Increasing System Power Efficiency through Driver Support for Runtime Idle Detection

April 23, 2008

Abstract

One way to improve a computer’s power efficiency is to power down devices that aren't currently being used, even though the system may still be active and in its normal operating power state (S0). The practice of powering down an idle device while the system remains running is referred to as “runtime idle detection” or “S0 idle.” A USB function driver supports runtime idle detection by implementing USB selective suspend. This white paper provides advice for driver developers on how to modify their function drivers to support runtime idle detection.

This information applies for the following operating systems:  
 Windows Server® 2008  
 Windows Vista®  
 Windows Server 2003  
 Windows® XP  
 Windows 2000

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://www.microsoft.com/whdc/system/pnppwr/powermgmt/s0idle_driver.mspx>

**Feedback:** Please tell us whether this paper is useful to you. Give us your comments at:   
 [http://go.microsoft.com/fwlink/?LinkId=104880](https://mail.windows.microsoft.com/owa/redir.aspx?C=a3fb4d360a624295ac588e56b791aba0&URL=http%3a%2f%2fgo.microsoft.com%2ffwlink%2f%3fLinkId%3d104880)

Disclaimer: This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2007–2008 Microsoft Corporation. All rights reserved.

Microsoft, Visual Basic, Windows, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Document History

| **Date** | **Change** |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | | | |
| April 28, 2008 | Added more information about idle timers to this paper.  Moved all information about USB selective suspend to “Selective Suspend in USB Drivers.” | | | |
| November 15, 2007 | First publication | | | |

Contents

[Introduction 4](#_Toc196811058)

[A Generic Algorithm for Supporting Runtime Idle Detection 5](#_Toc196811059)

[Details of the Generic Algorithm 6](#_Toc196811060)

[Implementing the Idle Timer 6](#_Toc196811061)

[Stopping the Power-Down Process 7](#_Toc196811062)

[Arming the Device for Wake 7](#_Toc196811063)

[Corner Cases 7](#_Toc196811064)

[Devices That Cannot Wake Themselves 8](#_Toc196811065)

[An Algorithm for Supporting Runtime Idle Detection on USB Devices 8](#_Toc196811066)

[Details of the USB Selective Suspend Algorithm 10](#_Toc196811067)

[Sending the USB Selective Suspend Request 11](#_Toc196811068)

[Canceling the USB Selective Suspend Request 11](#_Toc196811069)

[Powering Down the Device 11](#_Toc196811070)

[Finishing the USB Selective Suspend Operation 12](#_Toc196811071)

[USB Selective Suspend and Power Policy 12](#_Toc196811072)

[Evaluating Driver Support for Runtime Idle Detection 13](#_Toc196811073)

[Evaluating I/O Request Code Paths 13](#_Toc196811074)

[Evaluating Non-I/O Request Code Paths 15](#_Toc196811075)

[Disabling Runtime Idle Detection for Stop and Remove Requests 15](#_Toc196811076)

[Determining the Runtime Idle Detection Policy 16](#_Toc196811077)

[Issues to Consider for All Devices 17](#_Toc196811078)

[Issues to Consider for USB Devices 19](#_Toc196811079)

[Guidelines for Implementing Runtime Idle Detection 20](#_Toc196811080)

[Adding Runtime Idle Detection to a KMDF Function Driver 20](#_Toc196811081)

[Adding Runtime Idle Detection to a WDM Function Driver 21](#_Toc196811082)

[Testing Runtime Idle Detection Implementations 22](#_Toc196811083)

[Resources 22](#_Toc196811084)

# Introduction

Power efficiency is critically important for modern computers. One way to help limit a computer’s energy consumption is to power down devices that aren’t currently being used, even though the system might still be active and in its normal operating power state (S0). The practice of powering down idle devices while the system remains running is referred to as “runtime idle detection,” or sometimes as “S0 idle.”

USB drivers have specialized runtime idle detection requirements to support USB devices that are attached to the same parent, including individual functions on multifunction devices. To support runtime idle detection for a USB device, a function driver must implement USB selective suspend. This enables a function driver to suspend those devices or functions that are not currently in use, while the other devices or functions remain in their normal operating state.

This white paper provides guidelines for driver developers on how to modify existing function drivers to support runtime idle detection. It will help you scope the work that is required to implement runtime idle detection for your device and design an effective implementation. For USB drivers, these guidelines apply only for existing drivers that use the Windows Driver Model (WDM). All new USB drivers should use the Windows Driver Framework (WDF) and should support selective suspend as described in “Selective Suspend in USB Drivers,” which is listed in the “Resources” section.

This paper is intended for driver developers who are experienced with implementing power management, including how to correctly handle system power transitions. It assumes, for example, that you are familiar with how to correctly handle wait/wake I/O request packets (IRPs) and how to power down a device. For a complete discussion of these and other power-related topics, see “Plug and Play and Power Management in WDF Drivers” on the WHDC Web site and “Power Management” in the Windows® Driver Kit (WDK) documentation.

