Introduction to background tasks

Guidelines for developers

July 16th, 2012

Abstract

Windows 8 provides Windows Store apps with the ability to run app code, even when the app is suspended, by using background tasks. This paper describes the programming model to create background tasks, resource management policies for background tasks, and built-in user controls that allow the user to control per-app background task activity by using lock screen personalization. Some familiarity with Windows Store app development is assumed.

This information applies to the following operating system:   
 Windows 8

References and resources discussed here are listed at the end of this paper.

The current version of this paper is maintained on the web at:   
 <http://go.microsoft.com/fwlink/?LinkID=227329&clcid=0x409>

**Disclaimer**: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet website references, may change without notice. Some information relates to pre-released product which may be substantially modified before it’s commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2012 Microsoft. All rights reserved.

Description: Description: Description: Description: C:\Users\lisand\Desktop\logos\for_uploading\dep_MicrosoftLogotype.png

Contents

[Introduction 3](#_Toc330377350)

[Appropriate scenarios for background tasks 3](#_Toc330377351)

[Terms and definitions 4](#_Toc330377352)

[Background task basics 4](#_Toc330377353)

[Background task infrastructure 6](#_Toc330377354)

[Background tasks and the lock screen 6](#_Toc330377355)

[Background tasks and the Windows Store app lifecycle 8](#_Toc330377356)

[Background task implementation concepts 9](#_Toc330377357)

[Condition latching behavior 9](#_Toc330377358)

[Background task progress reporting 9](#_Toc330377359)

[Background task completion notification 9](#_Toc330377360)

[Background task cancellation 10](#_Toc330377361)

[Background task enumeration and reassociating handlers 10](#_Toc330377362)

[App updates or servicing 11](#_Toc330377363)

[Background task host executable 11](#_Toc330377364)

[Background task sample code 12](#_Toc330377365)

[Background task class 12](#_Toc330377366)

[Adding a background task to a solution 14](#_Toc330377367)

[Background task sample manifest 15](#_Toc330377368)

[Debugging background task activation 17](#_Toc330377369)

[Background task resource management 19](#_Toc330377370)

[CPU resource constraints 19](#_Toc330377371)

[Network resource constraints 20](#_Toc330377372)

[Critical background tasks 20](#_Toc330377373)

[Global pool 21](#_Toc330377374)

[What is CPU usage time? 21](#_Toc330377375)

[Idle background tasks 22](#_Toc330377376)

[Threading model for background tasks hosted in the app 22](#_Toc330377377)

[Decoupling the background task from the app 22](#_Toc330377378)

[Sharing state between the background task and the foreground app 23](#_Toc330377379)

[Background task best practices 23](#_Toc330377380)

[Resources 23](#_Toc330377381)

# Introduction

Windows 8 introduces a new model of app behavior. Windows Store apps in Windows 8 are full screen and the user is expected to interact only with the app that is in the foreground. The foreground app is assumed to be the most important to the user, so this app receives all the resources of the system. When an app is not in the foreground, it is suspended, and cannot run any code. A suspended app remains suspended until the user resumes it by bringing the app back to the foreground. With this model of app behavior, the user experience is never impacted by lags or delays caused by the execution of unimportant background apps. In addition, reducing unnecessary background activity optimizes battery life on a variety of form factors. The time taken to resume a suspended app is negligible and would appear to be almost instantaneous to most users.

Windows 8 provides a number of features to make an app update content even when the app is suspended:

* Windows push notifications can be used to keep the app tile fresh and up-to-date.
* Playback Manager can be used to play audio in the background.
* The background transfer API can be used to download and upload files in the background.
* File share contracts can be used to share data between apps.

Push notifications and the background transfer API are optimized for system performance and longer device battery life, so it’s best to use these features whenever possible. If a suspended app must run its own code to do other kinds of work, Windows 8 provides apps with the ability to create background tasks.

## Appropriate scenarios for background tasks

Allowing apps to run code in the background when they are suspended is a powerful feature and is designed primarily for the real-time class of apps such as mail, VOIP, and chat apps. The background task execution environment is a restricted resource-managed environment, and background tasks only receive a limited amount of system resources. Background tasks should be used for small work items that have no interaction with the user and only provide limited service. Long running or expensive workloads in the background will deplete the user’s battery and are not an appropriate use for background tasks.

Scenarios that are appropriate for background tasks include downloading mail in the background, or showing a toast notification for an incoming VOIP call or a chat message, or reacting to a change in system condition (for example, UserAway) and updating the server with this information. Scenarios that are not appropriate for background tasks are running SETI type workloads in the background, or anything that requires user interaction through displaying UI or audio.

## Terms and definitions

This paper uses the following terms:

Background task

A class or JavaScript page implemented by the app to provide functionality even if the app is not in the foreground.

Background trigger

A system-defined event that an app can associate with a background task. When the trigger is fired by the system, an app background task that is associated with the trigger is launched.

Background condition

A set of zero or more conditions that need to be satisfied before the background task can run.

BackgroundTaskHost.exe

A system-provided host executable to run the background task.

EntryPoint

The name of the C# or C++ class that implements the background task.

Executable

The name of the executable that hosts the background task class.

Foreground app

The app that the user is actively interacting with.

Lock screen

This is the first screen shown following a Windows 8 boot, resuming from sleep, or locking your PC. It presents a user-customizable surface that both conveys information and protects against accidental logon attempts.

Start page

The name of the JavaScript page that implements the background task.

# Background task basics

A background task is implemented by the app to provide functionality even when the app is suspended or not running. The background task will have one external event (a trigger), which indicates when the task should run, and a set of zero or more conditions, which must be true for the background task to run even if the trigger fires. Examples of triggers are time, push notifications, or system events.

A Windows Store app registers its background tasks with the background task infrastructure by using the **BackgroundTaskBuilder** class. The background task is implemented as a class that implements the **IBackgroundTask** interface. This class name is specified in the **TaskEntryPoint** property of the **BackgroundTaskBuilder** class. The background task class is hosted in an in-proc server DLL, which can be loaded in an app executable, as part of the app package, or in a system-provided host executable.

The background task can also be implemented as a JavaScript web worker. The JavaScript worker is included in the app as a JavaScript file, and the name of the JavaScript file is specified in the **TaskEntryPoint** property of the **BackgroundTaskBuilder** class. The JavaScript worker runs in the system-provided host process, similar to a web worker.

