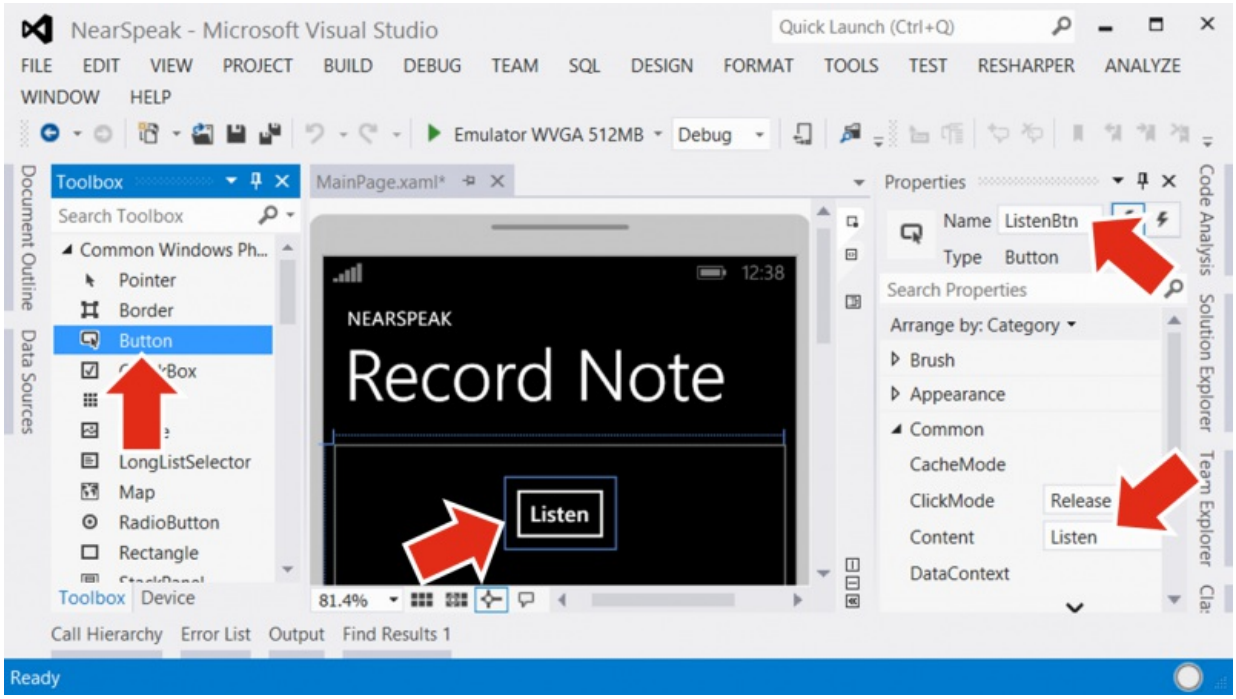
Next, you need to initialize the API classes of Windows Phone that we will use for writing to the NFC tag, as well as for speech recognition and the speech synthesizer. As we will use those classes throughout the app, it's best to define them as new member variables in `MainPage.xaml.cs`, and to initialize them in the constructor. You also need to add the relevant using statements for the classes we need; especially if you use Visual Studio Extensions like *ReSharper*, a lot of this work can be automated.

```
// Using declarations, at the top of MainPage.xaml.cs
using Windows.Networking.Proximity;
using Windows.Phone.Speech.Recognition;
```

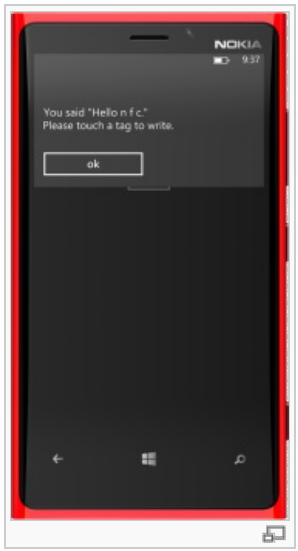
```
using Windows.Phone.Speech.Synthesis;

// Member variables declarations, at the top of the MainPage class definition
private ProximityDevice _device;
private SpeechRecognizer _recognizer;
private SpeechSynthesizer _synthesizer;

// Instantiate the classes in the MainPage() constructor
_device = ProximityDevice.GetDefault();
_recognizer = new SpeechRecognizer();
_synthesizer = new SpeechSynthesizer();
```

`ProximityDevice` is the class to interact with the NFC hardware. On the phone, there is usually only one NFC device present - so it's fine to just get the default device. On a Windows 8 desktop, there might be multiple NFC or other proximity devices - you would then typically ask the API to list all the available proximity devices and choose the most suitable one.

