* 1. 

Windows 7 Recipes

Taskbar Single Instance

Recipe version: 0.0.1

Last updated: 2/8/2011

Contents

Contents

[**Introduction** 3](#_Toc284449108)

[Objectives 3](#_Toc284449109)

[System Requirements 4](#_Toc284449110)

[Motivation 4](#_Toc284449111)

[The Solution 5](#_Toc284449112)

[Initializing a SingleInstanceManager 5](#_Toc284449113)

[Accepting Notifications from Other Application Instances 6](#_Toc284449114)

[Additional Configuration Options 8](#_Toc284449115)

[Recipe Structure 13](#_Toc284449125)

[Synchronization 13](#_Toc284449126)

[Communication 14](#_Toc284449127)

[Argument Notification 15](#_Toc284449128)

[Summary 16](#_Toc284449129)

# Introduction

* 1. The Windows 7 Taskbar includes many new features, and one of these new features is the Taskbar Jump-List, which allows users to perform various application related tasks without ever switching to the application window.
  2. For example, Windows Live Messenger allows users to change their status and Internet Explorer allows users to open a new tab and browse to frequently visited websites, without switching to the application.
  3.  
  4. **Figure**
  5. "Windows Live Messenger" and "Internet Explorer 8.0" Jump Lists

# Objectives

* 1. Windows 7 Taskbar tasks are simply command lines. When you click on a task, the associated command line is executed. In Word or Internet Explorer, for example, this usually launches a new instance of the application, but with Windows Live Messenger; the tasks seem to change the internal state of the currently running application rather than launch a new instance.
  2. This Taskbar Single Instance Recipe allows developers to easily develop applications that use "Messenger Like" tasks that change the state of the currently running instance, allowing it to react to incoming state-change notifications and act accordingly.

# System Requirements

* 1. To compile and run the recipe and samples the following items are required:
  + Microsoft Visual Studio 2008 SP1 or higher
  + Windows 7 – Note that only the samples require Windows 7. The Single Instance reusable library itself does not depend on Windows 7.

# Motivation

* 1. Adding a custom task to the Windows 7 Taskbar is easy, whether it is done directly via the Win32 API or using the [Windows API Code Pack](http://code.msdn.microsoft.com/WindowsAPICodePack) for managed code.
  2. C#

JumpListLink notepad = new JumpListLink {

Title = "Launch Notepad", Path = @"C:\Windows\notepad.exe" };

JumpListItem doc1 = new JumpListItem(@"doc1.txt");

JumpListItem doc2 = new JumpListItem(@"doc2.txt");

CustomCategory category = new CustomCategory("Special Items");

category.JumpListItems.Add(doc1);

category.JumpListItems.Add(doc2);

Taskbar.JumpList.UserTasks.Add(notepad);

Taskbar.JumpList.CustomCategories.Add(category);

* 1. VB.NET

Dim notepad As New JumpListLink() With { .Title = "Launch Notepad", \_

.Path = "C:\Windows\notepad.exe" }

Dim doc1 As New JumpListItem("doc1.txt")

Dim doc2 As New JumpListItem("doc2.txt")

Dim category As New CustomCategory("Special Items")

category.JumpListItems.Add(doc1)

category.JumpListItems.Add(doc2)

Taskbar.JumpList.UserTasks.Add(notepad)

Taskbar.JumpList.CustomCategories.Add(category)

**NOTE**: For more information on using the Windows 7 Taskbar APIs and for Native C++ code samples, please visit the [Windows 7 for Developers](http://windowsteamblog.com/blogs/developers/default.aspx) blog (<http://windowsteamblog.com/blogs/developers/default.aspx>).

Further information can also be found at [this](http://msdn.microsoft.com/en-us/magazine/dd942846.aspx) MSDN magazine article (<http://msdn.microsoft.com/en-us/magazine/dd942846.aspx>).

If all that is required is activating an ***IShellItem*** or ***IShellLink*** then the code above is sufficient. However, it should be noted that simply activating an ***IShellItem*** or an ***IShellLink*** from the Windows 7 Taskbar results in a **new process** being launched. If the shell link references our own executable, then activating it would result in the launching of a new instance of our process. Sometimes this is the required result (for example, when opening a new document in a new Notepad window), however we may want to use the taskbar tasks to affect our current instance's state (for example, by changing the status of a messenger application).