A background task must have exactly one trigger that describes the trigger event to launch the background task. The trigger is specified with the **SetTrigger** method of the **BackgroundTaskBuilder** class. Table 1 shows a list of all available trigger events.

Table 1 – Background task trigger events

|  |  |  |
| --- | --- | --- |
| **Background task trigger type** | **Trigger event** | **When the background task is triggered** |
| **ControlChannelTrigger** | ControlChannelTrigger | On incoming messages on the control channel. |
| **MaintenanceTrigger** | MaintenanceTrigger | It’s time for maintenance background tasks. |
| **PushNotificationTrigger** | PushNotificationTrigger | A raw notification arrives on the WNS channel. |
| **SystemEventTrigger** | InternetAvailable | The Internet becomes available. |
| **SystemEventTrigger** | LockScreenApplicationAdded | An app tile is added to the lock screen. |
| **SystemEventTrigger** | LockScreenApplicationRemoved | An app tile is removed from the lock screen. |
| **SystemEventTrigger** | ControlChannelReset | A network channel is reset. |
| **SystemEventTrigger** | NetworkStateChange | A network change such as a change in cost or connectivity occurs. |
| **SystemEventTrigger** | OnlineIdConnectedStateChange | Online ID associated with the account changes. |
| **SystemEventTrigger** | ServicingComplete | The system has finished updating an application. |
| **SystemEventTrigger** | SessionConnected | The session is connected. |
| **SystemEventTrigger** | SessionDisconnected | The session is disconnected. |
| **SystemEventTrigger** | SmsReceived | A new SMS message is received by an installed mobile broadband device. |
| **SystemEventTrigger** | TimeZoneChange | The time zone changes on the device (for example, when the system adjusts the clock for daylight saving time). |
| **SystemEventTrigger** | UserAway | The user becomes absent. |
| **SystemEventTrigger** | UserPresent | The user becomes present. |
| **TimeTrigger** | TimeTrigger | A time event occurs. |

A background task can optionally have zero or more conditions that must be true for the task to launch in addition to the trigger firing. Background conditions are used by an app to describe the set of conditions required to perform meaningful work.

For example, if the background task requires the Internet but there is not currently an Internet connection, an **InternetAvailable** condition can be used to tell the system to wait for the Internet connection to be restored before launching the background task. Background conditions are specified with the **AddCondition** method of the **BackgroundTaskBuilder** class. Table 2 shows a list of conditions.

Table 2 – Background task conditions

|  |  |
| --- | --- |
| Background task condition | The condition that must be satisfied |
| **InternetAvailable** | The Internet must be available. |
| **InternetNotAvailable** | The Internet must be unavailable. |
| **SessionConnected** | The session must be connected. |
| **SessionDisconnected** | The session must be disconnected. |
| **UserNotPresent** | The user must be away. |
| **UserPresent** | The user must be present. |

## Background task infrastructure

When the trigger is fired, the background task infrastructure launches the class associated with the trigger regardless of the state of the app. The activation of the background task does not involve any UI, and it does not bring the Windows Store app to the foreground.

Figure 1 shows the process of registering and launching a background task. When an app uses the **BackgroundTaskBuilder** class to register a background task, the BackgroundTaskBuilder code gives the trigger details to the trigger implementations. It then registers the app class using the trigger with the system infrastructure. When the trigger is fired, the system infrastructure activates the class within the app container.

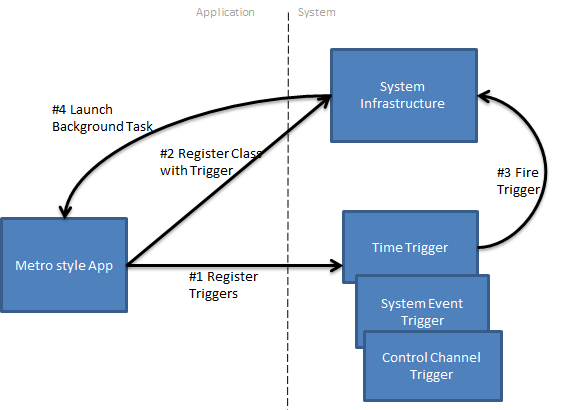


Figure 1 – Registering and launching a background task

## Background tasks and the lock screen

A good app on the lock screen is one that provides users with useful, at-a-glance information that is pertinent to the user, even when they’re not actively using their PC. Users should not have to dwell on the lock screen to get information from a lock screen app, nor should they need additional context from the app tile (such as images) to understand what is conveyed on the lock screen. Examples of apps that fit this mold are mail apps that show a count of unread email messages, a calendar app that shows upcoming appointments in the detailed status slot, or a messaging app that shows how many messages a user has missed.

Most importantly, though, a good lock screen app relays real-time information to a user, such as a communication app informing the user when they’ve received a message or when a call is incoming. This is particularly important because, when a user places an app on their lock screen, they’re indicating that that app is especially important to them. It should always reflect real-time information so the user has confidence that they can rely upon it when glancing at their screen for updates. As such, every app that is on the lock screen must implement a background task of either the Control Channel, Timer, or Push Notification type. These tasks will allow an app to always reflect accurate, up-to-date information in real-time. Going hand-in-hand with this, any app that needs this type of background task will need to be on the lock screen.