Next up, create the UI design for the app. Open `MainPage.xaml`, which will bring you to the UI Designer. Drag a Button from the *Toolbox* to the page. In the *Properties* pane, give the new button the name `ListenBtn` and the content `Listen`. The following screenshot highlights the steps and changes:



Starting Speech Recognition



After we have completed the basic project setup, we can now start implementing the functionality. Double-click on the button border to let Visual Studio automatically create an event handler for the clicked event. In this method, we will start the speech recognition. We will use the speech recognition without a pre-defined UI, so for the final app you should add status information to inform the user about the progress.

Speech recognition is an asynchronous process. One of the nice new features of the latest C# version is the new `async` / `await` pattern. Instead of making your code more difficult to maintain by introducing a lot of callback methods, you can simply await the asynchronous process. Windows Phone will continue executing your app and does not block its execution. However, the method will only continue to execute once the async process has finished. Essentially, C# splits your method in two behind the scenes, but saves you from doing that yourself.

Once we got the result of the speech recognition, we'll show it in a message box to inform the user. The first time the user starts speech recognition, the phone will show a dialog asking to accept the speech privacy policy. When doing free-form speech recognition, Windows Phone requires a network connection and uses a Microsoft web service to recognize the speech. If you limit the

speech to just a few pre-defined words, you can also do offline speech recognition. In our case, we want the user to say anything he likes, so we use the online variant.

```
private async void ListenBtn_Click(object sender, RoutedEventArgs e)
{
    // Start speech recognition and wait for the async process to finish
    var recoResult = await _recognizer.RecognizeAsync();

    // Inform the user about the result and the next steps
    MessageBox.Show(string.Format("You said \"{0}\"\nPlease touch a tag to write the
message.", recoResult.Text));
}
```

Writing the message to an NFC tag

If you would like devices to be able to recognize and automatically act upon the data you stored on the NFC tag, you have to use a standardized format. In the world of NFC, this has been standardized through the NDEF format. Essentially, a header in the tag tells the reading device how to interpret the payload that follows afterwards.

LaunchApp Tags

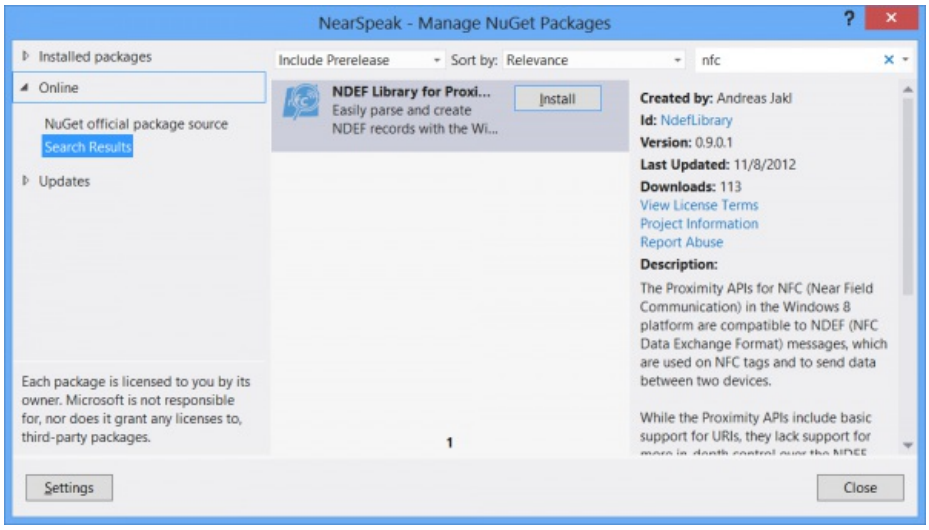
In our case, we will first use the Microsoft LaunchApp type, which directly links to your application and also passes arguments. It is the best and most direct way to launch your app on Windows, but is problematic for cross-platform scenarios if you plan to extend your app to other platforms later. This is why we will extend this app to also react to a custom URI scheme in a later step, which makes cross-platform tags easier.