In order to accomplish this, the developer would usually need to use a pattern for checking if the application is already running and, if it is, it notify the currently executing application instance of the required task and close the new instance. This would usually involve some sort of synchronization object to check if an instance is already running (for example, a Mutex) and a communication mechanism to perform the notification itself. Basically, this is an implementation of a singleton pattern.

The code for accomplishing this task is usually quite similar for many application types, and is usually written over and over again. The goal of this recipe is to provide a simple, reusable code library for both native code and managed code developers that, on the one hand, can be easily used with default settings, but also can be easily customized and extended if required on the other hand.

# The Solution

The recipe's solution is composed of one reusable class library for native code and one reusable class library for managed code with extension assemblies for Windows Forms and WPF applications. Note that the class structure for both native and managed versions is very similar; therefore all explanations are valid for both versions except if otherwise specified. You can choose which library you want to work with according to your needs.

This recipe provides you with ability to pass parameters encoded in the task to an already running instance of your application. This way you can send “messages” to your running application and update the application’s internal state.

Basically the solution is a customizable implementation of the [Singleton pattern](http://en.wikipedia.org/wiki/Singleton_pattern), as described later.

## Initializing a SingleInstanceManager

The **SingleInstanceManager** class is the key class for using the recipe's code. In order to use the recipe, create an instance of the **SingleInstanceManager** using the static ***Initialize****()* method. This method accepts an object of type **SingleInstanceManagerSetup**, which can be used to customize the behavior of the **SingleInstanceManager**. At a minimum, this setup object should be given an **application ID** which will be used to identify the application as a single instance.

* 1. Native C++

SingleInstanceManagerSetup simSetup(L"MyApp");

g\_SIM = SingleInstanceManager::Initialize(&simSetup);

// Your application code here

delete g\_SIM;

* 1. C#

using (SingleInstanceManager manager =

SingleInstanceManager.Initialize(new SingleInstanceManager("MyApp")))

{

// Your application code here

}

* 1. VB.NET

Using manager As SingleInstanceManager = \_

SingleInstanceManager.Initialize(New SingleInstanceManager("MyApp"))

    ' Your application code here

End Using

As long as the **SingleInstanceManager** object is kept alive, no other instance of the application will be executed in parallel to the first instance. If and when you try to launch a second application instance, by default the process of the second application will exit before executing any of you application code. If you prefer you can set up the SingleInstanceManager to throw an exception instead of exiting the process (see section "Additional Configuration Options").

## Accepting Notifications from Other Application Instances

In order to receive incoming arguments from additional launched application instances, you need to specify an arguments handler when creating the SingleInstanceManagerSetup object.

* 1. Native C++

void ArgsHandler(LPCWSTR\* pArguments, DWORD dwLength, LPVOID pContext)

{

// Handle incoming arguments

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

// The NULL argument is the context passed to the handler (e.g. an object

// instance to call one of it's methods).

SingleInstanceManagerSetup simSetup(L"MyApp", ArgsHandler, NULL);

g\_SIM = SingleInstanceManager::Initialize(&simSetup);

// Your application code here

delete g\_SIM;

return 0;

}

* 1. C#

static void ArgsHandler(string[] args)

{

// Handle incoming arguments

}

static void Main()

{

SingleInstanceManagerSetup setup = new SingleInstanceManagerSetup("MyApp")

{

ArgumentsHandler = ArgsHandler

};

using (SingleInstanceManager manager =

SingleInstanceManager.Initialize(setup))

{

// Your application code here

}

}

* 1. VB.NET

Sub ArgsHandler(ByVal args As String())

    ' Handle incoming arguments

End Sub

Sub Main()

    Dim setup As New SingleInstanceManagerSetup("MyApp") With \_

{ .ArgumentsHandler = AddressOf ArgsHandler }

    Using manager As SingleInstanceManager = \_

SingleInstanceManager.Initialize(setup)

        ' Your application code here

    End Using

End Sub

By default, whenever trying to initialize a **SingleInstanceManager** object in a second process (as part of launching a new instance of an application) while the first application’s instance already exists, the command line arguments of the second instance will be sent to the first instance which will be notified of the arguments by the handler it has configured. The arguments sent to the first instance can be customized via the setup object (see section "Additional Configuration Options"). This setup object gives you full control over the how the parameters are being sent between applications, as well as other notifications.