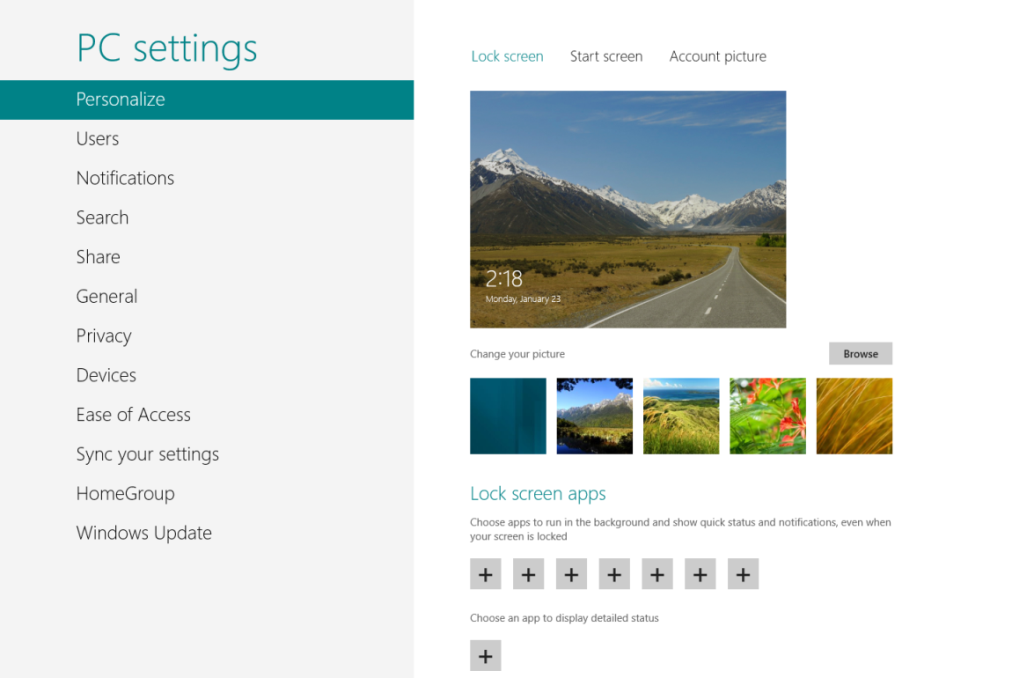


Figure 2 – Lock screen personalization UI and lock screen capable apps

As previously mentioned, access to specific types of triggers is restricted to only apps on the lock screen. If an app that is not on the lock screen tries to access such a trigger, its background task will not be triggered.

Table 3 – Background task trigger types and lock screen requirement

|  |  |
| --- | --- |
| Background task trigger type | Requires app on the lock screen |
| **TimeTrigger** | **Yes** |
| **PushNotificationTrigger** | **Yes** |
| **ControlChannelTrigger** | **Yes** |
| **SystemTrigger**\* | **No** |
| **MaintenanceTrigger** | **No** |

\*Some SystemTriggers, however, do need the app to be on the lock screen: **SessionConnected**, **UserPresent**, **UserAway** and **ControlChannelReset**. If you use these triggers without being on the lock screen, the **BackgroundTaskBuilder.Register()** call will fail with “**access denied**”.

A user can always remove the app from the lock screen to either make room for another app or to conserve resources if the user notices that an app is taking up too many resources in the Task Manager. An app can register to be notified when it is added or removed from the lock screen by registering a background task with the **LockScreenApplicationAdded** or **LockScreenApplicationRemoved** triggers provided in the **SystemTrigger** class.

An app can programmatically add itself to the lock screen by using the **BackgroundExecutionManager** class. For more information about how to programmatically add an application to the lock screen, lock screen app badge requirements, and lock screen best practices, see “Resources,” at the end of this paper.

## Background tasks and the Windows Store app lifecycle

When a background task is launched, an app can be in any of the following states: Running, Terminated, or Suspended. If an app is in the Running state, it is the only app in the foreground. If an app is not running, it is in either the Suspended or Terminated state. An app in the Suspended state is resident in memory, but its work is not being scheduled by the system. An app in the Terminated state is not in memory, either because it was never launched or because it was in the Suspended state and was terminated by the system.

The background task executes in either the system-provided host executable or the app process. For more information about the executable host rules, see “Background task host executable,” section later in this paper.

If a background task executes in the system-provided BackgroundTaskHost.exe, it is launched independently of the state of the app; it is not necessary for the system to change the app’s state. This option is preferable because launching the task within BackgroundTaskHost.exe is faster, uses fewer resources, and overall, provides better performance than launching the background task within the app. BackgroundTaskHost.exe is launched within the same app container as the app, and it will exit when it is no longer needed by the system.

If a background task executes within the app process, the background task infrastructure might need to change the state of the app:

* **Running**: If the app is running, it is already in the foreground and the background task is simply launched within the app.
* **Suspended**: If the app is suspended, most of its threads are unfrozen and the background task is launched. For details, see “Threading model for background tasks hosted in the app,” later in this paper.
* **Terminated**: If the app is terminated, it is not running at the time of the trigger so the app is launched and the background task is run. The app activation does not involve UI, and it does not bring the app to the foreground.

# Background task implementation concepts

Background tasks have one trigger and zero or more conditions associated with a task. All background task registrations are **persistent** (except for Control Channel background tasks); that is, the app doesn’t have to be running for the background task to be triggered. For example, if a background task should run 6 hours from now to download RSS feeds; the app registers the background task using **TimeTrigger** with the appropriate value. When the time arrives, the background task is launched regardless of whether the app is running or the computer has been restarted since the task was registered. The exception to this is Control Channel background tasks.

## Condition latching behavior

When a trigger is fired, the background task infrastructure checks whether all of the conditions are satisfied. If the conditions are not satisfied, the infrastructure waits until all the conditions are satisfied before executing the background task. This behavior is called *latching* because the background task infrastructure latches the trigger to a fired state.

If a condition changes value while a trigger is latched, the background task infrastructure checks again to determine whether the conditions are satisfied for the background task. When all of the conditions are satisfied, it launches the task. For example, a background task that runs one hour from now with an **InternetAvailable** condition will run after the hour expires only if the Internet is available. When the time trigger fires, the background task infrastructure latches the trigger, but it does not run the task until the condition **InternetAvailable** is true.

## Background task progress reporting

The background task can use the progress handler to report progress to the app when the app is running in the foreground. The progress reported from the background task can be used to display UI or a progress bar in the foreground app.

The **IBackgroundTaskInstance** (**WebUIBackgroundTaskInstance** in JavaScript) that is passed to the **Run** method is used to communicate progress to the foreground app. The interface has an optional **Progress** property that can be updated by the background task. The app must register a progress handler to receive progress updates from the background task when the app is in the foreground. The progress handler in the app does not receive the updates from the background task if the app is terminated. If the app was suspended when the task sent the progress, it will receive it next time the app is resumed. For an example, see “Background task sample code,” later in this paper.

## Background task completion notification

The foreground app can also register a completion handler to be notified when the background task completes. The completion status or any exception thrown in the **Run** method of the background task is passed to the completion handler in the foreground app as part of the **BackgroundTaskCompletedEventArgs** input parameter. If the app was suspended when the task completed, it will receive the completion notification next time the app is resumed. If the app was in the Terminated state, it does not receive the completion notification. If the background task needs to preserve the information that it ran successfully, it must persist the information with ApplicationData or other means, such as a file that the app can read when it comes back to the Running state. For an example, see “Background task sample code,” later in this paper.

