

**Windows Phone 7.5**

**Fundamentals**

# Windows Phone 7.5

# Fundamentals

**Author: Andrea Boschin**

**January 2012**

**Published by SilverlightShow**

**©2012 SilverlightShow**

Table of Contents

[Chapter 1: Local SQL Database 5](#_Toc314496263)

[How does it work. 5](#_Toc314496264)

[Mapping entities to tables 6](#_Toc314496265)

[Creating the database 8](#_Toc314496266)

[Querying the data 9](#_Toc314496267)

[Apply indexes 11](#_Toc314496268)

[When to use SQL CE 11](#_Toc314496269)

[Chapter 2: Accessing your phone 12](#_Toc314496270)

[Accessing phone information with DeviceStatus 12](#_Toc314496271)

[New Launchers and Choosers 13](#_Toc314496272)

[Accessing Contacts and Appointments 15](#_Toc314496273)

[Considerations 17](#_Toc314496274)

[Chapter 3: Programmatically use the phone camera 18](#_Toc314496275)

[Connecting to the camera(s) 18](#_Toc314496276)

[Handling the camera orientation 19](#_Toc314496277)

[Tune the camera settings 20](#_Toc314496278)

[And finally capture a shot... 22](#_Toc314496279)

[It's only the first step 23](#_Toc314496280)

[Chapter 4: Manipulating camera stream 24](#_Toc314496281)

[Accessing the raw stream 24](#_Toc314496282)

[Saving the stream to a file 26](#_Toc314496283)

[Updated example 27](#_Toc314496284)

[Chapter 5: Play with music 28](#_Toc314496285)

[The role of Agents 28](#_Toc314496286)

[Manage the playlist 29](#_Toc314496287)

[Starting the task 31](#_Toc314496288)

[When to use Audio Agents 32](#_Toc314496289)

[Chapter 6: Use Background agents 33](#_Toc314496290)

[Understanding Agents 33](#_Toc314496291)

[Create an agent 34](#_Toc314496292)

[A funny example 35](#_Toc314496293)

[A beautiful opportunity 36](#_Toc314496294)

[Chapter 7: Using advanced tiles API 37](#_Toc314496295)

[Accessing and changing your tile(s) 37](#_Toc314496296)

[Schedule tile updates 39](#_Toc314496297)

[Using secondary tiles 39](#_Toc314496298)

[Tile your apps. 41](#_Toc314496299)

[Chapter 8: Using Sockets 42](#_Toc314496300)

[What sockets can do (and what they can't) 42](#_Toc314496301)

[Implementing a simple UDP protocol (NTP) 43](#_Toc314496302)

[Using the NtpClient 45](#_Toc314496303)

[About the Author 47](#_Toc314496304)

# **Chapter 1: Local SQL Database**

Once you start programming Windows Phone just after the first experiments, when you begin writing something of real, immediately you need something to store information locally.

Also if the phone is a mobile device and it is almost always connected, there is information you cannot store remotely but you need to have always available. I'm speaking for sure about configuration settings but also about locally cached relational data that is required to run the application fast, without continuously access the network.

In Windows Phone 7.0 the sole storage available was the Isolated Storage but it appear like a filesystem and it requires lot of code to serialize and deserialize entities. Some open source solutions have been created to work around this problem. I for sure remember Sterling, a real object oriented no-sql database that I used often to write my own applications. In the upcoming release of Windows Phone 7.1, Microsoft decided to add a Local Database implementation that is targeted to business applications and is created on top of SQL Server CE. This new feature completes the storage solutions available for the phone, supporting scenario where you need a real relational database and is really easy to configure and use. It does not totally override other solutions. Sterling remains a good solution for many scenarios, but having a real relational store is almost wonderful in many cases.

[**Download the source code**](Source_code/SLPG.Mango.zip)

## How does it work.

As I've said, the new Local Database available in Windows Phone 7.1, is based on the SQL Server Compact Edition, a well known and popular edition of SQL Server, originally created to support Windows Mobile and Windows CE with a local relational store. Due to the fact that under the hoods of the Windows Phone 7.x there is Microsoft .NET Compact Framework 3.7, SQL CE is a obvious choice and it is for sure a solid and reliable tool to support real world applications.

But, as you know, the Silverlight API in Windows Phone 7 does not include ADO.NET so the access to a SQL CE database cannot be done using normal SQL queries. For this purpose the WP7's team taken the existing LinqToSQL code and ported it to the phone. This means that to access the SQL CE storage you have to use linq queries and the DataContext to modify and extract information from the database. LinqToSQL is not a real object oriented database. It use a simple attribute-based mapping that does not abstract so much the relational structure to a real consistent domain model and this is the reason why Microsoft decided to override it with the Entity Framework, but for the purpose of creating and accessing a local database it is a good, easy and realiable choice.

A SQL CE Database is usually associated with SDF files. In Windows Phone SDF still exists but they are placed into the isolated storage so the first thing you notice is a special connection string format that follows this pattern:

isostore://file.sdf

As a side consideration, having the sdf in isolated storage implies that the access to the file is strictly related to the application that creates it. With this release there is not any way of sharing a database between multiple applications without duplicating it across different isolated storage directories.

From the developer point of view, working with a SQL CE database means adding a reference to an assembly (System.Data.Linq.dll). Once the reference is added there is not any requirement of deploying SQL CE runtime, because it is part of the Windows Phone 7.1 so it does not increase the size of your application.

## Mapping entities to tables

In LinqToSQL there is a direct association between tables and entities so you will always have an entity mapped to each table, also in many-to-many scenarios where the relation table is represented by a real entity instead of two crossing collection as you expect. So in a Product/Order domain you will have also a ProductsOrders entity representing the many-to-many relation between the Product and Order entities.

The mapping of entities then, is really simplified because you have to create a class for each table, and always one-to-many associations. For this purpose you have a small set of attributes you can apply to the classes. TableAttribute to describe the table, ColumnAttribute for the fields and AssociationAttribute for relationships. Your entities are by default POCO object so you can add your own inheritance for the purposes of you application. As an example you can implement INotifyPropertyChanged to support databinding but you can also derive from base classes for same specific scenario. The following snippet shows an entity related to a feed with a bunch of properties mapped:

1: [Table(Name = "Feeds")]

2: public abstract class Feed

3: {

4: [Column(IsPrimaryKey = true)]

5: public int ID { get; set; }

6: [Column]

7: public string Title{ get; set; }

8: [Column]

9: public string Description{ get; set; }

10: [Column]

11: public string Address{ get; set; }

12: [Column]

13: public DateTime DateCreated { get; set; }

14: }

Since the TableAttribute simply names the table on the database, the ColumnAttribute is able to determine the behavior of the field. It has a number of properties defined. In this snippet you see I've defined the primary key using the IsPrimaryKey property.

To map associations you have to work on both sided of the relationship. This means adding a collection property on the "many" side and a reference property ong the "one" side. Here is an example of the relation between Feed and Post where Feed has many Posts:

1: // on the Feed side //////////////////////////////////////////////////////////////

2:

3: private readonly EntitySet<Post> postsRef = new EntitySet<Post>();

4:

5: [Association(Name = "FK\_Feed\_Posts", Storage = "postsRef", ThisKey = "ID", OtherKey = "FeedID")]

6: public EntitySet<Post> Posts

7: {

8: get { return this.postsRef; }

9: }

10:

11: // on the Post side //////////////////////////////////////////////////////////////

12:

13: private EntityRef<Feed> feedRef = new EntityRef<Feed>();

14:

15: [Association(Name = "FK\_Feed\_Posts", Storage = "feedRef", ThisKey = "FeedID", OtherKey = "ID", IsForeignKey = true)]

16: public Feed Feed

17: {

18: get { return this.feedRef.Entity; }

19: set

20: {

21: Feed previousValue = this.feedRef.Entity;

22:

23: if (previousValue != value || this.feedRef.HasLoadedOrAssignedValue == false)

24: {

25: if (previousValue != null)

26: {

27: this.feedRef.Entity = null;

28: previousValue.Posts.Remove(this);

29: }

30:

31: this.feedRef.Entity = value;

32:

33: if (value != null)

34: {

35: value.Posts.Add(this);

36: this.FeedID = value.ID;

37: }

38: else

39: {

40: this.FeedID = default(int);

41: }

42: }

43: }

44: }

Especially the Post side, it requires a number of lines of code, but since the code is always the same I suspect you can work to create a Visual Studio snippet to shorten and automate its creation.

The mapping is for sure simple and probably it sounds trivial to a refined palate used to the complex mapping of modern ORMs. However it reveals some surprises if you go deep analyzing the model. As LinqToSQL it supports a raw Inheritance "table-per-hierarchy" mapping. For people that are not used to ORMs lingo, "table-per-hierarchy" is the simpler model for mapping inheritance of entities. It involves having a single table mapping all the inherited entities so the fields that are not in common between inherited entities have to be nullable. Here is how to map inheritance:

1: [Table(Name = "Feeds")]

2: [InheritanceMapping(Code = "ATOM", Type = typeof(AtomFeed))]

3: [InheritanceMapping(Code = "RSS", Type = typeof(RssFeed), IsDefault = true)]

4: public abstract class Feed

5: {

6: [Column(IsDiscriminator = true)]

7: public string Type{ get; set; }

8:

9: // other properties here (these are common to other types

10: }

11:

12: public class AtomFeed : Feed

13: {

14: [Column(CanBeNull = true)]

15: public string Email { get; set; }

16: }

17:

18: public class RssFeed : Feed

19: {

20: [Column(CanBeNull = true)]

21: public string Generator { get; set; }

22: }

As you can see AtomFeed and RssFeed share all the properties of the base class but add other properties that are specific to each type. These are decorated with "CanBeNull" property because the row on the table will always contain all the properties of all the inherited entities. It is not the optimal solution but it is better than nothing.

## Creating the database

Also if at the basis of SQL CE there is SDF files, you should never directly access them but you have to manage you database directly from code. If you search the internet you will find some examples to download the SDF from the phone and the upload it again, but in this chapter I will cover the official way. To me the creation of the structure of the database is very straightforward using code but I think someone can take advantage of the direct access to the SDF providing pre-populated databases.

The last thing you have to do before creating the physical database, just after creating entities, is a special class called DataContext. The DataContext wraps all the operations you can do with a database like inserting, updating and deleting data and exposes a number of sets, one for each "table" you have in the SDF. The DataContext appear like this:

1: public class FeedReaderDataContext : DataContext

2: {

3: public const string ConnectionString = "isostore:/feeds.sdf";

4:

5: public FeedReaderDataContext(string connectionString)

6: : base(connectionString)

7: {

8: this.Feeds = this.GetTable<Feed>();

9: this.Posts = this.GetTable<Post>();

10: }

11:

12: public Table<Feed> Feeds { get; set; }

13: public Table<Post> Posts { get; set; }

14: }

In this example I show a datacontext made to store blog Feeds and Posts so in the first line I have a constant string with the connection string "isostore://feeds.sdf". Then I have two properties mapping the sets. A set is a collection created with the Table<T> class and gives a name to the "table" in the database. In the constructor I call the GetTable<T>() method that connects the DataContext sets with the SDF tables.

After the DataContext is in place you have to write code to create the database if it is not already in the isolated storage. For this purpose you have the CreateDatabase() method and the DatabaseExists() method. The pattern I use for the data access is creating a repository class that wraps the DataContext with my custom data access methods. In the repository I've a static method "Initialize" that is in charge of checking the existing database and eventually create it.

1: public class FeedReaderRepository

2: {

3: public static void Initialize()

4: {

5: using (FeedReaderDataContext dc = new FeedReaderDataContext(FeedReaderDataContext.ConnectionString))

6: {

7: if (dc.DatabaseExists() == false)

8: {

9: dc.CreateDatabase();

10:

11: // add here extra initialization like creating required rows

12:

13: dc.SubmitChanges();

14: }

15: }

16: }

17: }

In place of the comment I'm use to add rows that I expect already exist into the new database, like lookup records and initialized data. The operation of checking the database must be made at the beginning of the life of the application just before it starts accessing rows. So the better place is the App class constructor where I add the following line:

FeedReaderRepository.Initialize();

When you deploy an application it is expected you make some changes to the database during the application lifecycle. It is almost impossible that the first database you create on the phone is able to support all the future scenarios. For this purpose Windows Phone 7.1 provides a DatabaseSchemaUpdater class. It lets you access a version number for the deployed schema and a bunch of methods that supports the creation of new tables, columns, indexes and relations. As far I can see there is nothing that let you delete something in existing databases.