For a more general discussion of power efficiency, see “Windows Vista Energy Conservation” on the WHDC Web site.

The following table defines key terms that are used in this paper.

Power-related Terminology

| **Term** | **Definition** |
| --- | --- |
| arm for wake | The procedure that a function driver uses—before the driver powers down the device—to prepare the device to wake itself. |
| D*x* | A collective term for any of the device low-power states (D1 through D3). D0 is the normal device operating state. |
| D*x* IRP | A power IRP that the function driver sends to the device stack to transition the device to D*x*. |
| idle timer | A timer that tracks the time since a device was last used. When the timer reaches the selected idle time-out value, it expires and notifies the function driver to start the power-down process. |
| power policy owner (PPO) | The driver that controls the device’s power state and decides when to change that state. Each device stack has a single PPO, usually the stack’s function driver. |
| runtime idle detection | Conserving energy by powering down an idle device while the system remains in its normal operating state. |
| S*x* | A collective term for any of the system low-power states (S1 through S5), including standby, hibernate, and so on. S0 is the normal system operating state. |
| S*x* IRP | A power IRP that is sent to the device stack to notify it that the system is about to transition from S0 to S*x*. |
| USB selective suspend | A feature that enables a function driver to ask the parent driver to suspend an individual USB device or function while the other devices or functions that are connected to the same parent remain in their normal operating state. |
| Wake from S0 | A device waking itself when the system is in S0. This is the runtime idle detection scenario. |
| Wake from S*x* | A device waking itself and the system when the system is in S*x*. |

# A Generic Algorithm for Supporting Runtime Idle Detection

The basic process for implementing runtime idle detection in a driver is straightforward:

1. The PPO—which is usually the stack’s function driver—determines that the device is idle and starts the idle timer.

The discussion in this paper assumes that the PPO is the function driver.

2. When the device has been idle long enough that it should be powered down, the idle timer expires and notifies the function driver.

3. If the device can wake itself, the function driver arms the device for wake.

Devices that cannot wake themselves can also support runtime idle detection. For details, see “Devices That Cannot Wake Themselves” later in this paper.

4. The function driver powers down the device.

A robust runtime idle detection implementation must also be able to handle several inherent race conditions. For example, a race condition can occur when the function driver determines that the device should remain in D0 but the idle timer has already expired and has started the power-down process.

Figure 1 shows a state machine for implementing runtime idle detection. The algorithm for USB selective suspend must handle some additional issues, as discussed later in this paper.

**Note:** Figure 1 shows the core runtime idle detection scenario, but a complete implementation must also be prepared to handle several corner cases. For details, see “Corner Cases” later in this paper.

To effectively implement the logic shown in Figure 1, you must choose an optimal idle time-out value. For details, see “Evaluating Driver Support for Runtime Idle Detection” later in this paper.



Figure 1. Runtime idle detection flow chart

## Details of the Generic Algorithm

This section discusses the key details of the algorithm for runtime idle detection.

### Implementing the Idle Timer

The function driver can either use the idle time-out capabilities that the power manager provides or can implement its own idle timer.

The power manager’s time-out capability consists of a simple timer that sends a device power IRP. The driver opts to use this mechanism by calling **PoRegisterDeviceForIdleDetection**—typically from the *AddDevice* routine—and supplying time-out values for performance and conservation power schemes together with the D*x* state that the device should enter when it is idle. Each time that an I/O request arrives that requires hardware access, the driver calls **PoSetDeviceBusy** to restart the timer. When the timer expires, the power manager sends a device power IRP to the device stack so that the function driver can transition the driver to the idle power state. This approach is recommended for drivers that run only on Windows Vista and later releases.

On releases earlier than Windows Vista, the power manager’s idle time-out mechanism works well only for simple drivers and devices that do not support device wake. On these earlier releases, the power manager does not synchronize the idle device power IRPs with system power transitions. Thus, a device stack could potentially receive consecutive D*x* IRPs. The function driver must maintain enough state information to ensure that it does not try an incorrect state transition in such situations, and driver testing must verify that the idle code works correctly in all such cases.

For most drivers that must run on releases earlier than Windows Vista, implementing a device type–specific idle timer is simpler than using the power manager’s capability. At any given time, the function driver has more knowledge about the activity level of its device than the power manager does, so the function driver can more accurately start and cancel an idle timer. Moreover, the driver requires less state-tracking code because it manages the transitions itself.

### Stopping the Power-Down Process

To stop the power-down process and keep the device in D0, the function driver must cancel the idle timer before the timer expires and starts the power-down process. If the timer has expired, the only way to return the device to D0 is to finish powering down the device and then power it up again.