## Background task cancellation

Background tasks can be canceled by the system in certain conditions. To receive cancellation notifications, the app’s background tasks should register a cancellation handler by using the **IBackgroundTaskInstance**. When the system cancels a background task, it provides a reason in the **BackgroundTaskCancellationReason** parameter of the **BackgroundTaskCanceledEventHandler**.

If the background task does not respond to the cancellation notification and does not return from its background task within 5 seconds, the app is terminated. To ensure a graceful exit and ensure consistent app state, apps are encouraged to register a cancellation handler and save state and exit the background task in response to a cancellation request. For an example, see “Background task sample code,” later in this paper.

## Background task enumeration and reassociating handlers

When a Windows Store app is launched from the Terminated state, it can enumerate all background tasks it has registered with the system using the **BackgroundTaskRegistration.AllTasks** method. Because the app has just been launched, it must reassociate its progress and completion handlers with the background tasks. Even if the app has previously registered those associations, they are no longer valid because the app was terminated and removed from memory. Only the progress and completion handlers must be reassociated. The background task registrations themselves are persistent.

The progress and completion handlers are event delegates or functions and are not defined in the manifest as are the class names (**EntryPoint**), which is why they need to be associated after the app has been launched.

If the app was resumed from the Suspended state then it does not have to reassociate its progress and completion handlers for its background tasks. They remain valid because a suspended app remains in memory.

The following sample code shows how to enumerate tasks and reassociate handlers.

##### C#

foreach (var task in BackgroundTaskRegistration.AllTasks)

{

task.Value.Progress += new BackgroundTaskProgressEventHandler(OnProgress);

task.Value.Completed += new BackgroundTaskCompletedEventHandler(OnCompleted);

}

##### JavaScript

var iter = Windows.ApplicationModel.Background.BackgroundTaskRegistration.allTasks.first();

var hascur = iter.hasCurrent;

while (hascur)

{

var cur = iter.current.value

cur.addEventListener("progress", onProgress);

cur.addEventListener("completed", onCompleted);

hascur = iter.moveNext();

}

## App updates or servicing

Background task registrations persist across app updates. If an app is updated, its registrations continue to be valid and will be triggered. The app can also verify this by enumerating its background tasks as shown above. The app developer must ensure that the background task **EntryPoint** still exists in the updated version of the class; otherwise, the background task will error out during activation. To delete a previous background task registration upon update, an app can register a background task for a **ServicingComplete** trigger to be notified when the app is updated, and unregister background tasks that are no longer valid.

When an app is uninstalled, all its background task registrations are deleted from the system.

## Background task host executable

For C# and C++ apps, the background task is implemented in a class library, and this class library can run within the main app or in the system-provided host. Different background trigger types (task types) have different restrictions on where the background task can be hosted. By default when no Executable attribute is specified, the background task runs in the system-provided host. The app cannot specify the Executable attribute if it must load in the system-provided host. Only background tasks that contain PushNotificationTrigger or ControlChannelTrigger task types can specify an Executable attribute.

For JavaScript background tasks, the Executable attribute should never be specified because JavaScript apps are loaded in a web worker within the default system-provided host. In the case of a mixed app (JavaScript app with C# background task), the same rules as a regular C# background task apply.

These are the rules for the various background task trigger types for C# and C++ apps:

Table 4 – Background task trigger types and host Executable

|  |  |
| --- | --- |
| Background task trigger type | Host executable |
| **TimeTrigger** | Cannot be specified (default system host) |
| **SystemTrigger** | Cannot be specified (default system host) |
| **MaintenanceTrigger** | Cannot be specified (default system host) |
| **PushNotificationTrigger** | Application provided exe **OR** Not specified (default system host) |
| **ControlChannelTrigger** | Application provided exe |

# Background task sample code

The following sample code shows background task registration for a time trigger that fires in 15 minutes if the Internet is available. Note that the time trigger only accepts values greater than or equal to 15; smaller values fail during Register.

##### C#

private void RegisterBackgroundTasks()

{

BackgroundTaskBuilder builder = new BackgroundTaskBuilder();

// Friendly string name identifying the background task

builder.Name = "BackgroundTestClass";

// Class name

builder.TaskEntryPoint = "BackgroundTaskLibrary.TestClass";

IBackgroundTrigger trigger = new TimeTrigger(15, true);

builder.SetTrigger(trigger);

IBackgroundCondition condition = new SystemCondition(SystemConditionType.InternetAvailable);

builder.AddCondition(condition);

IBackgroundTaskRegistration task = builder.Register();

task.Progress += new BackgroundTaskProgressEventHandler(task\_Progress);

task.Completed += new BackgroundTaskCompletedEventHandler(task\_Completed);

}

##### JavaScript

function RegisterSampleBackgroundTaskWithCondition()

{

var builder = new Windows.ApplicationModel.Background.BackgroundTaskBuilder();

builder.name = "BackgroundTestWorker";

builder.taskEntryPoint = "BackgroundTestWorker.js";

var myTrigger = new Windows.ApplicationModel.Background.TimeTrigger(15, true);

builder.setTrigger(myTrigger);

var condition = new Windows.ApplicationModel.Background.SystemCondition(Windows.ApplicationModel.Background.SystemConditionType.internetAvailable);

builder.addCondition(condition);

var task = builder.register();

task.addEventListener("progress", task\_Progress);

task.addEventListener("completed", task\_Completed);

}

## Background task class

##### C#

**BackgroundTaskLibrary.TestClass** implements the **IBackgroundTask** interface that has a **Run** method. The **Run** method is where the actual background task code exists. The class that implements the **IBackgroundTask** interface must be sealed. The **Run** method receives **IBackgroundTaskInstance** as an input parameter that contains information about the task (trigger and conditions).

namespace BackgroundTaskLibrary

{

public sealed class TestClass:IBackgroundTask

{

private int globalcount;

void IBackgroundTask.Run(IBackgroundTaskInstance taskInstance)

{

globalcount = 0;

for (int i = 0; i < 100000; ++i)

{

Interlocked.Increment(ref globalcount);

taskInstance.Progress = (uint)globalcount;

}

}

}

}

By default Windows 8 assumes that the background task is done after the **Run** method returns. However, if the **Run** method initiates asynchronous operations, it might return before all work has completed because there could still be inflight asynchronous operations. To facilitate asynchronous work associated with a background task, applications should use the **BackgroundTaskDeferral** class.