## Querying the data

Now that your mapping is created and the database has been deployed to the device it is time to start querying the data. The pattern is you create the DataContext instance, pointing to the right database, then use the sets to make queries and the Insert and Delete methods to make changes. Using linq is really simple and well documented so in this paragraph I will only show the basic concepts. To deal with joins, grouping and other advanced operations please refer to the LinqToSQL documentation:

1: public IEnumerable<Feed> GetRecentFeeds()

2: {

3: using (FeedReaderDataContext dc = new FeedReaderDataContext(FeedReaderDataContext.ConnectionString))

4: return dc.Feeds

5: .OrderByDescending(f => f.DateCreated)

6: .Take(2)

7: .ToArray();

8: }

9:

10: public bool FeedExists(string address)

11: {

12: using (FeedReaderDataContext dc = new FeedReaderDataContext(FeedReaderDataContext.ConnectionString))

13: {

14: return (from fd in dc.Feeds

15: where fd.Address == address

16: select fd).Count() > 0;

17: }

18: }

In the above sample I've put togheter two queries. The first select the most recent feeds taking the first two from a list orderer descending by the column DateCreated. The important thing to remember is to always call ToArray() or ToList() before returning the entities because it performs the query. Returning the reqult of the Take(2) does not returns the entities but a query that cannot be executed outside of the scope of the DataContext. So enumerating the query will raise an exception if you do not call the ToArray method here. The other query search for a field by the address string and verify it exists. In this case the Count() method perform the query and the return value will be a scalar that is not connected with the Datacontext.

To update entities you have to make changed to objects you have extracted from the DataContext. When you make the changes they are tracked and calling the SubmitChanges() method will persist all the changes in a unique batch. This also applies to Insert and Delete that map to the InsertOnSubmit and DeleteOnSubmit methods. This way you can watch at the DataContext as an implementation of the Unit-Of-Work pattern. You make changed then, at the end, you call SubmitChanges to persist to the database. Here is some examples:

1: public void DeleteFeed(Feed feed)

2: {

3: using (FeedReaderDataContext dc = new FeedReaderDataContext(FeedReaderDataContext.ConnectionString))

4: {

5: feed.DateCreated = DateTime.Now;

6: dc.Feeds.DeleteOnSubmit(feed);

7: dc.SubmitChanges();

8: }

9: }

10:

11: public void SaveFeed(Feed feed)

12: {

13: using (FeedReaderDataContext dc = new FeedReaderDataContext(FeedReaderDataContext.ConnectionString))

14: {

15: feed.DateCreated = DateTime.Now;

16: dc.Feeds.InsertOnSubmit(feed);

17: dc.SubmitChanges();

18: }

19: }

20:

21: public void UpdateFeedDate(int id, DateTime date)

22: {

23: using (FeedReaderDataContext dc = new FeedReaderDataContext(FeedReaderDataContext.ConnectionString))

24: {

25: var feed = (from f in dc.Feeds

26: where f.ID == id

27: select f).FirstOrDefault();

28:

29: if (feed != null)

30: {

31: feed.DateLastUpdate = date;

32: dc.SubmitChanges();

33: }

34: }

35: }

This is the beautiful part of the game. Linq is the most flexible tool for querying the data and it is plenty of examples on the Internet if you need help to extract the data in the way you prefer. Refer to this [page](http://msdn.microsoft.com/en-us/vcsharp/aa336746) for a number of samples.

## Apply indexes

Everyone have used a database knows the importance of indexes. A local database on SQL CE probably will never contain million of records, nevetheless sometimes an index can make the difference when you need to search with a where  clause or when you need to sort a set using order by. In Windows Phone 7.1 you can easily apply indexes to tables and the underlying SQL CE engine will manage them transparently, without the need of wiring any code to handle them.

To apply an index you have simply to use the IndexAttribute, at the entity level, specifying the columns to which it applies, its name and the uniqueness constraint.

1: [Table(Name = "Feeds")]

2: [Index(Columns = "DateCreated", IsUnique = false, Name = "Feed\_DateCreated")]

3: public abstract class Feed : ObservableObject

4: {

5: /// ...

6: }

Index are created on the database when you call the CreateDatabase() method then are completely managed by SQL CE. If you need an additional index you can call the AddIndex method on the DatabaseSchemaUpdater class.

## When to use SQL CE

SQL CE is a simple database and for sure it will power up applications developed for Windows Phone 7.1. You may consider using it both when you need to store something a bit complex than a configuration, but also in scenarios where you need to cache data for a smart application to continue working while it is offline. In both the cases it is a good solution able to speed up you development and thanks to the updating features you are able to maintain your application, adding features during the application lifecyle, in a reliable way.

# Chapter 2: Accessing your phone

Time ago I wrote an article about the programmatic interaction with the phone services, with the title "[Taking advantage of the phone](http://www.silverlightshow.net/items/Windows-Phone-7-Part-6-Taking-advantage-of-the-phone.aspx)".

In that article I explored a bunch of features that enabled you to interact with some parts of the operating system with Launchers & Choosers, and getting informations about the device and the user. With the new release of the Windows Phone 7.5, these features has been expanded, with the addition of lot of new "handles" you can use to programmatically access informations and change them. In this chapter, I would like to give you a brief resume of these improvements that grant you new opportunities to better integrate your applications with the phone environment.

[**Download the source code**](Source_code/SLPG.Mango2.zip)

## Accessing phone information with DeviceStatus

A number of the information about the device hardware and status are now exposed throught a new static class called DeviceStatus. Differently from the old DeviceExtendedProperties this class exposes some strongly types properties that simplify the code and add interesting data. So, as and example, to access the DeviceName you can now write a single line: Here is the comparison between OS 7.0 and OS 7.5

1: // OS 7.0 --------------------------------

2:

3: object value;

4:

5: bool success = DeviceExtendedProperties.TryGetValue("DeviceName", out value);

6:

7: if (success)

8: MessageBox.Show(value.ToString());

9: else

10: MessageBox.Show("Unable to retrieve value");

11:

12: // OS 7.5 --------------------------------

13:

14: MessageBox.Show(DeviceStatus.DeviceName);

15:

Side by side with this useful but not life-changing feature, the new OS 7.5 takes some new properties and events. These properties allow you to react to the presence of an external keyboard, to its deploment and to the change of the power source. Knowing about the keybord may be the key to decide about the presentation of your user interface, and knowing about the power source can impact the behavior of an application that, as an example, is running under the locked screen. Detecting these changes is really simple as it is attaching an event, but you have to know that the event is not raised on the UI thread so you have to use the dispatcher to update the user interface.

1: public PhoneInfo()

2: {

3: InitializeComponent();

4:

5: DeviceStatus.KeyboardDeployedChanged += new EventHandler(DeviceStatus\_KeyboardDeployedChanged);

6: DeviceStatus.PowerSourceChanged += new EventHandler(DeviceStatus\_PowerSourceChanged);

7: }

8:

9: public void DeviceStatus\_PowerSourceChanged(object sender, EventArgs e)

10: {

11: this.Data.Dispatcher.BeginInvoke(this.Load);

12: }

13:

14: public void DeviceStatus\_KeyboardDeployedChanged(object sender, EventArgs e)

15: {

16: this.Data.Dispatcher.BeginInvoke(this.Load);

17: }

18:

19: public void Load()

20: {

21: // update here te user interface

22: }

## New Launchers and Choosers

As you remember for sure Launchers & Choosers are useful tools that let you interact with various services provided by the phone. With Launchers you can start something that does not return anything to your application. A simple example is given by the EmailComposeTask. It allows to start writing an email with the configured client and you do not have to expect any result from it. On the other side Choosers, like the name suggest, are useful to choose something into your phone. EmailAddressChooserTask lets you scan the address book searching for email address and obviously it returns the found address.

With OS 7.5 there are a small set of Launchers and Chooser added but they give some useful tasks. Here is the updated table containing the new tasks in red:

|  |  |
| --- | --- |
| **Launchers** | **Choosers** |
| * [EmailComposeTask](http://msdn.microsoft.com/en-us/library/#BKMK_Email) * [MarketplaceDetailTask](http://msdn.microsoft.com/en-us/library/#BKMK_MPDetail) * [MarketplaceHubTask](http://msdn.microsoft.com/en-us/library/#BKMK_MPHub) * [MarketplaceReviewTask](http://msdn.microsoft.com/en-us/library/#BKMK_MPReview) * [MarketplaceSearchTask](http://msdn.microsoft.com/en-us/library/#BKMK_MPSearch) * [MediaPlayerLauncher](http://msdn.microsoft.com/en-us/library/#BKMK_MediaPlayer) * [PhoneCallTask](http://msdn.microsoft.com/en-us/library/#BKMK_Phone) * [SearchTask](http://msdn.microsoft.com/en-us/library/#BKMK_Search) * [SmsComposeTask](http://msdn.microsoft.com/en-us/library/#BKMK_SMS) * [WebBrowserTask](http://msdn.microsoft.com/en-us/library/#BKMK_Web) * [BingMapsTask](http://msdn.microsoft.com/en-us/library/microsoft.phone.tasks.bingmapstask(v=vs.92).aspx) * [BingMapsDirectionsTasks](http://msdn.microsoft.com/en-us/library/microsoft.phone.tasks.bingmapsdirectionstask(v=vs.92).aspx) | * [CameraCaptureTask](http://msdn.microsoft.com/en-us/library/#BKMK_Camera) * [EmailAddressChooserTask](http://msdn.microsoft.com/en-us/library/#BKMK_Email) * [PhoneNumberChooserTask](http://msdn.microsoft.com/en-us/library/#BKMK_Phone) * [PhotoChooserTask](http://msdn.microsoft.com/en-us/library/#BKMK_Photo) * [SaveEmailAddressTask](http://msdn.microsoft.com/en-us/library/#BKMK_SaveEmail) * [SavePhoneNumberTask](http://msdn.microsoft.com/en-us/library/#BKMK_SavePhone) * [AddressChooserTask](http://msdn.microsoft.com/en-us/library/microsoft.phone.tasks.addresschoosertask(v=vs.92).aspx) * [GameInviteTask](http://msdn.microsoft.com/en-us/library/microsoft.phone.tasks.gameinvitetask(v=vs.92).aspx) * [SaveRingtoneTask](http://msdn.microsoft.com/en-us/library/microsoft.phone.tasks.saveringtonetask(v=vs.92).aspx) * [SaveContactTask](http://msdn.microsoft.com/en-us/library/microsoft.phone.tasks.savecontacttask(v=vs.92).aspx) |

Under the Launchers category we see the new Bing Maps services. They let you open the mapping application at a given position and getting directions betweek two points. The last one starts the mapping application loading a route between the provided coordinates and initiate navigation. When you are in a location where this service is enabled you will ear the voice suggesting directions while driving.

On the side of the Choosers you can finally add a contact to the address book using the SaveContactTask, search for an address in the same way you can search for a number with AddressChooserTask, start a game invitation with the GameInviteTask and save a ringtone to the media library. The SaveRingToneTask require you specify the file to add respecting the following constraints:

* Ringtone files must be of type MP3 or WMA.
* Ringtone files must be less than 40 seconds in length.
* Ringtone files must not have digital rights management (DRM) protection.
* Ringtone files must be less than 1 MB in size.

After you add the ringtone it is available in the Settings page where you can choose a tone for the phone. Please be aware that the ringtone is only available for incoming calls and not for all the other notifications like SMS, Email and so on.

Some of these choosers may seem to be launchers because you don't expect any result from them (e.g. the SaveRingtoneTask) but they are really choosers because they return the result of the operation with a possible exception occured.