## Additional Configuration Options

The **SingleInstanceManager** behavior can be customized via various properties of the setup object (**SingleInstanceManagerSetup**). These properties are outlined in this section. Note that in Native C++, these properties are accessible via "*Set…()*" and "*Get…()*" methods and are also settable via the class constructor.

### Application ID

The ***ApplicationId*** property is used to identify an application and is used as the key by which the application's single instance will be identified. Note that an application ID is unique to each logged on user (that is, if two application instances are running under different users on the same machine, application ID will not match).

The **ApplicationId** is the only mandatory property when constructing a SingleInstanceManagerSetup object.

### Arguments Handler

The ***ArgumentsHandler*** property is used to specify a handler method to receive notifications of incoming arguments (for an example, see section "Accepting Notifications from other Application Instances").

### Context (Native Only)

The ***Context*** property is available only in native code, and is used to specify a context object to be passed to the arguments handler.

* 1. Native C++

void ArgsHandler(LPCWSTR\* pArguments, DWORD dwLength, LPVOID pContext)

{

MyClass \*pClass = (MyClass \*) pContext;

pClass->SomeMethod(...); // Handle incoming arguments

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

MyClass \*pClass = new MyClass();

SingleInstanceManagerSetup simSetup(L"MyApp", ArgsHandler, pClass);

g\_SIM = SingleInstanceManager::Initialize(&simSetup);

// Your application code here

delete g\_SIM;

delete pClass;

return 0;

}

### Termination Option

The ***TerminationOption*** property allows the user to choose a course of action in cases where an application instance is already running and the current instance needs to be closed (after notifying the original instance of the arguments). The enum allows two possible values:

1. **Exit** (default) – Exit the current process without returning to the user code. The managed code calls "*System.Environment.Exit()*" and the native code calls "*ExitProcess()*", and both use the specified exit code in the *ExitCode* property.
2. **Throw** –Throws an exception indicating that another application instance is already running. The user is then able to catch the exception and exit gracefully.

#### Exit Code

The *ExitCode* property allows specifying the exit code to be used in case the ***TerminationOption*** property is set to "Exit" and an application instance is already running. The default value is 0.

### Arguments Provider

The ***ArgumentsProvider*** property allows you to override the default response for the arguments passed to the first application instance. By default, the arguments passed are the command line arguments of the current process, which uses "***Environment****.****GetCommandLineArgs****()*" for managed code and "***GetCommandLineW****()*" and "***CommandLineToArgvW****()*" for native code.

### Delivery Strategy Factory

The *Factory* property allows the user to customize the argument delivery strategy that is used by the **SingleInstanceManager**. By default, the "***RemotingStrategyFactory***" is used for managed code and "***NamedPipeStrategyFactory***" is used for native code. These factory objects produce "***RemotingStrategy***" strategy objects and "***NamedPipeStrategy***" strategy objects respectively.

The delivery strategy factory is the class used by the SingleInstanceManager to both:

* **Notify** the original application instance of the command line arguments, if this application instance is not the first one
* **Receive** notification of incoming command line arguments from other application instances if the current instance is the first one

The specific strategy implementation contains all the logic and code that is required to enable the communication between the two processes. If the default supplied strategy does not meet the user's needs, the user can supply another implementation that can be easily plugged in.

### Arguments Handler Invoker

The ***ArgumentsHandlerInvoker*** is the object used by the **SingleInstanceManager** to invoke the registered argument handler. By default, the handler is invoked on the thread pool in managed code and directly on the incoming thread in native code, however two specializations are available for Windows Forms and WPF applications:

1. ***SingleInstance.WindowsForms.WindowsFormsInvoker*** in assembly *SingleInstance.WindowsForms* –Invokes the UI thread handler of the control the invoker is initialized with.
   * + - 1. C#

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

MainForm mainForm = new MainForm();

using (SingleInstanceManager manager =

SingleInstanceManager.Initialize(

GetSingleInstanceManagerSetup(mainForm)))

{

Application.Run(mainForm);

}

}

private static SingleInstanceManagerSetup

GetSingleInstanceManagerSetup(MainForm mainForm)

{

return new SingleInstanceManagerSetup("WinFormSample")

{

ArgumentsHandler = mainForm.ProcessCommandLineArgs,

ArgumentsHandlerInvoker = new WindowsFormsInvoker(mainForm),

DelivaryFailureNotification = ex =>

MessageBox.Show(ex.Message, "An error occured")