The **Run** method should request a **BackgroundTaskDeferral** if it has any asynchronous operations in the **Run** method and then call the **Complete** method of its **BackgroundTaskDeferral** instance when it is has completed all asynchronous operations. The **Complete** call should be the last operation performed in response to the **Run** method being invoked so as to ensure that all the work initiated by and related to the **Run** method completes before the deferral is completed.

public sealed class TestClass:IBackgroundTask

{

    async void Run(IBackgroundTaskInstance taskInstance)

    {

        BackgroundTaskDeferral deferral = taskInstance.GetDeferral();

        try

        {

            await *SomeOperationAsync*();

            await *SomeOtherOperationAsync*();

            CompleteSomeSyncOperation();

        }

        finally { deferral.Complete(); }

    }

}

If using WinRT asynchronous operation instances directly, the completion handler of the asynchronous function can call **Complete** after all work has completed.

public sealed class BackgroundClass:IBackgroundTask

{

    public void Run(IBackgroundTaskInstance taskInstance)

    {

        BackgroundTaskDeferral def = taskInstance.GetDeferral();

        IAsyncAction op1 = Something.DoAsync();

        op1.Completed = new AsyncActionCompletedHandler(

            (IAsyncAction act, AsyncStatus stat) =>

            {

                def.Complete();

            });

    }

}

##### JavaScript

BackgroundTestWorker.js is a JavaScript file included with the app. The script in this file runs when the background task is triggered. The script can access the background task’s properties and events from the WebUIBackgroundTaskInstance object. A JavaScript background task must call **close()** after it has completed its work so that the task can be shut down.

Without a **close()** statement at the end, the background task infrastructure assumes that the task is still running and keeps the JavaScript host active. This consumes memory and other resources unnecessarily.

(function()

{

"use strict";

var globalCount= 0,

backgroundTask = Windows.UI.WebUIBackgroundTaskInstance.current;

for (int i = 0; i < 10; i++)

{

globalCount += 10;

backgroundTask.progress = globalCount;

}

backgroundTask.succeeded = true;

// close must be called to indicate that the task has

// completed

close();

})();

## Adding a background task to a solution

To add a C# class library that implements the **IBackgroundTask** interface to your project, follow these steps:

1. Add a Windows Store app class library project to the solution.
2. Add a reference to the newly added class library project in the main Windows Store app project that registers the background tasks.
3. Ensure that the output type of the class library is **Winmd** and not DLL.
4. Add the required manifest entries for background tasks to the Windows Store app project (not the class library). For details, see “Background task sample manifest,” later in this paper.
5. Seal the class that implements the **IBackgroundTask**.

To add a JavaScript background worker to your project, add a new dedicated worker file.

## Background task sample manifest

Windows Store apps must be manifested to use background task extension contracts. Every Windows Store app has an associated manifest that is used by the app to indicate that it is using the background task extension contract. An app’s manifest can contain multiple background task extensions with different class names (**EntryPoint**) or JavaScript files (**StartPage**), executable, or task types within the same app. The background task extension must specify the triggers (task type) used by the class or JavaScript file. If the trigger type is not specified correctly, the **Register** operation fails.

The manifest also includes the LockScreen elements in the VisualElements section. This is required to ensure that the app can be added to the lock screen. For UI best practices and other information about lock screen badges, see “Resources,” at the end of this paper.

These are the task types that need to be specified in the manifest for the different types of background task triggers.

Table 5 – Background task trigger types and manifest task type

|  |  |
| --- | --- |
| Background task trigger type | Manifest task type |
| **TimeTrigger** | timer |
| **SystemTrigger** | systemEvent |
| **MaintenanceTrigger** | systemEvent |
| **PushNotificationTrigger** | pushNotification |
| **ControlChannelTrigger** | controlChannel |

The following sample manifest is for an app that uses the time trigger and the system event trigger.