In the following sample I show how to use the AddressChooserTask and BingMapsDirectionsTasks to make a route between two of your contacs.

1: public partial class UsingLaunchers : PhoneApplicationPage

2: {

3: public AddressChooserTask ChooseStart { get; set; }

4: private string StartAddress { get; set; }

5: private string StartLabel { get; set; }

6: public AddressChooserTask ChooseEnd { get; set; }

7: private string EndAddress { get; set; }

8: private string EndLabel { get; set; }

9:

10: public UsingLaunchers()

11: {

12: InitializeComponent();

13:

14: this.ChooseStart = new AddressChooserTask();

15: this.ChooseStart.Completed += new EventHandler<AddressResult>(chooseStart\_Completed);

16:

17: this.ChooseEnd = new AddressChooserTask();

18: this.ChooseEnd.Completed += new EventHandler<AddressResult>(chooseEnd\_Completed);

19: }

20:

21: void chooseStart\_Completed(object sender, AddressResult e)

22: {

23: if (e.Error == null)

24: {

25: this.txtStart.Text =

26: this.StartAddress = e.Address;

27: this.StartLabel = e.DisplayName;

28: }

29: }

30:

31: void chooseEnd\_Completed(object sender, AddressResult e)

32: {

33: if (e.Error == null)

34: {

35: this.txtEnd.Text =

36: this.EndAddress = e.Address;

37: this.EndLabel = e.DisplayName;

38: }

39: }

40:

41: private void bSearchDirections\_Click(object sender, RoutedEventArgs e)

42: {

43: if (!string.IsNullOrEmpty(this.StartAddress) &&

44: !string.IsNullOrEmpty(this.EndLabel))

45: {

46: GeoCoordinate start =

47: this.GeoCodeAddress(this.StartAddress);

48: GeoCoordinate end =

49: this.GeoCodeAddress(this.EndAddress);

50:

51: BingMapsDirectionsTask task = new BingMapsDirectionsTask

52: {

53: Start = new LabeledMapLocation

54: {

55: Label = this.StartLabel,

56: Location = start

57: },

58: End = new LabeledMapLocation

59: {

60: Label = this.EndLabel,

61: Location = end

62: }

63: };

64:

65: task.Show();

66: }

67: }

68:

69: private GeoCoordinate GeoCodeAddress(string address)

70: {

71: // TODO: implement here a GeoCoding service to get coordinates from address

72: return new GeoCoordinate();

73: }

74:

75: private void bChooseStart\_Click(object sender, RoutedEventArgs e)

76: {

77: this.ChooseStart.Show();

78: }

79:

80: private void bChooseEnd\_Click(object sender, RoutedEventArgs e)

81: {

82: this.ChooseEnd.Show();

83: }

84: }

Please pay attention I have not implemented a GeoCoding of the addresses retrieved from the AddressChooserTask. You have to do this for example using the Bing Maps Geocoding Service but it requires you subscribe to the service for a developer key before to place a call to the online services.

## Accessing Contacts and Appointments

Using a Chooser to search for an Address is for sure a good thing but sometimes it may happen you want to directly integrate the contacts into your applications. With the new OS 7.5 it is now possible to enumerate the contacts and directly access their information. The Contacts class provides a way to start searches into contacts and then displays results with a detailed object model that exposes almost every available information. These searches span over different accounts you added to your phone and, infact, you have an Accounts property that lets you know what the accounts you are searching over are. Unfortunately there is not any way of searching over one of the catalogs but the SearchAsync method always start the search over all the accounts.

To make a search you have to create an instance of the Contacts class then use the StartAsync method providing the search key and the field you want to use for your search. The search key changes its meaning depending on the field you are searching. If you search the DisplayName it is used as a LIKE criteria but if you search for an EmailAddress the part before the @ is searched by start and the other part performs a smart matching trying various combinations. The following box shows how to search contacts by DisplayName

1: private void Button\_Click(object sender, RoutedEventArgs e)

2: {

3: string searchKey = this.SearchKey.Text;

4:

5: Contacts contacts = new Contacts();

6: contacts.SearchCompleted += new EventHandler<ContactsSearchEventArgs>(contacts\_SearchCompleted);

7: contacts.SearchAsync(searchKey, FilterKind.DisplayName, null);

8: }

9:

10: private void contacts\_SearchCompleted(object sender, ContactsSearchEventArgs e)

11: {

12: this.MyContacts.ItemsSource = e.Results;

13: }

After you got the contacts you can access the information stored in the result set and perform linq queries with LinqToObjects. The set is really complete and includes lot of collections with PhoneNumbers, EmailAddresses, and so on. In the following box I display the PhoneNumbers collection when the item is selected in the ListBox

1: private void MyContacts\_SelectionChanged(object sender, SelectionChangedEventArgs e)

2: {

3: Contact contact = this.MyContacts.SelectedValue as Contact;

4:

5: if (contact != null)

6: this.MyContactsNumbers.ItemsSource = contact.PhoneNumbers;

7: else

8: this.MyContactsNumbers.ItemsSource = Enumerable.Empty<Contact>();

9: }

In this case each phone number has a Kind property that let you know if the number is registered as a Mobile, Work, etc... From the Contacts result set you are also able to access the picture if it is available. Using the GetPicture method you get a stream to the image or "null" of the picture is not available. Here is the previous method changed to also diplay the contact picture:

1: private void MyContacts\_SelectionChanged(object sender, SelectionChangedEventArgs e)

2: {

3: Contact contact = this.MyContacts.SelectedValue as Contact;

4:

5: if (contact != null)

6: {

7: Stream picture = contact.GetPicture();

8:

9: if (picture != null)

10: {

11: BitmapImage image = new BitmapImage();

12: image.SetSource(picture);

13: this.MyContactsImage.Source = image;

14: }

15: else

16: this.MyContactsImage.Source = null;

17:

18: this.MyContactsNumbers.ItemsSource = contact.PhoneNumbers;

19: }

20: else

21: {

22: this.MyContactsImage.Source = null;

23: this.MyContactsNumbers.ItemsSource = Enumerable.Empty<Contact>();

24: }

25: }

The same way you can access contacts you can also access appointments. Using the Appointments class you can scan the calendar of the phone to integrate your application with it. The way you perform searches is very similar to contacts but you have to provide a timeframe. In the following snippet I make a search for the current week.

1: void AppointmentsExplorer\_Loaded(object sender, RoutedEventArgs e)

2: {

3: DateTime start = DateTime.Today.Subtract(

4: TimeSpan.FromDays((int)DateTime.Today.DayOfWeek));

5:

6: Appointments appointment = new Appointments();

7: appointment.SearchCompleted += new EventHandler<AppointmentsSearchEventArgs>(appointment\_SearchCompleted);

8: appointment.SearchAsync(start, start.AddDays(7), 100, null);

9: }

10:

11: void appointment\_SearchCompleted(object sender, AppointmentsSearchEventArgs e)

12: {

13: this.MyAppointments.ItemsSource = e.Results;

14: }

The SearchAsync method has various overloads and in this case you are able to make your search on a specific account providing an instance from the Accounts property.

## Considerations

The changes I've explained in this chapter are someway good because they simplify some tasks and add interesting features. The sole problem to me is the read-only access to calendar and appointments. In my opinion the read-only access to the store is good for contacts by I preferred to have a read-write access to the calendar to be able to add appointments from an application. I'm able to figure out a wide set of cases where this would be a beautiful feature.

# Chapter 3: Programmatically use the phone camera

Since the first release, Windows Phone devices have got at least a camera on board but it remained only a beautiful gadget, useful only to take shots of your sons or during your favorite rockstar's concert, because in the OS7.0 there were few api to access the camera.

In the first release of the operating system there was only a CameraCaptureTask able to start the camera and wait for the user to take a picture, then return it to the application leaving to the user the choose of the resolution, of the focus and other things. It is for sure a beautiful feature but the best is being able to integrate the camera stream into your own application and manipulate it, controlling all the features to get the most useful images from it for the sake of the application.

With OS7.5 it is now possible to deep integrate the camera using a set of API that let you fine tune the image you get from the shot, and decide when to take a shot without asking the user to press the hardware button. There are two levels of integration with the camera. You can simply command the camera and take pictures or you can also directly access the stream and manipulate it for the most complicated elaborations. In this chapter, I will cover the simplest case, showing how to replicate the most common camera features into your own application.

[**Download the source code**](Source_code/SLPG.MangoCamera.zip)

## Connecting to the camera(s)

As you know for sure, Windows Phone must have at least a camera, positioned on the back, that is usually referred as the "main camera" because it have the most advanced features like higher resolution and so on. But, there are lots of models that benefit of a secondary camera that is positioned on top of the screen and used when you have to place a video call. This camera is called "front facing" and it is not required by the minimal requirements.

So, when you need to access the phone camera, the first thing you have to do is checking if the camera you want to use is available. Using the PhotoCamera class this check is quite simple. You have an IsCameraTypeSupported method you can use to test using a CameraType enumerator as input. So your code can choose if it need to use FrontFacing or Primary in this way:

1: private PhotoCamera Camera { get; set; }

2:

3: private void MainPage\_Loaded(object sender, RoutedEventArgs e)

4: {

5: if (PhotoCamera.IsCameraTypeSupported(CameraType.FrontFacing))

6: this.Camera = new PhotoCamera(CameraType.FrontFacing);

7: else if (PhotoCamera.IsCameraTypeSupported(CameraType.Primary))

8: this.Camera = new PhotoCamera(CameraType.Primary);

9: else

10: {

11: MessageBox.Show("Cannot find a camera on this device");

12: return;

13: }

14:

15: // TODO: initialize here the camera

16: }

The PhotoCamera class is the starting point you use to access the camera and it represents the type you choosed into the application for the entire lifetime. Once you attached a camera you can attach an "Initialized" event that is useful to know when the information of the camera is available to the application. As I will show in a few it is required to access some important info about the capabilities of the camera.

Now, after connecting to the camera, the first ask is for sure how to present the incoming stream to the user interface. It is probably as simple as you cannot expect. The trick is made using the VideoBrush. If you do not know what it is, you have to experiment it immediately. As the name suggest the VideoBrush is a Brush like the one you use to specify a color or a gradient for a shape. So, with a VideoBrush you can choose to paint almost everything using a video as source. This means you can paint a rectangle background, a border, a circle stroke or a text foreground with the live output of your camera. Ok, let me show how simple it is. First of all decide the shape you want to use - in my case it is a Border - and set the VideoBrush as Background:

1: <Border x:Name="cameraView" BorderBrush="Black" BorderThickness="5" Grid.Column="0">

2: <Border.Background>

3: <VideoBrush x:Name="cameraViewBrush" />

4: </Border.Background>

5: </Border>

Then, go to your codebehind and just after the hook up of the wanted camera connect the PhotoCamera instance to the output brush. It is a single line of code:

this.cameraViewBrush.SetSource(this.Camera);

This simply code automatically makes your rectangle showing the stream that is coming from the camera. If you try to run the application you will see it painted on the background of the border. If you want to enjoy try to change the output shape. I really cannot figure out how can you use a text painted with your camera but it is amazing to see and if you think how simple it is probably is becomes amazing the double.

## Handling the camera orientation

So, please stop now the experimenting with the strangest shapes and restore the simple rectangle, then run the application and try to rotate the camera changing the orientation of the screen. What you will observe is almost so strange like a keyhole painted with the camera stream but mostly disturbing. In few words, when the orientation changes, the video orientation does not change in the same way.