};

}

1. ***SingleInstance.Wpf.ApplicationDispatcherInvoker*** in *assembly SingleInstance.Wpf* –Invokes the handler on the WPF application dispatcher thread.
   * + - 1. C#

[STAThread]

static void Main()

{

using (SingleInstanceManager manager =

SingleInstanceManager.Initialize(

GetSingleInstanceManagerSetup()))

{

App app = new App();

app.InitializeComponent();

app.Run();

}

}

private static SingleInstanceManagerSetup

GetSingleInstanceManagerSetup()

{

return new SingleInstanceManagerSetup("WpfSample")

{

ArgumentsHandler = args =>

((App)Application.Current).ProcessCommandLineArgs(args),

ArgumentsHandlerInvoker = new ApplicationDispatcherInvoker(),

DelivaryFailureNotification = ex =>

MessageBox.Show(ex.Message, "An error occured")

};

}

* + - * 1. VB.NET

<STAThread()> \_

Sub Main()

Using manager As SingleInstanceManager = \_

SingleInstanceManager.Initialize( \_

GetSingleInstanceManagerSetup())

Dim app As New App()

app.InitializeComponent()

app.Run()

End Using

End Sub

Private Function GetSingleInstanceManagerSetup() \_

As SingleInstanceManagerSetup

Return New SingleInstanceManagerSetup("WpfSampleVB") With \_

{ \_

.ArgumentsHandler = Function(args As String()) \_

DirectCast(Application.Current, \_

App).ProcessCommandLineArgs(args), \_

.ArgumentsHandlerInvoker = New ApplicationDispatcherInvoker(), \_

.DelivaryFailureNotification = Function(ex) \_

MessageBox.Show(ex.Message, "An error occured") \_

}

End Function

* + - * 1. Note that usually the WPF application generates a "*Main()*" method automatically. In order to write your own custom "*Main()*" method in a WPF application, C# developers should mark the "App.xaml" file as a "Page" in the "Build Action" property and then write a "*Main()*" method. VB.NET developers should simply write a "*Main()*" method and choose the appropriate startup object under the "Application" property page in the project settings.
        2. In addition, the *SingleInstanceManager* object can be initialized in various places except a custom "*Main()*" method (for example, the "*Initialized"* event of the main window).
  1. Usage of these specialized invokers is recommended when the callback should be processed on the UI thread.

### Instance Notification Option

The ***InstanceNotificationOption*** property is used to determine the behavior when the first application instance is launched under elevated security rights (that is, run as administrator), and the second instance is not an administrator. This enum has two options:

1. **NotifyAnyway** (default) – Notifies the original application instance anyway.
2. **NotifyOnlyIfAdmin** – Notifies the first application instance only if the second application instance is also run as an administrator.

***Note*** *that this setting has an effect only if the first application instance is run as an administrator. This is a critical point, as improper implementation here can expose your application to attacks*

### Delivery Failure Notification

The **DeliveryFailureNotification** is used to provide the SingleInstanceManager with a callback method if the delivery of the arguments from the second process to the first one was unsuccessful.

# Recipe Structure

The following section outlines the internal structure of this recipe. This is optional reading in case you are interested in how the recipe works behind the scenes.

The recipe is composed of three main parts:

* The synchronization mechanism
* The communication mechanism
* The arguments notification

## Synchronization

A named Mutex kernel object is used for checking if the application instance is the first (original) instance or if an application instance is already running. This check is performed by the *SingleInstanceManager* object and is not customizable without refactoring the recipe's code.

* 1. Native C++

BOOL SingleInstanceManager::IsFirstApplicationInstance()

{

\_ASSERT\_EXPR(m\_hMutex == NULL,

L"Mutex has already been initialized by this instance!");

m\_hMutex = CreateMutex(NULL, TRUE, m\_lpApplicationId);

if (m\_hMutex == NULL)

throw SingleInstanceManagerException(

L"Error creating mutex used for single instance ensurence!");

return (GetLastError() != ERROR\_ALREADY\_EXISTS);

}

* 1. C#

private bool IsFirstApplicationInstance()

{

Debug.Assert(singleInstanceMutex == null);

bool createdNew;

singleInstanceMutex = new Mutex(true, applicationId, out createdNew);

return createdNew;

}

The return value from the "*IsFirstApplicationInstance()*" method is used to determine if the current application instance should run and listen for incoming argument notification, of if it should notify an already running application instance of the command line arguments and exit.