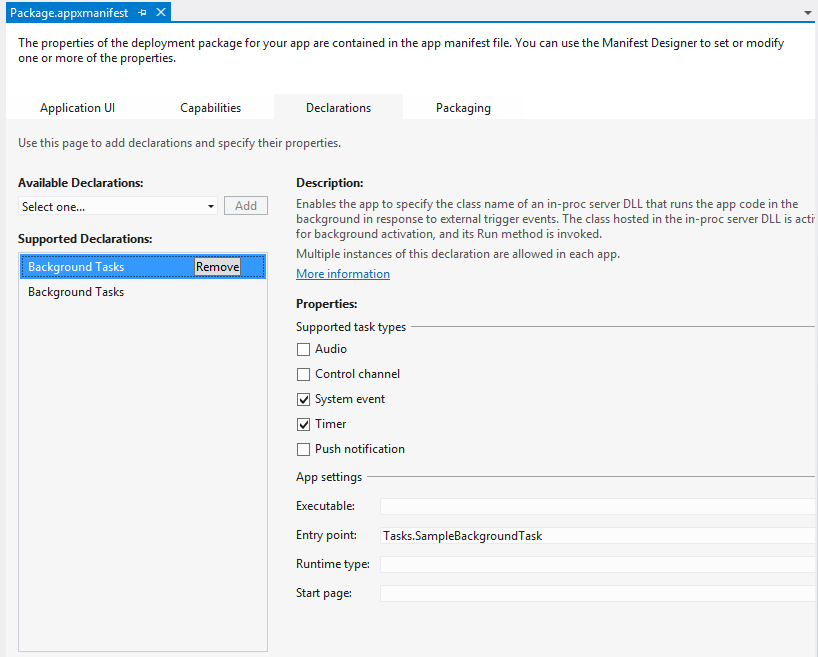


Figure 3 - Sample manifest for C# app

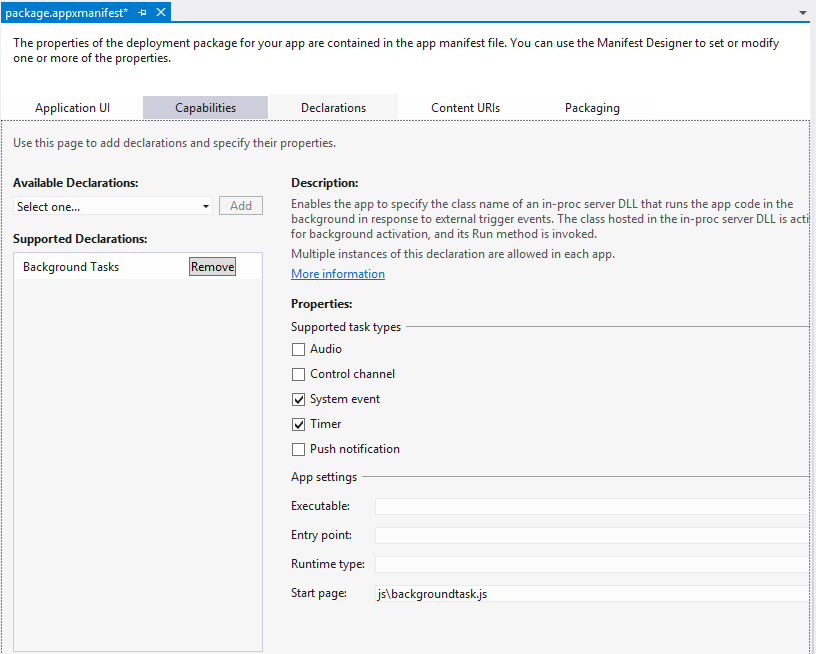


Figure 4 - Sample manifest for JavaScript app

The extension also supports an **Executable** element that specifies the executable that loads the in-proc DLL that contains the background task class. The Executable rules are defined in the previous section.

For more information about authoring manifests, see “Resources,” at the end of this paper.

## Debugging background task activation

The background task can be triggered through Visual Studio, so that it’s easy to debug during development. Simply put a breakpoint in the Run method and trigger the background task using the suspend drop down menu available in the Debug Location toolbar. This drop down shows the names of the background tasks that can be activated by Visual Studio.

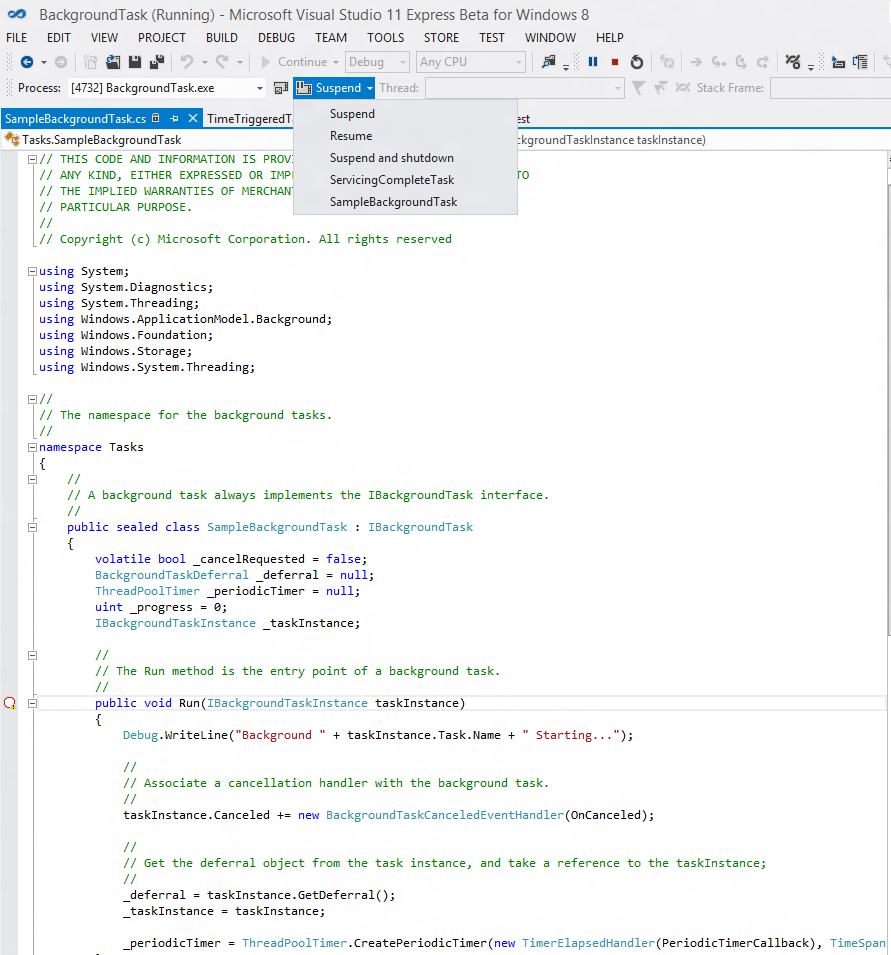


Figure 5 - Debugging background tasks

If the background task is not being activated for some reason even though the task is described appropriately in the manifest, registered in the app code, and meets the lock screen requirements (if needed), enable debug tracing in the event viewer to debug further.

To enable debug tracing of background tasks, use the following steps:

1. Open event viewer by pressing the Windows button and searching for eventvwr.exe.
2. Go to Application and Services Logs -> Microsoft -> BackgroundTaskInfrastructure in the event viewer.
3. In the actions pane, select View-> Show Analytic and Debug Logs to enable the diagnostic logging.
4. Select the Diagnostic log and click Enable Log.
5. Now try activating the background tasks again.
6. View the Diagnostic logs for detailed error information.