The problem here is that the phone does not directly handle the orientation of the video stream so for example, if you take the camera in LandscapeLeft the video is presented in the correct orientation but if you rotate to LandscapeRight the video is upside down. The PhotoCamera class provides an Orientation property that helps setting the correct degrees to a CompositeTransform So you have to change the xaml for your video brush this way:

1: <VideoBrush x:Name="cameraViewBrush">

2: <VideoBrush.RelativeTransform>

3: <CompositeTransform x:Name="cameraViewBrushTransform" CenterX=".5" CenterY=".5" />

4: </VideoBrush.RelativeTransform>

5: </VideoBrush>

After this you have to attach the OrientationChanged event and apply a correction to the brush to make the video appear always in the right orientation. In the following box I show a CorrectViewFinderOrientation method that applies to Landscape orientations but it is for sure simple to extend to Portrait:

1: private void MainPage\_Loaded(object sender, RoutedEventArgs e)

2: {

3: // TODO: camera initialization goes here

4:

5: this.CorrectViewFinderOrientation(this.Orientation);

6: this.OrientationChanged += new EventHandler<OrientationChangedEventArgs>(MainPage\_OrientationChanged);

7: }

8:

9: private void MainPage\_OrientationChanged(object sender, OrientationChangedEventArgs e)

10: {

11: this.CorrectViewFinderOrientation(e.Orientation);

12: }

13:

14: private void CorrectViewFinderOrientation(PageOrientation orientation)

15: {

16: if (orientation == PageOrientation.LandscapeLeft)

17: this.cameraViewBrushTransform.Rotation = this.Camera.Orientation - 90.0;

18: else if (orientation == PageOrientation.LandscapeRight)

19: this.cameraViewBrushTransform.Rotation = this.Camera.Orientation + 90.0;

20: }

## Tune the camera settings

When you run the standard camera application there are a number of properties you can tune just before taking your shots. Depending of the features of the camera you can choose to use a set of resolutions, and you can also choose to set the flash light in defferent ways. When you have a PhotoCamera instance, just after the Initialization event has been raised, you can benefit of some information that give you a view of the features of the camera you choosed.

The PhotoCamera class exposes the AvailableSizes collection and an IsFlashModeSupported method to test the availability of a given flash mode. On my HTC HD7 the AvailableSizes collections states the following: 320x240, 640x480, 800x600, 1024x768, 1280x960, 1600x1200, 2048x1536 and 2592x1944 with the last resolution of about 4,8 MegaPixels.

On the side of the flash the FlashMode enumerator gives four modes available:

* **On:** means the flashlight always illuminate the scene
* **Off:**means that the flaslight is switched off
* **Auto:**the camera automatically choose when to switch on the flashlight
* **RedEyeReduction:**The camera does a preview flash to reduce the red-eye effect that is mostly evident when you take a picture of a child

On my phone the test for **FlashMode.RedEyeReduction**returns false stating this device does not have this feature but it returns true in the other cases.

Once you have decided the resolution and the flash mode you need you can set the relative property to change it on the phone camera. In this box I reported a snippet that does the test for both the collections and then changes resolution and flash mode according to the press of a button in the user interface:

1: private void Camera\_Initialized(object sender, CameraOperationCompletedEventArgs e)

2: {

3: this.AvailableFlashModes = new Queue<FlashMode>();

4:

5: if (this.Camera.IsFlashModeSupported(FlashMode.Auto))

6: this.AvailableFlashModes.Enqueue(FlashMode.Auto);

7: if (this.Camera.IsFlashModeSupported(FlashMode.On))

8: this.AvailableFlashModes.Enqueue(FlashMode.On);

9: if (this.Camera.IsFlashModeSupported(FlashMode.RedEyeReduction))

10: this.AvailableFlashModes.Enqueue(FlashMode.RedEyeReduction);

11: if (this.Camera.IsFlashModeSupported(FlashMode.Off))

12: this.AvailableFlashModes.Enqueue(FlashMode.Off);

13:

14: this.SetFlashMode();

15:

16: this.AvailableSizes = new Queue<Size>();

17:

18: foreach (Size size in this.Camera.AvailableResolutions)

19: this.AvailableSizes.Enqueue(size);

20:

21: this.SetSize();

22: }

23:

24: private void SetSize()

25: {

26: this.AvailableSizes.Enqueue(

27: this.AvailableSizes.Dequeue());

28:

29: Size size = this.AvailableSizes.ElementAt(0);

30:

31: this.Camera.Resolution = size;

32:

33: Deployment.Current.Dispatcher.BeginInvoke(() =>

34: {

35: this.bResolution.Content = string.Format("{0}x{1}", size.Width, size.Height);

36: });

37: }

38:

39: private void SetFlashMode()

40: {

41: this.AvailableFlashModes.Enqueue(

42: this.AvailableFlashModes.Dequeue());

43:

44: FlashMode mode = this.AvailableFlashModes.ElementAt(0);

45:

46: this.Camera.FlashMode = mode;

47:

48: Deployment.Current.Dispatcher.BeginInvoke(() =>

49: {

50: switch (mode)

51: {

52: case FlashMode.Off:

53: bFlashMode.Content = "OFF";

54: break;

55: case FlashMode.On:

56: bFlashMode.Content = "ON";

57: break;

58: case FlashMode.Auto:

59: bFlashMode.Content = "AUTO";

60: break;

61: case FlashMode.RedEyeReduction:

62: bFlashMode.Content = "R-E";

63: break;

64: }

65: });

66: }

Just after the initialization I copy the available sizes and flash modes in two Queue<T> collections. Then every time a user calls SetSize or SetFlashMode I dequeue the last value and enqueue it on the tail of the collection. So the first element always indicates the current value and it rotates along the available values. According to this value I set the button text and the Resolution and FlashMode properties.

Another parameter you need to tune is the focus of the image. This operation is assisted by the autofocus feature of the camera and also it can be different on different devices. All the phones support the normal center-weighted focus but some other devices can focus on a give point on the frame. So also for focus you have to use the IsFocusAtPointSupported and IsFocusSupported to decide if and how to start focusing. Also you have to pay attention about not start focusing when the camera is capturing a picture.

1: private void MainPage\_Loaded(object sender, RoutedEventArgs e)

2: {

3: // TODO: initialize here

4:

5: this.Camera.AutoFocusCompleted += new EventHandler<CameraOperationCompletedEventArgs>(Camera\_AutoFocusCompleted);

6: }

7:

8: private void listener\_Hold(object sender, Microsoft.Phone.Controls.GestureEventArgs e)

9: {

10: if (this.Camera.IsFocusAtPointSupported) // if supports focus at point calculates the percentage on the screen size

11: {

12: Point point = e.GetPosition(this.cameraView);

13: this.cameraView.BorderBrush = new SolidColorBrush(Colors.Black);

14: double focusX = point.X / this.cameraView.ActualWidth;

15: double focusY = point.Y / this.cameraView.ActualHeight;

16: this.Camera.FocusAtPoint(focusX, focusY);

17: }

18: else if (this.Camera.IsFocusSupported) // if it supports normal focus it simply start

19: {

20: this.cameraView.BorderBrush = new SolidColorBrush(Colors.Black);

21: this.Camera.Focus();

22: }

23: }

24:

25: private void Camera\_AutoFocusCompleted(object sender, CameraOperationCompletedEventArgs e)

26: {

27: Deployment.Current.Dispatcher.BeginInvoke(() =>

28: {

29: this.cameraView.BorderBrush = new SolidColorBrush(Colors.Red);

30: });

31: }

When the focus is completed the AutoFocusCompleted event is raised but you have to be aware it is not on the UI Thread so you have to use the Dispatcher to change the UI aspects.

## And finally capture a shot...

Now we are ready to take a shot and doing whatever we want with it. To capture an image you have to call the CaptureImage method. It is really important to not overlap focus and capture operation because the device will raise an exception if you try to do an operation while the other is in progress. The capture of an image is, as usual, an asynchronous operation. You start the capture and two events notifies when the capture has been completed.

With the CaptureThumbnailAvailable you get a notification when the camera completed the capture and prepared a thumbnail of the resulting picture and then with CaptureImageAvailable you are notified when the full image is ready. Here is a sample code:

1: private void MainPage\_Loaded(object sender, RoutedEventArgs e)

2: {

3: // TODO: initialize here

4:

5: this.Camera.CaptureImageAvailable += new EventHandler<ContentReadyEventArgs>(Camera\_CaptureImageAvailable);

6: this.Camera.CaptureThumbnailAvailable += new EventHandler<ContentReadyEventArgs>(Camera\_CaptureThumbnailAvailable);

7: }

8:

9: private void listener\_Tap(object sender, Microsoft.Phone.Controls.GestureEventArgs e)

10: {

11: this.Camera.CaptureImage();

12: }

13:

14: void Camera\_CaptureThumbnailAvailable(object sender, ContentReadyEventArgs e)

15: {

16: Deployment.Current.Dispatcher.BeginInvoke(() =>

17: {

18: BitmapImage image = new BitmapImage();

19: image.SetSource(e.ImageStream);

20: this.lastShoot.Source = image;

21: this.lastShootFrame.Visibility = System.Windows.Visibility.Visible;

22: });

23: }

24:

25: private void Camera\_CaptureImageAvailable(object sender, ContentReadyEventArgs e)

26: {

27: Deployment.Current.Dispatcher.BeginInvoke(() =>

28: {

29: MediaLibrary ml = new MediaLibrary();

30: ml.SavePictureToCameraRoll(

31: string.Format("{0:yyyyMMdd-HHmmss}.jpg", DateTime.Now), e.ImageStream);

32: });

33: }

As you can see the events are raised on the background thread so also in this case I need to marshal the resulting image to the UI thread before to do anything. When I get the thumbnail I show it on a corner for the screen and finally when the full image is ready it is saved to the camera roll into the media library.

## It's only the first step

The use of the PhotoCamera class gives you a deep control over the phone camera. Controlling the flash, the resolution and the focus, taking shots and presenting the live stream to the UI are for sure great capabilities to integrate into your own application. Nevertheless there are some cases where it may not suffice but as I've already said this is only the first step forward to the complete camera ownership. But about this I will speak another time.

# Chapter 4: Manipulating camera stream

In the previous chapter I introduced how to access the cameras of the phone and emulate a photo camera using the Windows Phone API. These API are really straightforward and almost everyone can take pictures easily for the purposes of the application. But for the most demanding, taking pictures does not suffice.

As an example you can think to an application that needs to measure the light incoming from the lens or that is able to read a barcode without asking the user to click a button.

These two samples, but also lot of other I can imagine, require the access to the raw stream coming from the camera. Only in this way the developer can examine the stream, detect shapes, perform calculations, and so on. And only in this way he can develop the most compelling features.

[**Download the source code**](Source_code/SLPG.Mango_camerastream.zip)

## Accessing the raw stream

For the purpose of demonstrating how to access the camera stream, I've created a simple histogram calculator app. A color histogram is a simple chart showing the distribution of colours along the pixels of the image. Lots of professional and semi-professional cameras on the market offer this feature that is useful to people to improve the quality of the pictures. The approach to use in this case, is to capture a snapshot of the raw stream and then iterate the pixels counting the single color occurrences.

Just to be clear, it cannot be a simple image capture performed using the CaptureImage() method I've described in the last chapter. This method is good to take still pictures to save to the media library, but it cannot be as fast as I need, to calculate the histogram on the fly on the continuously changing image.

To be fast enough I have to access the raw stream and for this purpose they exist two methods called GetPreviewBufferArgb32 and GetPreviewBufferYCbCr. They act making a copy of the current frame, incoming from the stream, to a buffer. The two methods differ for the format of the buffer they fill. The**GetPreviewBufferYCbCr**is able to encode the information using the YCbCr format that is an expression of Luminance and Chrominance. In this way Cb and Cr describe the chrominance component made of a composition of red and blue and on the other side the Y describes the luminance component. The Y taken by itself is an index of light intensity.