You can write LaunchApp tags directly using the Windows Proximity APIs. However, it is easier to use the free and open source NDEF Library for Proximity APIs. It's released under the LGPL license, so you can use it for free, even in closed source apps. Additionally, the library will make it easier to write more advanced NFC tag contents later on, e.g., to store multiple records on the tag in case you want to port the app to Android and also store an Android Application Record on the same tag.

Importing the NDEF Library can be done in a matter of seconds. But first of all, you need to ensure your NuGet Package manager is up-to-date. The manager takes care of downloading libraries for you, integrates those into your project and even keeps them up-to-date. A great timesaver! The pre-installed NuGet package manager version can not handle portable class libraries like the NDEF Library, which are compatible to both Windows Phone 8 and Windows 8. Therefore, you need to update to the latest version of the NuGet manager (>= 2.1). Update through: Tools -> Extensions and Updates... -> Updates (left sidebar) -> Visual Studio Gallery

Once you made sure that NuGet is the latest version, you can proceed to installing the NDEF Library. This is done in three simple steps:

1. Tools -> Library Package Manager -> Manage NuGet Packages for Solution...
2. Search "Online" for "NDEF"
3. Install the "NDEF Library for Proximity APIs (NFC)"



Preparing the Message

Now the NDEF library is ready to use! We just need to define the contents and write them to the tag. The NDEF format is designed in a way that an NDEF message can contain one or more NDEF records. You can think of the message as a box or container, where you place one or more items inside. In our case, we will just include a single item - the LaunchApp record. Add this code snippet to the same method that we have used before.

```
// Create a LaunchApp record, specifying our recognized text as arguments
var record = new NdefLaunchAppRecord { Arguments = recoResult.Text };
// Add the app ID of your app!
record.AddPlatformAppId("WindowsPhone", "{...}");
// Wrap the record into a message, which can be written to a tag
var msg = new NdefMessage { record };
```

The LaunchApp record type can contain arguments that will be passed to the app when it's launched. In our case, we will use the text that the user spoke. Additionally, you need to define the application ID, so that the phone will know which app to launch (or to download from the store in case it's not yet installed on the phone). You can find the app ID of your app if you go back to the `WMAppManifest.xml` file, switch to the *Packaging* tab and copy the *Product ID* to the clipboard. Now, go back to your code in `MainPage.xaml.cs`, and replace the {...} text with your product ID.

Writing to the Tag

Actually writing to the tag is very simple and can be done with a single line of code:

```
// Write the message to the tag
_device.PublishBinaryMessage("NDEF:WriteTag", msg.ToByteArray().AsBuffer(),
MessageWrittenHandler);
```

We use our ProximityDevice to publish the message. The first parameter specifies the action you would like the NFC driver to perform. The proximity APIs opted to supply this type as a string - most likely this decision was made for easier extensibility and more flexibility when it comes to different device drivers from manufacturers. You can get an overview of different strings to write as the publish type at MSDN.

As the second parameter, we need to send the raw data to write to the tag. With the `ToByteArray()` method, you can convert the NDEF message to a byte array. The APIs can not directly work with a byte array, so you need to create a buffer based on the array first. The `AsBuffer()` method does that for you. To use this method, you need to include the following using statement:

```
// For AsBuffer() method to convert a byte array (byte[]) to an IBuffer
using System.Runtime.InteropServices.WindowsRuntime;
```

The last parameter of the `PublishBinaryMessage()` method is the callback function that will be called when writing the tag was successful. Here, we specified a method called `MessageWrittenHandler`, which we also need to implement:


```
private void MessageWrittenHandler(ProximityDevice sender, long
messageId)
{
    // Message was written successfully - inform the user
    Dispatcher.BeginInvoke(() => MessageBox.Show("NFC Message
written"));
}
```

The callback would be executed in an extra thread. In Windows Phone, it is not possible to interact

with the user interface from any other thread than the one and only user interface thread. The `Dispatcher.BeginInvoke()` code ensures that the code we use to show the message box is executed in the user interface thread instead of the call-back thread.