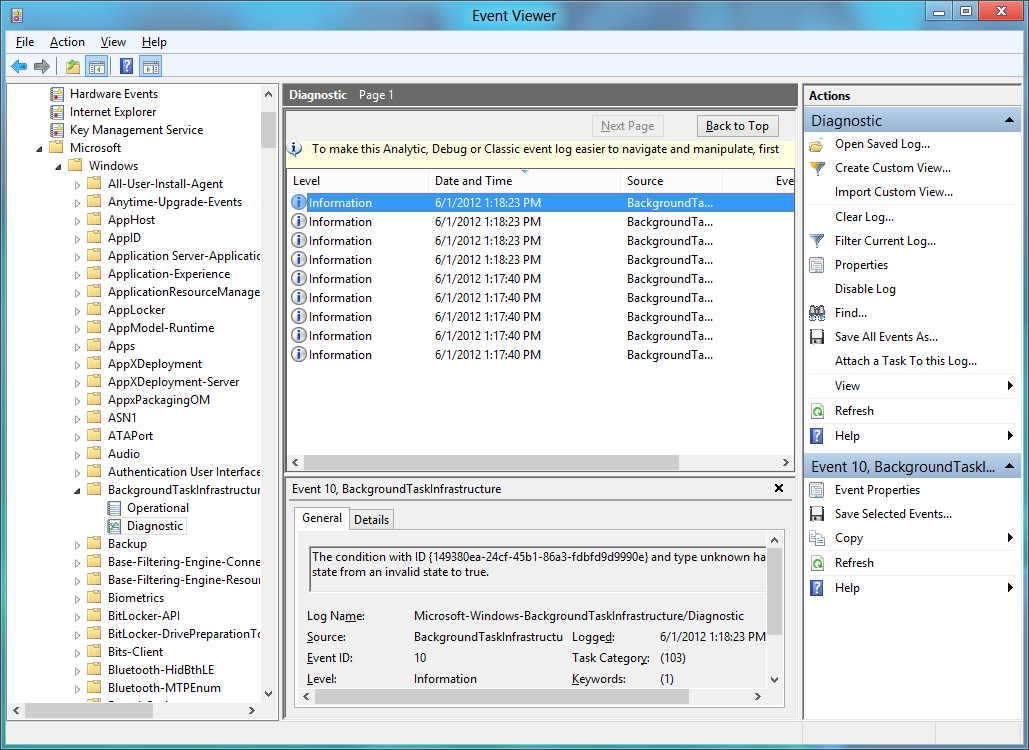


Figure 6 - Event viewer for background tasks debug information

# Background task resource management

Background tasks are meant to be short-lived tasks that do not consume a lot of resources. Background tasks have CPU and Network resource usage constraints applied to them. This prevents an app in the background from depleting the device’s battery without the user being aware.

If an app is running and the user is interacting with the app in the foreground, then no CPU or Network resource constraints are applied to the app’s background tasks. If an app is not active in the foreground, then its resources are constrained as described in this section.

## CPU resource constraints

An app on the lock screen receives a certain amount of CPU time at regular intervals for its background tasks. If the app uses all of its available CPU time, its background tasks are suspended until the app’s CPU quota is replenished at the next generation for CPU quota updates.

Each app on the lock screen receives 2 seconds of CPU time every 15 minutes, which can be used by all of the background tasks of the app. At the end of 15 minutes, each app on the lock screen receives another 2 seconds of CPU time for use by its background tasks. Any unused CPU time in the 15-minute interval is lost and not accumulated by the app. Each app not on the lock screen receives 1 second of CPU time every 2 hours.

**Table 6 – CPU resource constraints on background tasks**

|  |  |  |
| --- | --- | --- |
|  | CPU resource quota | Refresh period |
| Lock screen app | 2 CPU seconds | 15 minutes |
| Non-lock screen app | 1 CPU second | 2 hours |

## Network resource constraints

Network usage can represent a significant drain on a device battery, and so it is also constrained during background task execution. However, if a device is running on AC power, then background tasks are not network constrained. They are free to use as much network bandwidth as they need (gated, of course, by any carrier-specific bandwidth or usage constraints). Note that CPU usage for a background task is always resource constrained even if the device is running on AC power.

Network resource constraints are a function of the amount of energy used by the network interface, which is modeled as the amount of time the network is used to transfer bytes (for download or upload); it is not based on any throughput metrics. The throughput and energy usage of a network interface varies based on factors like link capacity, physical position of the device, or the current load on the WiFi network. WiFi networks typically have lower energy consumption per byte sent and received when compared to cellular mobile networks.

The following table characterizes the approximate amount of data an application should be able to download, based on different network throughputs. This assumes a resource-constrained WiFi network for developers building background tasks. The example WiFi interface also assumes minimal interference and ideal conditions.

**Table 7 – Example network throughput for background tasks on DC with a WiFi network**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Average throughput | Data throughput, in megabytes (MB)  for lock screen apps | | Data throughput, in MB, for non-lock screen apps | |
| **Every 15 minutes** | **Per day** | **Every 2 hours** | **Per day** |
| 1 megabits per second (Mbps) | 0.469 | 45 | 0.625 | 7.5 |
| 10 Mbps | 4.69 | 450 | 6.25 | 75 |

## Critical background tasks

Real-time applications like VOIP that rely on the **Control Channel** or **Push Notification** trigger may not want their critical background tasks to be suspended due to resource constraints in the middle of something important; for example, an incoming call. Hence, background tasks that use the Control Channel or Push Notification trigger receive guaranteed application resource quotas (CPU and network) for every running task. The resource quotas and constraints are as mentioned in previous sections and remain the same for these background tasks.

The developer does not have to do anything different programmatically; the system knows Control Channel and Push Notification background tasks, and they are all treated as critical background tasks.

## Global pool

Even with the assigned quota for each application, there will be scenarios where these fixed resource constraints may not be enough, and for those situations, there is a global pool from which applications can use CPU and network resources. The global pool is designed for rare instances where the guaranteed resources provided to apps are not enough and they need more CPU or network resources to complete their tasks. Apps should not rely on the availability of global pool because it’s a shared resource and other apps could have depleted the global pool. The global pool is replenished every 15 minutes at varying rates of refill depending on whether the computer is running on AC or DC power.

When developing your application, you should disable global pool to ensure that your background tasks run within the default CPU and network resource constraints.

**Table 8 – Registry keys to control global pool behavior**

|  |  |  |  |
| --- | --- | --- | --- |
| Value name | Type | Default value | Description |
| **HKEY\_LOCAL\_MACHINE\ SOFTWARE\Microsoft\Windows NT\CurrentVersion\BackgroundModel\Policy\CpuEnableGlobalPool** | **DWORD** | 1 | Controls the CPU global pool. A value of zero disables CPU global pool. |
| **HKEY\_LOCAL\_MACHINE\ SOFTWARE\Microsoft\Windows NT\CurrentVersion\BackgroundModel\Policy\NetEnableGlobalPool** | **DWORD** | 1 | Controls the network global pool. A value of zero disables network global pool. |

If a background task is suspended because it has used its 2 seconds of CPU time and there are no resources in the global pool to use, a message is logged in the event viewer (see Table 8). A similar message is logged if the background task is suspended because it used all of its network resources and there are no network resources available in the global pool. In addition, the **BackgroundTaskInstance** class also has a **SuspendedCount** field that contains the number of times the background task has been suspended due to resource management constraints.