The YCbCr format can be useful in some applications but it is for sure hard to manipulate because there is not a direct relationship between each value and a single pixel of the image. Also, for the purpose of calculating a histogram, it would be better to have the picture represented as a simple RGB buffer to iterate, to easily count color occurrences. The **GetPreviewBufferArgb32**method is the best choice because it fills the buffer representing for each pixel an Int32 value that express Red, Green and Blue plus the Alpha component, determining the level of transparency of the pixel. To extract values from the Int32 you can use the following formula

* **Blue** = pixel & 0xf
* **Green** = (pixel & 0xf0) >> 8
* **Red** = (pixel & 0xf00) >> 16
* **Alpha** = (pixel & 0xf000) >> 24

Unfortunately it does not exists a "buffer changed" event so, to write the histogram example, I have to start a timer that trigger a method to recalculate le histogram every single timeout. The calculation of the histogram is not so heavy so the lenght of the timeout may be sufficently short to give to the user the appearance of a chart update almost in realtime, also if this is not exacly the truth. Here is the code to start the timer:

1: public Histogram()

2: {

3: InitializeComponent();

4:

5: // add here further initializations

6:

7: if (PhotoCamera.IsCameraTypeSupported(CameraType.Primary))

8: {

9: this.Camera = new PhotoCamera(CameraType.Primary);

10: this.Camera.Initialized += new EventHandler<CameraOperationCompletedEventArgs>(Camera\_Initialized);

11: this.cameraViewBrush.SetSource(this.Camera);

12: this.CorrectViewFinderOrientation(this.Orientation);

13: }

14: else

15: throw new NotSupportedException("Camera is not supported");

16: }

17:

18: private void Camera\_Initialized(object sender, CameraOperationCompletedEventArgs e)

19: {

20: Timer timer = new Timer(UpdateHistogram, null, 0, 1000);

21: }

In the constructor of the page I make the needed checks to verify that a camer is available and I also setup the brush to project the viewfinder to the screen. This part is exacly the same I've explained in the last chapter. Then, when the camera has been initialized I create the timer to call the UpdateHistogram method once a second. This is not really a fast update but you can try shorter values by yourself.

In the UpdateHistogram method I access the buffer using the GetPreviewBufferArgb32 method and then I calculate the chart on the returned buffer. The calculation is made iterating each pixel of the image and calculating the average value between the RGB components. Then the resulting valaue is used as an index on the array representing the counts for the 256 possible values of the average. At the end of the loop the array contains a count value for each color present in the picture:

1: private void UpdateHistogram(object state)

2: {

3: int max = (int)this.Camera.PreviewResolution.Width \* (int)this.Camera.PreviewResolution.Height;

4: int[] buffer = new int[max];

5: this.Camera.GetPreviewBufferArgb32(buffer);

6:

7: int [] histogram = new int[256];

8:

9: foreach (int pixel in buffer)

10: {

11: byte r = (byte)((pixel >> 16) & 0xff);

12: byte g = (byte)((pixel >> 8) & 0xff);

13: byte b = (byte)(pixel & 0xff);

14: int value = (r + g + b) / 3;

15: histogram[value]++;

16: }

17:

18: int h = histogram.Max();

19:

20: Deployment.Current.Dispatcher.BeginInvoke(

21: () =>

22: {

23: this.Bitmap.Clear();

24:

25: for(int i = 0; i<256; i++)

26: {

27: int v = histogram[i] \* 100 / h;

28: this.Bitmap.DrawLine(i, 100, i, 100 - v, Colors.Red);

29: }

30:

31: this.Bitmap.Invalidate();

32: });

33: }

Interesting to say, the asynchronous nature of the Timer is here an opportunity to perform the histogram calculation on a separated thread without impacting the update of the user interface. Only when the buffer has been scanned and the histogram is calculated the Dispatcher marshal the thread to the user interface and the calculated values are plotted to a WriteableBitmap that shows a horizontal bar chart of 256 values. This chart is then presented to the ui with the call to Invalidate().

## Saving the stream to a file

The capability of directly access the raw stream does not means we are able to save the stream to a file. Given that we are not able to record the every single change in the buffer, taking a snaphot of the stream with GetPreviewBufferArgb32 does not suffice to say we can be as fast as it is required to record a video without hiccups.

It is for this reason it has been included a FileSink class that is able to automatically read the stream from a camera and save it to the IsolatedStorage. This class is really easy to use, and it can automatically encode the raw stream to a MP4 video. This format can be direclty handled by the MediaElement, so you can easily play again a video you have recorded.

The FileSink class acts as a junction between the camera and a file in the isolated storage. To initialize the FileSink class you have to provide an instance of **VideoCaptureDevice** that represent the camera you have to record, and the filename of the output file. To get access to the VideoCaptureDevice you have to use the Capturesource class:

1: this.Source = new CaptureSource();

2: this.Source.VideoCaptureDevice = CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();

The GetDefaultVideoCaptureDevice method gives you a reference to the default camera but you are able to enumerate the available cameras using the GetAvailableVideoCaptureDevices method. Using the returned VideoCaptureDevice class you can detect supported formats, and other properties like the FriendlyName. Once you get the reference to the device you can connect it to the FileSink:

1: this.Sink = new FileSink();

2: this.Sink.CaptureSource = this.Source;

3: this.Sink.IsolatedStorageFileName = "video.mp4";

4:

5: this.Source.Start();

After the device has beend start, using the Start method, the FileSink class begins to pump data from the camera directly to the "video.mp4" file, located in the IsolatedStorage. During this process you can also redirect the output of the CaptureSource to a VideoBrush for the purpose of showing a preview of the recorded video.

1: this.Brush = new VideoBrush();

2: this.cameraView.Background = this.Brush;

3: this.Brush.SetSource(this.Source);

To end the recording you have to call the Stop method on the CaptureSource and then you have to disconnect the FileSink from the source setting its properties to null. This automatically free the output stream and becomes the recorded video available for the play. If you have a preview on a VideoBrush you have also ti disconnect the Source from the brush to be sure you can restart the capture in a later time.

1: this.Source.Stop();

2: this.Sink.CaptureSource = null;

3: this.Sink.IsolatedStorageFileName = null;

4: this.cameraView.Background = null;

5: this.Brush = null;

To play the recorded video you have to connect it to a MediaElement. This means opening a stream to the Isolated Storage file and passing it as a source for the MediaElement:

1: this.Stream = new IsolatedStorageFileStream("video.mp4", FileMode.Open, FileAccess.Read, IsolatedStorageFile.GetUserStoreForApplication());

2: this.media.SetSource(this.Stream);

3: this.media.Play();

As a side note please be aware that the CaptureSource class is also able to still take pictures using the CaptureImageAsync method. It may be an alternative way to the use of the PhotoCamera class. Of course, once you have recorded a video it is in charge to you the code to upload it to the network if you want to persist it to a file on a pc. Isolated Storage in windows Phone is limited only by the size of the available memory, but it is not a good idea to waste it with huge video files.

## Updated example

In the solution I provided with the last chapter I've added another project that shows the things I've explained in these rows. The first example shows how to calculate the histgram and if you load it on your development device you can appreciate the resulting chart. Also another example shows how to record to a file. The application alternatively record to a file and reply the recorded video. The sequence show how to correcly attach and detach the stream to avoid the lock of something.

# Chapter 5: Play with music

If you get the Windows Phone "Mango" update on your phone, and you are use to listen to music on your phone, you have for sure noticed some improvements in the music capabilities of the phone. Since the release of the OS7.0, the music hub is an important application that is able to integrate with the Zune marketplace and enable download and listen of music on the phone.

One of the beautiful capabilities of the hub is the ability of playing music also when the phone is doing other things. Once you started a playlist you can exit from the music hub and the music continue to play in background also if you lock the phone.

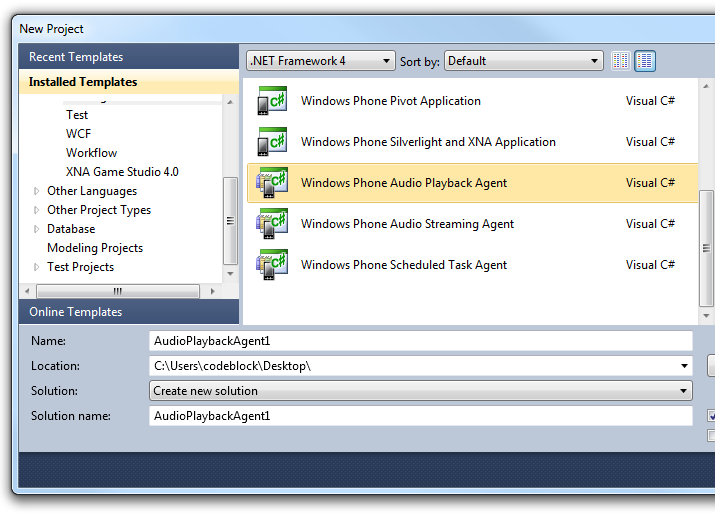
In the new OS7.5 these capabilities are improved because now you get a new set of controls in the lock screen that let you control the music without the need of unlocking the phone. But this is only the most visible change in the music capabilities. From the programming side you have much more opportunities of integrating with the music hub, feeding and controlling the player from inside your applications.

## The role of Agents

Playing background audio is not simply feeding something with a playlist and then leaving it free of managing the events occured during its lifetime. If so, it would be very limitative and it would be much simpler to load songs on the music hub and the let it manages them. In Windows Phone 7.5 we can instead directly manage the playlists and being notified about the commands sent by the user from the lock screen and from the audio player bar that usually opens when you press the volume switch.

As you can figure, these tasks can be accomplished only starting a background thread that runs also when the application exited. It is natural to understand that we have to do some work when there are other application running on the screen and also when the lock screen has been teared down.

From the OS7.5 the phone has the capability of running background agents and to control the tracks you have to play, you have to build a special kind of agent. There are two classes you can inherit from to start this work. The AudioPlayerAgent is made to be feed with tracks that you can stream from the network or from the Isolated Storage; it is the perfect point to create your own player to connect with a live service and so on. With the AudioStreamingAgent instead you can create your own stream on the fly or you can manipulate different formats not directly supported by the build-in player. For the sample of this chapter I will show the basic AudioPlayerAgent. First of all open Visual Studio and select File -> New -> Project

[](http://www.silverlightshow.net/Storage/Users/AndreaBoschin/_____Capture.png)

As you can see there are two project templates that are ready to start your work. Side by side with the generic Scheduled Task Agent there are an Audio Playback Agent and an Audio Streaming Agent. Select the first one as in the figure and then OK to ask Visualo Studio to create the skeleton of the project. The project that is created by Visual Studio contains a single class called AudioPlayer.

The project you have created is a class library, so you have to add its reference to a Windows Phone Application. After you done this task your task is ready to be used. Under the hoods Visual Studio added a reference to your AudioPlayerAgent into the WMAppManifest.xaml. This reference simply adds the class to the extended tasks that run in background:

1: <Tasks>

2: <DefaultTask Name="\_default" NavigationPage="MainPage.xaml" />

3: <ExtendedTask Name="BackgroundTask">

4: <BackgroundServiceAgent Specifier="AudioPlayerAgent" Name="AudioPlaybackAgent1" Source="AudioPlaybackAgent1" Type="AudioPlaybackAgent1.AudioPlayer" />

5: </ExtendedTask>

6: </Tasks>

## Manage the playlist

Now that the Agent is ready to start, it is time to understand its internal work and to write the code to manage the playlist and feed the player. Look at the AudioPlayerAgent class:

1: public class AudioPlayerAgent : BackgroundAgent

2: {

3: protected virtual void OnError(

4: BackgroundAudioPlayer player,

5: AudioTrack track,

6: Exception error,

7: bool isFatal);

8: protected virtual void OnPlayStateChanged(

9: BackgroundAudioPlayer player,

10: AudioTrack track,

11: PlayState playState);

12: protected virtual void OnUserAction(

13: BackgroundAudioPlayer player,

14: AudioTrack track,

15: UserAction action,

16: object param);

17: }

There are two key methods you have to handle. The OnUserAction method is made to notify the actions of the user on the playback controls. When the user hits a button on the lock screen this method is called and you are in charge of taking the right action on the current playing track or on the playlist. The method notifies about track-related actions like Play, Pause, Stop, Forward and Rewind but also on SkipNext and SkipPrevious that ask to change the currently playing track.

The OnPlayerStateChanged method instead, notifies about the progress of the playback and about the changes of its state. So, as an example, if the track is ended this method is called with PlayState.TrackEnded and you have to feed the player with the next track available. Mostly important is that you get notifications also about the actions you trigger in the OnUserAction method.