## Communication

The communication mechanism used by the **SingleInstanceManager** is fully implemented by an instance of an object derived from the abstract ***ArgumentDeliveryStrategy****,* which allows two modes of communication:

* Receiving incoming arguments from any second application instance, if running as the first application instance
* Notifying another application instance of arguments
  1. Native C++

class SINGLEINSTANCE\_API ArgumentsDeliveryStrategy abstract

{

protected:

virtual void OnInitializeFirstInstance(LPCWSTR lpApplicationId) = 0;

virtual void OnDeliverArgumentsToFirstInstance(LPCWSTR lpApplicationId,

LPWSTR \*pArgs, DWORD dwLength) = 0;

virtual void OnCleanup();

void NotifyArgumentsReceived(LPCWSTR \*pArgs, DWORD dwLength,

BOOL bIsRemoteAdmin);

};

* 1. C#

public abstract class ArgumentsDeliveryStrategy

{

protected abstract void OnInitializeFirstInstance(string applicationId);

protected abstract void OnDeliverArgumentsToFirstInstance(string applicationId,

string[] args);

protected virtual void OnCleanup() { }

protected void NotifyArgumentsReceived(string[] args, bool remoteIsAdmin)

{

// Implementation not shown for brevity

// …

}

// Internal and private methods not shown for brevity

}

A default out-of-the-box implementation is provided for both managed and native code:

* ***NamedPipeStrategy*** (native code) – A communication mechanism that uses named pipes for its implementation
* ***RemotingStrategy*** (managed code) – A communication mechanism that uses .NET Remoting via named pipes (IPC) for its implementation

A developer can write an implementation of the above abstract class, and specify a factory object that creates it (derived from *DeliveryStrategyFactory*) when initializing a *SingleInstanceManagerSetup* object. This allows for full customization of the communication mechanism without refactoring the code.

## Argument Notification

Once an incoming argument notification is received by the first running application instance, a notification is raised to the developer if a notification handler was specified upon initialization of the *SingleInstanceManager*. This notification is invoked by an instance of an object implementing *IArgumentsHandlerInvoker* in managed code or derived from the *ArgumentsHandlerInvoker* abstract class in native code.

* 1. Native C++

class SINGLEINSTANCE\_API ArgumentsHandlerInvoker abstract

{

friend SingleInstanceManager;

protected:

virtual void Invoke(PFNARGSHANDLER pHandlerToInvoke, LPCWSTR \*pArgs,

DWORD dwLength, LPVOID pContext) = 0;

};

* 1. C#

public interface IArgumentsHandlerInvoker

{

void Invoke(Action<string[]> handlerToInvoke, string[] args);

}

Default out-of-the-box invokers are provided for native and managed code:

1. ***PassThroughInvoker*** (native code) –Invokes the handler on the incoming thread from the *ArgumentDeliveryStrategy* instance.
   * + - 1. Native C++

void PassThroughInvoker::Invoke(PFNARGSHANDLER pHandlerToInvoke,

LPCWSTR \*pArgs, DWORD dwLength, LPVOID pContext)

{

pHandlerToInvoke(pArgs, dwLength, pContext);

}

1. ***ThreadPoolInvoker*** (managed code) – Invokes the handler on a thread pool thread.
   1. C#

public void Invoke(Action<string[]> handlerToInvoke, string[] args)

{

ThreadPool.QueueUserWorkItem(

obj => handlerToInvoke((string[])obj), args);

}

As described in a previous section, two additional customized invokers are available for Windows Forms and WPF application (see "Additional Configuration Options" section for further information).

Note that the developer can implement his own invoker and specify an instance of it when constructing the *SingleInstanceManagerSetup* object. This allows further customization for specific application types.

# Summary

* 1. This Windows 7 Recipe allows developers to easily develop applications that use "Messenger Like" tasks to change the state of the currently running instance, allowing it to react to incoming state change notifications and act accordingly. It features implementations for both native and managed code developers, and it allows easy customization of many aspects of its operation.
  2. Using this recipe in your application is as easy as one-two-three – add a dependency to the code library, initialize a SingleInstanceManager (usually most defaults will do), and you're good to go!