What happens if writing the tag was not successful, for example when the tag was too small or is not writable? Unfortunately, the Proximity APIs do not inform the app about that. You only get a call-back if writing was successful. The only possible workaround is to also register for the `DeviceArrived` event, and then start a timer. If the message wasn't written after around 1-2 seconds, you know that something went wrong - but not why.

Windows Phone in general is compatible to *NFC Forum Type 1 - 4* tags, some phones might also be able to write to *Mifare Classic* tags. The tags need to be NDEF Formatted (which they have to be if they are advertised as NFC Forum tags). Unfortunately, Windows Phone can not currently format factory empty tags; so make sure you buy the right tags from the factory.

Speak When Launched

The last step until we are finished is to actually speak the text when the app is launched. The Windows Phone operating system handles the LaunchApp tags by default, so we do not worry about that. The platform will launch our app (if the user allows this to happen) and sends us the arguments that we have written to the tag.

The app gets the arguments through the page navigation parameters used in Windows Phone. In general, a WP app is built using one or more pages (XAML pages). The app navigates between those pages and sends parameters from one page to the next. This approach is a bit similar to navigating between different HTML pages, where you can also pass parameters using the URL.

The arguments from the LaunchApp tag are encoded into the query string, the pre-defined key name is `ms_nfc_launchargs`. This key name will always be used by the operating system for LaunchApp types; we just need to see if it is present in our query string.

To analyze the query string when our MainPage is navigated to, you need to override the `OnNavigatedTo()` method from the base class. After calling the base class functionality (we still want the rest of the framework to do its work), we can do our custom processing. In this case, we check the query string if it contains the key we are looking for:

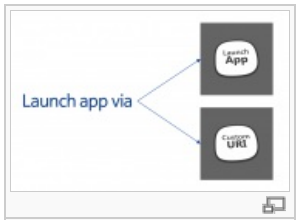
```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);

    // In case our app was launched through the tag, the operating system
    // sends the arguments through the ms_nfp_launchargs parameter
    if (NavigationContext.QueryString.ContainsKey("ms_nfp_launchargs"))
    {
        // Speak the text stored as argument on the NFC tag
        await
_synthesizer.SpeakTextAsync(NavigationContext.QueryString["ms_nfp_launchargs"]);
    }
}
```

Speaking the text is again simple: we use the synthesizer member variable to speak the value (query string) of the parameter. Also speech synthesis is an asynchronous process, so we use the keyword await and do not need to further worry about it.

Finished!

Congratulations, your app is finished! By following this tutorial, you have learned how to create a Windows Phone 8 app and use many of its unique features, including speech recognition, speech synthesis, NFC, including libraries with the NuGet package manager and launching apps through tags.

Bonus: Launching through Custom URI Schemes



If you would like to further extend the app so that it can also be launched through a custom URI scheme and not just the LaunchApp tag, follow the rest of this tutorial. Custom URI schemes have the advantage that they take less writable space on the tag (the app ID of a WP8 app is rather long; if you also add the app ID of your port on Windows 8, there is even more overhead). Additionally, custom URI schemes are easier to use in cross-platform scenarios, as the LaunchApp type is unique to Windows phone and only supported by that platform.

The downside of using a custom URI scheme is that it's not unique to your app - anyone else could implement an app that registers for the same URI scheme. In contrast to that, the LaunchApp tag includes your unique app ID.

Registering for a URI Scheme

The first step is to register for the URI scheme. This process is slightly different when comparing Windows 8 and Windows Phone 8; we're covering the Windows Phone 8 way here. First, close the `WMAppManifest.xml` file, and open it again by right-clicking on the file -> Open With -> XML (Text) Editor. Scroll down to the `</Tokens>` element, and add the following protocol registration:

```
// ...
</Tokens>
<Extensions>
  <Protocol Name="nearspeaktutorial"
            NavUriFragment="encodedLaunchUri=%s"
            TaskID="_default" />
</Extensions>
```

This will register the app for our own URI scheme `nearspeaktutorial`. You can customize this part and use your own protocol name. Note that most of the standard protocols are reserved by the system and can't be used in your app (e.g., `http`). The other parts of the XML protocol definition code are fixed, do not change them.