The AudioPlayer class created by the Visual Studio template already implements lot of the logic needed to make it work. The sole thing it misses is the tracks. So at the very start of the class add an array of AudioTrack instance and a \_current property that identified the number of the current track running in this array:

1: private static volatile int \_current = 0;

2:

3: private AudioTrack[] tracks = new AudioTrack[]

4: {

5: new AudioTrack(new Uri("Hard As Rock.mp3", UriKind.Relative),

6: "Hard as rock", "AC/DC", "Ballbreaker", new Uri("Hard As Rock.jpg", UriKind.Relative)),

7: new AudioTrack(new Uri("Run Around.mp3", UriKind.Relative),

8: "Run around", "Blues traveller", "Four", new Uri("Four.jpg", UriKind.Relative)),

9: };

In this sample I statically declare the array but you can obviously load it from every kind of media. It can be loaded from the network or generate it dinamically. Also you do not need to have a static array but you can also generate the next track at runtime on a random basis. Important to say is that the \_current variable is declared static because the AudioPlayer class is not persistent but an instance is created every time an action is required.

The AudioTrack class represents a single track of the playlist. It is provided with properties for Title, Artist, Album, AlbumArt and so on. You can use these properties to provide information to the phone and it shows them whenever it is required. The template of the AudioPlayer class declares a GetNextTrack and GetPreviousTrack methods. Implementing the body of these methods let you feed the player with the right track.

1: private AudioTrack GetNextTrack()

2: {

3: AudioTrack track = this.tracks[\_current];

4: if (++\_current >= this.tracks.Length) \_current = 0;

5: return track;

6: }

7:

8: private AudioTrack GetPreviousTrack()

9: {

10: AudioTrack track = this.tracks[\_current];

11: if (--\_current < 0) \_current = this.tracks.Length - 1;

12: return track;

13: }

## Starting the task

Adding this class to the WMAppManifest.xaml does not automatically start the playback. To make the last step you have to use the BackgroundAudioPlayer class. This class is designed to simulate the actions of a user and it has methods to Play, Pause, Stop and so on. So when you need to start the audio you have to call the Play method. After this action the phone loads the AudioPlayerAgent and starts the music. You can also use this class to control le lifetime of the track and playlist from your own application user interface:

1: private void bPlay\_Click(object sender, RoutedEventArgs e)

2: {

3: if (BackgroundAudioPlayer.Instance.PlayerState != PlayState.Playing)

4: BackgroundAudioPlayer.Instance.Play();

5: }

6:

7: private void bPause\_Click(object sender, RoutedEventArgs e)

8: {

9: if (BackgroundAudioPlayer.Instance.CanPause &&

10: BackgroundAudioPlayer.Instance.PlayerState == PlayState.Playing)

11: BackgroundAudioPlayer.Instance.Pause();

12: }

13:

14: private void bStop\_Click(object sender, RoutedEventArgs e)

15: {

16: if (BackgroundAudioPlayer.Instance.PlayerState == PlayState.Playing)

17: BackgroundAudioPlayer.Instance.Stop();

18: }

19:

20: private void bNext\_Click(object sender, RoutedEventArgs e)

21: {

22: if (BackgroundAudioPlayer.Instance.PlayerState == PlayState.Playing)

23: BackgroundAudioPlayer.Instance.SkipNext();

24: }

25:

26: private void bPrev\_Click(object sender, RoutedEventArgs e)

27: {

28: if (BackgroundAudioPlayer.Instance.PlayerState == PlayState.Playing)

29: BackgroundAudioPlayer.Instance.SkipPrevious();

30: }

As you can see the PlayerState property is used to detect the current state of the player. Calling Play when the player is currntly playing will raise an exception so you have to carefully test the current state to avoid unwanter errors. The PlayerState is also useful if it is used in the PlayerStateChanged event that is notified when the value changes. In this snipped it is used to change the buttons IsEnabled property according with the state:

1: private void Instance\_PlayStateChanged(object sender, EventArgs e)

2: {

3: PlayState state = BackgroundAudioPlayer.Instance.PlayerState;

4:

5: this.bPlay.IsEnabled = state != PlayState.Playing && state != PlayState.Unknown;

6: this.bPause.IsEnabled = state == PlayState.Playing && state != PlayState.Unknown;

7: this.bStop.IsEnabled = state == PlayState.Playing && state != PlayState.Unknown;

8: this.bNext.IsEnabled = state == PlayState.Playing && state != PlayState.Unknown;

9: this.bPrev.IsEnabled = state == PlayState.Playing && state != PlayState.Unknown;

10:

11: if (BackgroundAudioPlayer.Instance.Track != null)

12: {

13: this.txtTrack.Text = BackgroundAudioPlayer.Instance.Track.Title;

14: this.txtAuthor.Text = BackgroundAudioPlayer.Instance.Track.Artist;

15: this.txtAlbum.Text = BackgroundAudioPlayer.Instance.Track.Album;

16: this.txtDuration.Text = BackgroundAudioPlayer.Instance.Track.Duration.ToString();

17:

18: using (IsolatedStorageFile file = IsolatedStorageFile.GetUserStoreForApplication())

19: {

20: string coverFile = Uri.UnescapeDataString(BackgroundAudioPlayer.Instance.Track.AlbumArt.OriginalString);

21:

22: if (file.FileExists(coverFile))

23: {

24: Stream stream = file.OpenFile(coverFile, FileMode.Open, FileAccess.Read);

25:

26: BitmapImage cover = new BitmapImage();

27: cover.SetSource(stream);

28: this.imgCover.Source = cover;

29: }

30: }

31: }

32: }

Finally you have access to the currently playing track. The BackgroundAudioPlayer instance exposes a TrackProperty that is the current AudioTrack provider by the background agent. In this code I use this information to display the AlbumArt and other properties on the page:

## When to use Audio Agents

While I was working on the samples for this chapter I often asked myself about the usefulness of these features. I think they are a good entry point for provider that wants to feed phones with music service that are alternative to the zune marketplace. With the ability of connect to the network to stream content and to save it to the marketplace I think it will be very easy to integrate some source to the phone. Also I think there are interesting opportunities in the AudioStreamingAgent due to its capability of playing audio that is generated on board of the phone. As for any other feature of this operating system ther will be someone ready to take advantage of them.

# Chapter 6: Use Background agents

One of the most important features added in Windows Phone 7.5 is the multithreaded environment. This missing feature was very criticized in the previous version, but the main reason of its lack was the high battery consumption that is usual in multithread phones. In this release the team has agreed to add the multithread capability but it worked hard to reach a good balance between multithreading and battery life. This is the main reason for the introduction of Background Agents that are a way to manage the battery drain in junction with parallel work made by running applications.

[**Download the source code**](Source_code/SLPG.WP7_Mango.zip)

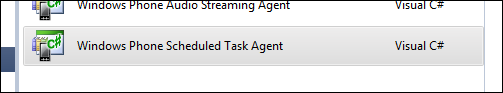
## Understanding Agents

Starting from the release of OS7.5, the developer can create background agents. As the name suggests, an agent works in background, hosted in a separated thread and is able to make some work. Important to say is that an agent must be initiated by an application. The application has to register the agent on the ScheduledActionService then it can continue its work or exit and the agent will be called by the Scheduler seamless. This imply that agents are not daemons; there is not any way to register and run the agent like a gui-less application that perform its work continuously. This just because agents are designed with battery life in mind so they only can be scheduled to run in two ways:

1. **PeriodicTask** – These agents are called every 30 minutes (the timeout is fixed and cannot be changed). They can run for a very short timeframe and perform lightweight tasks
2. **ResourceIntensiveTask –**These agents are triggered by a set of requirements like processor activity, network, power and so on. They can work for a relatively long period.

The limits imposed to the developer are really heavy; you can only register two background agents per application, and they have to be of different type. This means your application can only have a single PeriodicTask and a single ResourceIntensiveTask.

To create the background agent you can start in the same way we did when I’ve explained the Audio Playback Agent. In Visual Studio you can open the Add New Project dialog and choose the Windows Phone Scheduled Task Agent.

[](http://www.silverlightshow.net/Storage/Users/AndreaBoschin/image_2.png)

This option will create a new project and a ScheduledAgent class that is already configured and ready to be filled with your implementation. Then, when you connect this project with the main application, the WMAppManifest.xml file is modified with the reference to the class. Differently from an Audio Playback Agent this does not suffice to make your agent up and running. You have obviously to implement you agent logic but you have also to register the agent with the ScheduledActionService:

1: private void RegisterTasks()

2: {

3: string taskName = ScheduledAgent.Name;

4:

5: PeriodicTask existing = ScheduledActionService.Find(taskName) as PeriodicTask;

6:

7: if (existing != null)

8: {

9: ScheduledActionService.Remove(taskName);

10: }

11:

12: PeriodicTask task = new PeriodicTask(taskName)

13: {

14: Description = "Download local images from panoramio"

15: };

16:

17: ScheduledActionService.Add(task);

18: }

This code may be added to the App.xaml.cs and its purpose is to check if an instance of the agent already exists before adding it to the ScheduledAgentService. The service accepts an instance of PeriodicAgent or ResourceIntensiveTask but the sole reference to your implementation is in the taskName that has to be the name specified in the WMAppManifest.xaml. The PeriodicTask, I used in this example, has some other properties in addition to Description but none of them let you specify the timing of the schedule. Unfortunately it is fixed to 30 minutes.

## Create an agent

Implementing the agent is very straightforward. The class that implements the agent is inherited by ScheduledTaskAgent and it exposes an OnInvoke method. This method is called every time the scheduler needs to execute your code. The method provides a single parameter that is useful to know some information about the task that is running. As an example you can know the LastScheduledTime or the LastExitReason that contains the following values:

1: public enum AgentExitReason

2: {

3: None,

4: Completed,

5: Aborted,

6: MemoryQuotaExceeded,

7: ExecutionTimeExceeded,

8: UnhandledException,

9: Terminated,

10: Other,

11: }

Another important reason to use the provided parameter is to understand if your task is running as a PeriodicTask or ResourceIntensiveTask. This is useful when you register the sample agent for both the types. You can understand when you are in one case or in the other examining the underlying type:

1: if (task is PeriodicTask)

2: {

3: // do the periodic job

4: }

5: else

6: {

7: // do the resource intensive job

8: }

You have to be careful when you implement the agent because the type you choose gives you some limits. Particularly the PeriodicTask is designed to be lightweight and the code you put inside the OnInvoke method must have a very light thumbprint in terms of elaboration time and used resources. As the AgentExitReason testifies, the runtime can kill your work in case of MemoryQuotaExceeded and in case of ExecutionTimeExceeded. A periodic task can run only for about 25 seconds and can allocate a maximum of 5 MB of memory. A resource intensive task instead can run for about 10 minutes before the runtime kill it.

Given these limits and the timeout of 30 minutes between one execution and the other, you can easily understand that debugging an agent may become a nightmare. If you run the agent in its normal context you have to wait for 30 minutes before the breakpoints are hit. On the other side if you test the agent outside of the context you can easily overwhelm the resource limits and then discover that your agent won’t run when executed by the scheduler. To simplify the debug tasks it exists a LaunchForTest method into the ScheduledActionService. This method lets you specify a timeout, useful to exit the application manually, then it runs the agent once without waiting for the expiration of the regular timeout. You can only debug the agent once per session but this method is important to speed-up your work.

ScheduledActionService.LaunchForTest(task.Name, TimeSpan.FromSeconds(5));

## A funny example

To demonstrate the use of agents I’ve created an interesting and beautiful example. If you used google maps at least once, you know about Panoramio. It is a funny service that is able to show geo referenced photos over the google map. For this example I’ve managed to acquire the GPS position of the device when the agent run, download the available images and populate its tile with one randomized on the first twenty. The effect is pretty beautiful, expecially if you are moving during a travel because once on every 30 minutes the tile is updated with a photo of a place located near you.