Table 9 – Event viewer message when a background task is suspended

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Provider | Level | Event ID | Channel | Message |
| Microsoft-Windows-BackgroundTaskInfrastructure | Informational | 4 | Operational | Background task for package “” with entry point “” was suspended due to CPU resource management policy. |

## What is CPU usage time?

CPU usage time refers to the amount of CPU time used by the app and not the wall clock time of the background task. For example, if the background task is waiting in its code for the remote server to respond, and it is not actually using the CPU, then the wait time is not counted against the CPU quota because the background task is not using the CPU. CPU usage information is available in Task Manager or with more sophisticated tools like xperf, which is available in the Windows Performance Analysis Tools. For more information, see “Resources,” at the end of this paper.

The reason for using CPU usage as a unit of resource management instead of total running time or wall clock time is that CPU usage time is a fairer metric for measuring actual energy usage of an app. If an app is not using the CPU and might be waiting for input, the CPU could go to a low power state, conserving battery life.

## Idle background tasks

When a device is ready to enter Connected Standby mode, idle background tasks that prevent entry to Connected Standby mode are cancelled. A background task is considered idle if, in the last two and half minute window, there was no input-output (IO) activity for the app and it is not being suspended due to exhaustion of its resource quota (which is an indicator of activity). IO activity is defined as all input/output associated by the app, including disk and network activity.

If an idle background task is detected, the background task infrastructure will first try to cancel the task with a **CancelReason** of **Abort**. If the task is well behaved, then it can use this cancel notification to complete and return gracefully from the background task. If the task is not well behaved and does not return gracefully from the background task, then the background task infrastructure will terminate the app. In this case, a message is logged in the **BackgroundTaskInfrastructure** channel in eventvwr.

# Threading model for background tasks hosted in the app

If background tasks authored in C# or C++ are hosted within the app instead of the default system-provided host (BackgroundTaskHost.exe or WWAHost.exe), there are some threading model complexities to keep in mind.

## Decoupling the background task from the app

For non-JavaScript apps, the background tasks are hosted in an in-proc DLL that is loaded in a multi-threaded apartment (MTA) within the app. For JavaScript apps, background tasks are launched in a new single-threaded apartment (ASTA) within the WWA host process. The actual background task class can be either an STA or MTA threading type. Because background tasks can run when an app is in a Suspended or Terminated state, they need to be decoupled from the foreground app. Loading the background task DLL in a separate apartment enforces separation of the background task from the app while allowing the background task to be controlled independently of the app.

When an app is suspended, the UI STA thread is blocked in the Windows kernel. This thread is released only when the app transitions back to a Running state in response to user interaction. When the app is in the background and a background task is triggered, all threads in the app except the UI STA thread are unfrozen and the background task is activated in an MTA thread. The UI STA thread remains blocked. If the background task tries to access an object that is owned by the blocked UI STA thread, then it will deadlock. To prevent this deadlock, the background task should not share data between the app and the background task. Any shared objects must aggregate the Free Threaded Marshaler (FTM). The Control Channel trigger demonstrates such usage. For more information about the Control Channel, see “Resources,” at the end of this paper.

## Sharing state between the background task and the foreground app

Another aspect to keep in mind if the background task is loaded within the app instead of the default system-provided host executable is that it cannot rely on accessing the memory of the foreground app. Background tasks run regardless of the current state of the app, so background tasks cannot rely on having the app around when they run. The only reliable way for the background task to share state with the app is to use persistent storage, such as ApplicationData, or files.

# Background task best practices

When you use background tasks in your app, follow these best practices:

* Design background tasks to be short lived.
* Design the lock screen user experience as described in the “*Guidelines and checklists for lock screen tiles.*”
* Do not specify the Executable attribute to ensure the task launches in the system-provided host.
* Describe the background task class name or JavaScript file name accurately in the manifest.
* Look in the event viewer for error messages if the background task is not being activated.
* Use persistent storage to share data between the background task and the app.
* Register for progress and completion handlers in the app.
* Register for a background task cancellation handler in the background task class.
* Register for the **ServicingComplete** trigger if you expect to update the app.
* Ensure that the background task class library is referenced in the main project and its output type is winmd.
* Describe the triggers in the background manifest accurately.
* Verify if the app needs to be on the lock screen.
* Do not display UI other than toast, tiles or badges from a background task.
* Do not rely on user interaction in background tasks.

# Resources

##### Windows Store app development

Background task sample

<http://go.microsoft.com/fwlink/?LinkId=227509>

Background networking whitepaper (Control Channel trigger)

<http://go.microsoft.com/fwlink/?LinkId=241645>

ControlChannelTrigger TCP socket sample

<http://go.microsoft.com/fwlink/?LinkId=241562>

Guidelines and checklists for Lock screen tiles

<http://go.microsoft.com/fwlink/?LinkId=241560>

Lock Screen overview

<http://go.microsoft.com/fwlink/?LinkId=241559>

Lock Screen sample application

<http://go.microsoft.com/fwlink/?LinkId=239970>

Push Notification background task sample (Raw Notifications)

<http://go.microsoft.com/fwlink/?LinkId=241553>

Push Notification background task sample

<http://go.microsoft.com/fwlink/?LinkId=241341>

Learn to build Windows Store apps

<http://go.microsoft.com/fwlink/?LinkId=227172>

Application lifecycle

<http://go.microsoft.com/fwlink/?LinkId=227176>

##### Windows Store app reference documentation

Audio with the Playback Manager - Windows.Media.PlayTo

<http://go.microsoft.com/fwlink/?LinkId=227494>

Background task and lock screen - Windows.ApplicationModel.Background

<http://go.microsoft.com/fwlink/?LinkId=227174>

JavaScript background task documentation

<http://go.microsoft.com/fwlink/?LinkId=241460>

Background transfer - Windows.Networking.BackgroundTransfer

<http://go.microsoft.com/fwlink/?LinkId=227496>

File share contracts - Windows.ApplicationModel.DataTransfer

<http://go.microsoft.com/fwlink/?LinkId=227498>

File share contracts - Windows.ApplicationModel.DataTransfer.ShareTarget

<http://go.microsoft.com/fwlink/?LinkId=227499>

Windows push notification service - Windows.Networking.PushNotifications

<http://go.microsoft.com/fwlink/?LinkId=227501>

##### Additional resources

Manifest authoring support in Visual Studio – Manifest Designer

<http://go.microsoft.com/fwlink/?LinkId=227503>

Windows Performance Analysis Tools

<http://msdn.microsoft.com/en-us/performance/cc825801.aspx>

Comments and questions

[bgtaskfeed@microsoft.com](mailto:bgtaskfeed@microsoft.com)