Mapping the URI

The next part is probably the most complicated, but only needs to be done once. For all your future apps, you can use pretty much the same code.

In the previous example, we took care of parameters that were sent right to our MainPage. For handling the launch via a custom URI scheme, we need to go to a lower level and customize the behavior of the *UriMapper*. The UriMapper is always called when navigating in your app, as well as when the app is launched. Therefore, you can customize the behavior of your app even before the MainPage is loaded - and load for example a different page, depending on the contents of the custom URI scheme.

In our case, we will find out if the app launch URI contains our custom protocol name: if the app has been launched via the custom URI scheme, the argument `{{{1}}}` is added to the URI. In case we can find this parameter, we will extract the text to speak (`Good morning.`) and launch the `MainPage.xaml` with those parameters. Essentially, we're redirecting the app to a custom address. As we're clever, we will send the argument to the page using the `ms_nfp_launchargs` key name, which we're already handling from the LaunchApp tags. This saves us from writing a second handler, which would replicate exactly the same functionality that we have already implemented.

In case the app was not launched through the URI, or we're in the middle of a different navigation within the app, we just return the original and unmodified URL so that the default app framework can do its tasks.

To add the class, right-click the project and choose Add -> New Item... -> Class. Give it the name `NearSpeakUriMapper.cs`. Derive the class from `UriMapperBase` and impement the `MapUri(Uri uri)` method, as mandated by the base class. Next, write the code to analyze the URI and redirect the flow of the app in case it was launched through our custom URI:

```
class NearSpeakUriMapper : UriMapperBase
{
    public override Uri MapUri(Uri uri)
    {
        // Example: "Protocol?encodedLaunchUri=nearspeaktutorial:Good+morning."
        var tempUri = HttpUtility.UrlDecode(uri.ToString());
        var launchContents = Regex.Match(tempUri,
@"nearspeaktutorial:(.*)$").Groups[1].Value;
        if (!String.IsNullOrEmpty(launchContents))
        {
            // Launched from associated "nearspeaktutorial:" protocol
            // Call MainPage.xaml with parameters
            return new Uri("/MainPage.xaml?ms_nfp_launchargs=" + launchContents,
UriKind.Relative);
        }

        // Include the original URI with the mapping to the main page
        return uri;
    }
}
```

Activating the URI Mapping

Now that the URI mapping class is prepared, we need to tell our app's framework to actually make use of the class. To do so, open `App.xaml.cs` and search for the (by default collapsed) *Phone application initialization* region. Inside this region, you will find the `InitializePhoneApplication()` method. Right below the line that creates a new instance of the `RootFrame`, add our custom URI mapper:

```
// Find this method in the "Phone application initialization" region of your App.xaml.cs
private void InitializePhoneApplication()
{
    RootFrame = new PhoneApplicationFrame();
    // Insert this line:
    RootFrame.UriMapper = new NearSpeakUriMapper();
    // ...
}
```

Now, whenever your app is launched through a tag that contains a standardized URL record with the contents

`nearspeaktutorial:Hello+world` (or any other text to speak), the phone launches your app and it speaks the message. To write such a tag, you can either use a tool like Nfc Interactor, or if you'd like to write the tag from within your application, follow for example the instructions in the How to Store Application Data on NFC Tags article.

Finished! (2)

Now after adding the bonus content, your app can be launched via the Microsoft-specific LaunchApp NFC tag, as well as through an own custom URI scheme that you defined. In both cases, the app will immediately speak the voice message you stored on the tag through the text-to-speech framework (speech synthesis) - which you recorded using the speech recognition feature of Windows Phone.

Have fun with the app and your future NFC apps! You can download the full source code of the sample project from this page.

The official NearSpeak app that also includes translation using the Microsoft Translator web service can be downloaded from the Windows Phone store. I hope you enjoyed the behind-the-scenes look on how to create a fully functional and great app for Windows Phone! Please do not use the NearSpeak name, graphics or URI scheme in your own app, as the app is already in the store and publicly available.

--ajakl 16:30, 16 December 2012 (EET)

Note: This is a community entry in the Windows Phone 8 Wiki Competition 2012Q4.