In the Invoke method I use the GeoCoordinateWatcher to read the current position. The problem here is about the execution model of the GeoCoordinateWatcher. It is made to notify when the position has changed raising an event. Here is the code I’ve prepared:

1: protected override void OnInvoke(ScheduledTask task)

2: {

3: GeoCoordinateWatcher watcher = new GeoCoordinateWatcher(GeoPositionAccuracy.High)

4: {

5: MovementThreshold = 10

6: };

7:

8: watcher.PositionChanged += (s, e) =>

9: {

10: watcher.Stop();

11:

12: GeoCoordinate position = e.Position.Location;

13:

14: if (position.IsUnknown)

15: {

16: Random rnd = new Random(DateTime.Now.Millisecond);

17: position = this.PointOfInterests[rnd.Next(0, this.PointOfInterests.Length - 1)];

18: }

19:

20: this.UpdateImage(position);

21: };

22:

23: watcher.Start(false);

24: }

In this snippet I initialize the GeoCoordinateWatcher, and then I attach the PositionChanged event. This event immediately notifies the new position and it is here I retrieve the current position. In the Positionchanged event I check the retrieved position; if it is Unknown I use a special array, filled with a number of random positions, to provide a random location to display.

In the UpdateImage I use the Panoramio’s API to retrieve the available images using the Json format. In the GetRandomLocalImage I raffle an image from the set I found so in case of repeated update the photo will change every time:

1: private void UpdateImage(GeoCoordinate position)

2: {

3: if (position != null)

4: {

5: PanoramioService.GetRandomLocalImage(

6: position.Latitude,

7: position.Longitude,

8: s =>

9: {

10: this.LoadImage(s);

11: this.NotifyComplete();

12: },

13: ex => this.NotifyComplete());

14: }

15: }

Important to say, no matter whether I found or not an image, I call the NotifyComplete method. This method let the runtime know that the work is done. Finally in the LoadImage method I use the new API dedicated to the tiles to change the background image and the title according with the found image:

1: private void LoadImage(PanoramioImageResult image)

2: {

3: ShellTile tile =

4: ShellTile.ActiveTiles.FirstOrDefault();

5:

6: if (tile != null)

7: {

8: tile.Update(

9: new StandardTileData

10: {

11: Title = image.Title,

12: BackgroundImage = image.File,

13: Count = null

14: });

15: }

16: }

## A beautiful opportunity

There is no doubt that Background Agents are something missing in OS7.0. There are a lot of cases when you can enrich your applications with some background notifiers and tasks. I hope in the next releases it will be removed some of the annoying limits, first of all the fixed 30 minutes schedule, that is really hard to understand, also with the purpose of sparing battery. I think a little configurability would be better.

# Chapter 7: Using advanced tiles API

There is not any doubt, the first thing you meet when you use Windows Phone are the tiles. These are the large squares on the home screen that identify some applications and they are also a distinguishable character that makes your Windows Phone unique.

As you know for sure, the tiles can be attached or detached from the home screen and some particular software can take advantage of double size tiles. While this is not a feature available to developers, in OS7.5, the tiles gained new features and a new set of APIs that you can use to enrich your applications. As an example you are now able to update your tiles from inside the application and you can use double faced tiles to improve information to the user. In this chapter I would like to explore these new features and show how to take advantage of them, while porting your software to the new operating system.

## Accessing and changing your tile(s)

Once your application is running, the user can pin its tile on the home screen. If this happened, you can have access to a bunch of tile's properties you can change at every time while the application is running, but please take note that it implies that the tile exists. Obviously, none of these APIs can work if your application is not pinned.

The root of your work is the ShellTile class that gives you access to a collection of ActiveTiles. At the very first place in this collection there is a structure that represents the main tile of your application. Reading the previous paragraph, you can expect this collection is empty when you application has not been pinned, but it is not true. The collection always contains at least one element, and you can update it every time also if the application's tile is not in the home screen. This does not cause a visible change but if the user chooses to pin the application at a later time its tile will reflect these changes. The ShellTile class returned by this collection contains an "Update" method that is useful to change the tile's content:

1: StandardTileData data = new StandardTileData

2: {

3: Title = "My tile!",

4: Count = 10,

5: BackgroundImage = new Uri("/Background.png", UriKind.RelativeOrAbsolute),

6: BackTitle = "This is the back",

7: BackContent = "Hallo!",

8: BackBackgroundImage = new Uri("/Background.png", UriKind.RelativeOrAbsolute)

9: };

10:

11: ShellTile.ActiveTiles.First().Update(data);

The properties of the StandardTileData class reflect the parts of the tile and are divided in two categories: Front and Back. So, we have a "Title" property that reflects the title of the tile on the front side and the "BackTitle" that has the same meaning for the back side. This led to the consideration that in OS 7.5, tiles can have two sides available. It suffices you set the value of one of the "Back" properties and the tile automatically activates the back side on a random schedule.

[](http://www.silverlightshow.net/Storage/Users/AndreaBoschin/______Capture.png)

Tiles have also a background image, usually of the size of 173x173 pixels, that fills the entire square. The image must reside in the project as a resource and we can refer to it with a relative uri that starts at the root of the project. So "/Background.png" refers to the standard image, usually created by visual studio in the default project. Of course you can set different images for each side of the tile.

If you generate images on the fly, while the application is running, you can also use isolated storage to peek up images to show in the tiles. For this purpose you have to generate images in the standard path "/Shared/ShellContent", and then use the "isostore:" prefix when you refer to it in the tile's uri. Here is an extended example that shows this tecnique with a generated background image. The image contains a simple gradient as shown on the left side.

1: private const string TilePath = "/Shared/ShellContent/tile.jpg";

2:

3: void MainPage\_Loaded(object sender, RoutedEventArgs e)

4: {

5: WriteableBitmap bmap = new WriteableBitmap(173, 173);

6:

7: for (byte x = 0; x < 173; x++)

8: for (byte y = 0; y < 173; y++)

9: bmap.Pixels[y \* 173 + x] = (0xff << 24) | (x << 16) | y;

10:

11: using (IsolatedStorageFile file = IsolatedStorageFile.GetUserStoreForApplication())

12: {

13: if (file.FileExists(TilePath))

14: file.DeleteFile(TilePath);

15:

16: using (IsolatedStorageFileStream stream = file.CreateFile(TilePath))

17: bmap.SaveJpeg(stream, 173, 173, 0, 100);

18: }

19: }

20:

21: private void bCheckTiles\_Click(object sender, RoutedEventArgs e)

22: {

23: StandardTileData data = new StandardTileData

24: {

25: Title = "andrea",

26: Count = 10,

27: BackTitle = "This is the back",

28: BackContent = "Hallo!",

29: BackBackgroundImage = new Uri("isostore:/Shared/ShellContent/tile.jpg", UriKind.RelativeOrAbsolute)

30: };

31:

32: ShellTile.ActiveTiles.First().Update(data);

33: }

Finally you probably noticed an asimmetry in front and back properties. While front properties have a "Count" value, the background can specify a "BackContent". The "Count" property is made to specify a number to show on the right-top corner. This may be useful to applications that can check for email messages and so on. The BackContent instead, is used to fill the main content of the tile background. The tile figure on the left side shows the word "Hallo!" in the place where content is displayed.

## Schedule tile updates

Once you configured your application's tile, your next need is probably to change the tile's content to notify the user about a service you are monitoring. This usually means use a time-based schedule that is able to check on a server if there is something to notify. Since OS 7.0, Windows Phone has a ShellTileSchedule. It was very limited because you were only able to update the background image, using a remote Uri to call on a scheduled basis. In OS 7.5 this has not changed a lot. The only thing we get added is the capability of also update secondary tiles (I will speak about secondary tiles in a few). You can connect the ShellTileSchedule to an arbitrary tile using the ActiveTiles collections as shown here:

1: ShellTileSchedule schedule = new ShellTileSchedule(ShellTile.ActiveTiles.First())

2: {

3: Interval = UpdateInterval.EveryHour,

4: MaxUpdateCount = 0,

5: Recurrence = UpdateRecurrence.Interval,

6: StartTime = DateTime.Now,

7: RemoteImageUri = new Uri("http://xamlplayground.org/wp7updates/tile.jpg")

8: };

9:

10: schedule.Start();

The ShellTileSchedule constructor accepts a reference to a ShellTile in the collection so you can schedule multiple updates, one for each tile of your application.

As I shown in the example of the previous chapter about background agents, Windows Phone 7.5 gives an additional opportunity to schedule tile updates. You are able to use a ScheduledAgent and from inside the agent you are able to access the ActiveTiles collection and update almost every property you think useful to notify the user, with the sole limitation of a fixed schedule of 30 minutes. In the Panoramio example I showed how to access GPS position and retrieve a random image from Panoramio to update the application tile with a local image. Please check out the previous chapter for a detailed explanation.

## Using secondary tiles

A great new opportunity with Windows Phone 7.5, is the ability of configure more than a tile, to pin to the home screen. These tiles can work as shortcuts for services exposed by your applications. To make a practical example, you can think at secondary tiles to expose different locations for a weather channel or to have a shortcut to airplanes flights and so on. The sole limit is your fantasy.

A secondary tile is simply created using the static Create method only on user input. This means your application cannot create tiles automatically, but only when the user explicitly ask it, interacting with the application interface. To create the tile you have to fill a StandardTileData class and pass it to the method. As a result the application exits and the user is brought to the tile position in the home screen. This also means that you can only create a secondary tile for each session because, the process close the application every time. Here is the code to create the tile:

1: private void PinToStart(string address)

2: {

3: Server server = Repository.GetServer(address);

4:

5: if (server != null)

6: {

7: ShellTile tile = ShellTile.ActiveTiles.FirstOrDefault(

8: t => t.NavigationUri.ToString().EndsWith("server=" + address));

9:

10: if (tile == null)

11: {

12: string destFileName = CopyImageToShellContent(address);

13:

14: StandardTileData newTile = new StandardTileData

15: {

16: Title = server.Name,

17: Count = 0,

18: BackgroundImage = new Uri("Background.png", UriKind.Relative),

19: BackTitle = server.Name,

20: BackBackgroundImage = new Uri(destFileName, UriKind.Absolute),

21: BackContent = AppStrings.Title

22: };

23:

24: ShellTile.Create(new Uri("/Pages/VNCPanorama.xaml?server=" + address, UriKind.Relative), newTile);

25: }

26: }

27: }

This sample, taken directly from the next release of my [Silver VNC client for Windows Phone 7](http://www.xamlplayground.org/post/2011/11/18/Silver-VNC-10-for-Windows-Phone-Mango.aspx), shows how to create the tile. The important thing to note is the uri you have to pass to the Create method. This uri (called NavigationUri) is the one called when the user hits the pinned tile. It indicates the page inside your application that will be used as landing page for the request and can have multiple query string parameters to let the application decide how to answer to the request. Thanks to these parameters you can have multiple tiles that connect to the application with different actions. Inside the page that receives the user request you can parse the query string to evaluate the action to perform:

1: protected override void OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e)

2: {

3: this.txtName.Text = string.Empty;

4: this.txtAddress.Text = string.Empty;

5: this.Load();

6: this.EvaluateFromPinnedTile(e);

7: base.OnNavigatedTo(e);

8: }

9:

10: private void EvaluateFromPinnedTile(NavigationEventArgs e)

11: {

12: if (e.NavigationMode == System.Windows.Navigation.NavigationMode.New)

13: {

14: Match match = Regex.Match(e.Uri.ToString(), ".\*?server=(?<address>.\*)");

15:

16: if (match.Success)

17: {

18: string address = match.Groups["address"].Value;

19:

20: Server server = Repository.GetServer(address);

21:

22: if (server != null)

23: this.Connect(server);

24: else

25: MessageBox.Show(string.Format(AppStrings.Message\_UnknownServer, address));

26: }

27: }

28: }

Into the OnNavigated method I call the EvaluateFromPinnedTile method that is responsible to check if the navigation occurs because the user hits the pinned tile. In the method I check the NavigationMode property that is always set to "New" when it is raised from a secondary tile and the use a regular expression to effectively check the incoming uri and retrieve the address. If the call is validated then I retrieve the Server information, from the local database, and then I call the Connect method to start the VNC connection.

To be double sure to not have exceptions I also check that the server exists on the database. Also if I delete the tile when the server is deleted from the database, this check is more defensive. Here is the deletion code:

1: private void TryUnpinFromStart(string address)

2: {

3: ShellTile tile = ShellTile.ActiveTiles.FirstOrDefault(t => t.NavigationUri.ToString().EndsWith("server=" + address));

4:

5: if (tile != null)

6: {

7: tile.Delete();

8: }

9: }

While programming secondary tiles, you must be cautious because the user can always remove the pinned tile from the home screen, without the application get any warn about it. So please always be careful about what you get and always try to be in sync between secondary tiles and application data.

## Tile your apps.

Tiles are probably the most distinctive feature you have in Windows Phone 7. As it happen for every hard decision you can only love or hate them. I meet lot of people that take tiles as an example of the wrong of the Windows Phone and lot of other people that think in the opposite way. My think is that for common people tiles are a great opportunity of access all the capabilities of the phone without have to deal with complicated interfaces. Only hit and pin, is the only thing you need. From the developer side, using tiles correctly gives you an additional gear to make you applications more flexible and effective. However, my think is that tiles are great; given I love the simplicity, they are one of the best features of my favorite phone.

# Chapter 8: Using Sockets

Also if the technical specifications of Windows Phone 7.0 stated that it is compatible with Silverlight 3.0, this only means that every feature you can use in the phone is available on the desktop but not viceversa. Sockets are a clear demonstration of this sentence. While they are perfectly available on Silverlight 3.0, they are locked down in the phone for the, so called, "security purposes". Curiously if you watch at the codebase of Silverlight for Windows Phone 7.0 you see that, the Socket class exists but it is declared "internal".

The new OS 7.5 have made Sockets available to developers, opening the way to a wide series of applications that uses this low-level communication way. Sockets are for sure difficult from the programming side, but they offer a connected/disconnected, fast and reliable way of exchange data that is unreplaceable for many purposes.

[**Download the source code**](Source_code/XPG.NTP.zip)

## What sockets can do (and what they can't)

Sockets are something that we use every day. Also if we are not aware of this, they are at the very base of each network protocol, comprised HTTP, SMTP, PO3, and so on. Their purpose is to create a communication channel between two boundaries where they can send and receive something that usually is a binary message. There are two ways to create a Socket connection. The TCP and the UDP. Since TCP is an always connected protocol that is tailored to exchange continuous streams, UDP is mostly made to exchange fast and short packets of information - called datagrams - while the boundaries aren't continuously connected. To make a practical example, HTTP is based on the TCP protocol where the client and the server establish a connected channel and they both send and receive on this channel until the communication ends. On the other side, DNS is mostly UDP based because the client sends a request datagram and the server answers with another datagram but there is not a real channel established between the two boundaries. Do not underestimate UDP since it is often very effective. Infact a number of streaming protocols are based on it.

While Silverlight only supports TCP, Windows Phone 7.5 also supports UDP Sockets. I'm always really amazed about the confusion of team's choices, but to me it is really not understandable the reason that leads to diverge the sockets capabilities in this matter. But so it is. So, while in Silverlight for desktop we can only use TCP and a limited set of ports, if application is not fully trusted, with Windows Phone 7.5 we have the very full set of knifes to work with.

No matter you are using a TCP or UDP connection, a communication is ruled by the same process and by the same structures. Once you have chosen the endpoint to connect to, in terms of ip address and port, the process of communications is the following:

1. Create the Socket class
2. Connect the socket
3. Send and receive data
4. Shutdown the connection

The correct sequence of operations at point 3 is determined by the network protocol you are implementing. As an example if you are placing an HTTP call, first of all you send the request and receive the response, but there are many cases where you can simply start to receive a continuous stream because the protocol continuously broadcast packets.

All the phases of the socket connection are ruled by at least an instance of SockerAsyncEventArgs. This class contains the reference to the remote endpoint, the operation in progress/completed, the data that is sent to or received by the peer. In a simple connection you will have only a single instance but, for performance purposes (e.g. in a multithreaded environment), you can have a pool of instances to use for the various operations.

## Implementing a simple UDP protocol (NTP)

For the sake of the chapter I will show now how to implement a simple protocol. The NTP protocol I will show is very basic, just because I do not want to spend most of the time explaining the protocol itself. NTP stands for Network Time Protocol and despite the very simple datagram you have to exchange over an UDP socket, the RFC is really complex because of the server implementation that have to ensure redundancy and precision across multiple sources.

To make an NTP request you have to do the following:

1. Connect to a NTP server on port 123
2. Prepare a datagram of 48 zeroed bytes with the first byte set to 0x1b
3. Send the datagram over UDP
4. Receive the same datagram filled with time information
5. Close the socket
6. Decode and use the result

The returned bytes represent a DateTime and I do not enter in the format of the response. If you are interested simply watch at the decoding method in my example. The first thing to do is to create an NtpClient class. This class will represent a single server and is initialized with the remote endpoint:

1: public class NtpClient

2: {

3: private Socket Socket { get; set; }

4: private SocketAsyncEventArgs Args { get; set; }

5:

6: public NtpClient(EndPoint endPoint)

7: {

8: this.Args = new SocketAsyncEventArgs

9: {

10: RemoteEndPoint = endPoint

11: };

12:

13: this.Args.Completed += new EventHandler<SocketAsyncEventArgs>(Operation\_Completed);

14: }

15:

16: private void Operation\_Completed(object sender, SocketAsyncEventArgs e)

17: {

18: }

19: }

In this first part I create an instance of the SocketAsyncEventArgs and initialize it with the RemoteEndPoint. This instance is shared by all the network calls so it is defined as a private member property. The code also attaches the Completed event of this instance. Every time I will perform an action on the socket, using this instance, the Completed event is raised and the LastOperation property will be set with a value representing the action I did.

The GetNetworkTimeAsync method initializes the Socket class and initiates the connection. Under the hoods, the meaning of "ConnectAsync" is different between TCP and UDP but it has to be made for both the protocols. Here I am using an UDP connection so the values in the ctor indicate ProtocolType.Udp and a SocketType.Dgram. If I would use a TCP channel I should use SocketType.Stream and ProtocolType.Tcp.

1: public void GetNetworkTimeAsync()

2: {

3: this.Socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);

4: this.Socket.ConnectAsync(this.Args);

5: }

From this point, the lifetime of the class is governed by the Operation\_Completed event. Every time I perform an action this method collects the result. So when I call connect the method receives the result, but the same happen when I call SendAsync and ReveiceAsync. Inside the Operation\_Completed I create a chain of operation, always calling the following action as the result of another.

1: private void Operation\_Completed(object sender, SocketAsyncEventArgs e)

2: {

3: if (e.SocketError == SocketError.Success)

4: {

5: switch (e.LastOperation)

6: {

7: case SocketAsyncOperation.Connect:

8: this.SendRequest();

9: break;

10: case SocketAsyncOperation.Send:

11: this.ReceiveResponse();

12: break;

13: case SocketAsyncOperation.Receive:

14: this.ReadAndClose();

15: break;

16: }

17: }

18: else

19: this.OnError(e.SocketError);

20: }

The chain indicates that following the Connect I call SendRequest then ReceiveResponse and finally ReadAndClose. These decisions are wrapped from an if construct that handles the socket errors. Only when e.SocketError equals Success the chain continues. The SendRequest method creates the datagram and sends it over the channel:

1: private void SendRequest()

2: {

3: byte[] ntpData = new byte[48];

4: Array.Clear(ntpData, 0, ntpData.Length);

5: ntpData[0] = 0x1b;

6:

7: this.Args.SetBuffer(ntpData, 0, ntpData.Length);

8: this.Socket.SendAsync(this.Args);

9: }

As you can see the 48 bytes array is cleared and the first byte is set to 0x1b then the entire buffer is assigned to the SocketAsyncEventArgs using the SetBuffer method. Finally it is passed to the SendAsync method so it is sent to the connected peer. The ReceiveResponse method instead, does not change the SocketAsyncEventArgs but simply calls the ReceiveAsyncMethod. The purpose of this method is not receiving bytes but putting the socket in receive mode so when a packet arrives it is received.

1: private void ReceiveResponse()

2: {

3: this.Socket.ReceiveAsync(this.Args);

4: }

When the packet arrives, it is written inside the buffer prepared into the SocketAsyncEventArgs and I receive this data in the Operation\_Completed method when LastOperation is Receive. This meand that between ReadAsync and Operation\_Completed it may pass a long time depending on how fast is the answer from the server.

1: private void ReadAndClose()

2: {

3: byte[] ntpData = this.Args.Buffer;

4:

5: const byte offsetTransmitTime = 40;

6: ulong intpart = 0;

7: ulong fractpart = 0;

8:

9: for (int i = 0; i <= 3; i++)

10: intpart = 256 \* intpart + ntpData[offsetTransmitTime + i];

11:

12: for (int i = 4; i <= 7; i++)

13: fractpart = 256 \* fractpart + ntpData[offsetTransmitTime + i];

14:

15: ulong milliseconds = (intpart \* 1000 + (fractpart \* 1000) / 0x100000000L);

16:

17: TimeSpan timeSpan = TimeSpan.FromTicks((long)milliseconds \* TimeSpan.TicksPerMillisecond);

18:

19: DateTime dateTime = new DateTime(1900, 1, 1);

20: dateTime += timeSpan;

21:

22: DateTime networkDateTime = dateTime.ToLocalTime();

23:

24: this.OnCompleted(networkDateTime);

25:

26: this.Socket.Shutdown(SocketShutdown.Both);

27: this.Socket.Close();

28: this.Socket = null;

29: }

The ReadAndClose method receives the buffer and decodes its content according with the NTP protocol specifications. Finally it calls ShutDown and Close to terminate the connection. After this the Socket instance is cleared because on the next request I have to create it again.

## Using the NtpClient

Once the NtpClient class has been created it is time to use it in a Windows Phone application. Using this class is straightforward. You create the client instance and then call the GetNetworkTimeAsync every time you need to request the datetime. The class exposes two events Completed and Error that are raised accordin with the result of the request:

1: public partial class MainPage : PhoneApplicationPage

2: {

3: public NtpClient Client { get; set; }

4:

5: public MainPage()

6: {

7: InitializeComponent();

8:

9: this.Client = new NtpClient(new DnsEndPoint("0.pool.ntp.org", 123));

10: this.Client.Completed += new EventHandler<NtpQueryCompletedEventArgs>(Client\_Completed);

11: this.Client.Error += new EventHandler<NtpQueryErrorEventArgs>(Client\_Error);

12: }

13:

14: private void Button\_Click(object sender, RoutedEventArgs e)

15: {

16: this.Client.GetNetworkTimeAsync();

17: }

18:

19: private void Client\_Completed(object sender, NtpQueryCompletedEventArgs e)

20: {

21: this.Result.Text = e.Result.ToString(CultureInfo.CurrentCulture);

22: }

23:

24: private void Client\_Error(object sender, NtpQueryErrorEventArgs e)

25: {

26: this.Result.Text = e.Error.ToString();

27: }

28: }

The NtpClient class does not gracefully handles a double connection request but simply throw away the preceding destroying the socket before creating another one. This is for sure an improvement you can do by yourself.

The sockets are a great opportunity for doing beautiful applications. An example of the power of sockets is my [Silver VNC Client for Windows Phone](http://www.xamlplayground.org/post/2011/11/18/Silver-VNC-10-for-Windows-Phone-Mango.aspx) that is available on marketplace The VNC client uses a TCP channel to connect to a VNC server and remotely view the desktop of a computer.

I hope you enjoyed this ebook.

# About the Author

Andrea Boschin is 41 years old from Italy and currently lives and works in Treviso, a beautiful town near Venice. He started to work in the IT relatively late after doing some various jobs like graphic designer and school teacher. Finally he started to work into the web and learned by himself to program in VB and ASP and later in C# and ASP.NET. Since the start of his work, Andrea found he likes to learn new technologies and take them into the real world. This happened with ASP.NET, the source of his first two MVP awards, and recently with Silverlight, that he started to use from the v1.0 in some real projects.