### Arming the Device for Wake

If a device can wake itself, the function driver must arm it for wake before requesting a D*x* IRP to power down the device. Generally, for a good user experience:

* Devices such as a mouse (which must respond to user interactions) or a network adapter (which must respond to events generated by external sources) should be able to wake themselves.
* Devices such as a hard drive (which are not required to respond to unsolicited events) are not required to wake themselves. The function driver can wake these devices.

**Note:** For some devices, arming the device for wake from S0 is different from arming the device for wake from S*x*. For example, a network adapter might wake from S0 when any packet arrives, but wake from S*x* only when certain Magic Packets arrive.

## Corner Cases

Figure 1 shows the core runtime idle detection scenario, but a complete implementation must also be able to handle corner cases such as the following:

* The function driver cancels the timer to disable it, but the timer has already expired.

The driver must power down the device and then power it up again before it can cancel the timer.

* An S*x* IRP arrives during the power-down process.

If a device must be armed differently for wake from S*x* than for wake from S0, the function driver must:

1. Wait until the device has powered down.

2. Power up the device again.

3. Arm the device for wake from S*x*.

4. Power down the device.

The function driver must also go undergo this process if, for example, the device uses a different D*x* state for wake from S0 than it does for wake from S*x*.

* The wait/wake IRP fails after the function driver sends the IRP down the stack, but the function driver receives the failure notification while it is requesting or processing the D*x* IRP.

A driver can handle this situation in one of the following ways:

Leave the device powered down.

However, the attempt to arm the device for wake failed, so the device cannot wake itself and will not respond to user actions. The function driver must determine when to wake the device and initiate the wake process.

After the device has powered down, immediately power it up again.

This might waste power, but the device remains functional.

## Devices That Cannot Wake Themselves

A device that cannot wake itself can still support runtime idle detection. The function driver wakes the device as required, for example, in these cases:

* When an IRP arrives—such as a Plug and Play state change IRP, a read or write IRP, and so on—that can be processed only when the device is in D0.
* When a Windows Management Instrumentation (WMI) request arrives that disables runtime idle detection support.
* When an S*x* IRP arrives, which indicates that the system is about to transition from S0 to S*x*.

If the device uses a different D*x* state for wake from S*x* than for wake from S0, the function driver must wake the device and then transition it to the appropriate D*x* state.

# An Algorithm for Supporting Runtime Idle Detection on USB Devices

USB devices support runtime idle detection through USB selective suspend. This technique enables a USB function driver to request that its device be powered down without affecting other devices that are connected to the same hub. WDM function drivers for USB devices must implement the runtime idle detection algorithm that is described in this paper.

WDF function drivers implement selective suspend by using the support that is provided in the driver frameworks and in the underlying Windows-supplied USB drivers. For detailed information about supporting selective suspend in a WDF driver, see “Selective Suspend in USB Drivers” on the WHDC Web site.

**Caution:** A USB function driver must use USB selective suspend to support runtime idle detection. Failing to do this can prevent other devices from selectively suspending.

USB selective suspend serves several purposes, including:

* Providing a workaround for host controller limitations.

For example, the root ports on an Intel 82801BA I/O Controller Hub (ICH2) host controller will not power up unless the entire host controller is power cycled.

* Providing a workaround for hardware limitations.

For example, some hubs cannot manage the power state separately for each port. The hub can power down a single device; but to power up the device again, the hub and all attached devices must be power cycled.

* Preventing the function driver for a function on a multifunction USB device from issuing a wait/wake notification when the underlying device remains in D0.

The last item in the preceding list is an issue only for multifunction USB devices. From the perspective of the generic parent driver, the power state is associated with the device itself, not the individual functions. Consider what would happen to a multifunction device with two functions, A and B, if USB selective suspend were not supported:

1. Function A becomes idle. Its function driver arms the device for wake and sends a D*x* IRP to direct the generic parent driver to power down the device.

Function B remains active, so the device must remain in D0.

2. Even though Function A is in D*x* and armed for wake, the device remains in D0.

A USB device cannot signal wake when it is in D0, so the function cannot wake itself.

3. Function A is now effectively disabled, unless the function driver wakes it.

USB selective suspend enables function drivers to suspend individual functions and wait for a wake notification in the same way that they would for a stand-alone device. However, the parent driver synchronizes the power states of the various functions and does not actually power down the device unless all functions have been suspended.

To support USB selective suspend, a WDM function driver must implement a somewhat more complex runtime idle detection algorithm than the one discussed earlier for non-USB devices. Specifically:

* For USB selective suspend, when the idle timer expires, the WDM function driver sends a USB selective suspend IRP to the bus driver that notifies it to power down the device.

The parent driver initiates the power-down process as soon as it is appropriate to do so. The function driver does not start the power-down operation until the parent driver requests the action.

For comparison, when the idle timer expires for a non-USB device, the function driver immediately starts the power-down operation.

The USB selective suspend mechanism introduces a new pending state in which the device is idle and the idle timer has expired, but the parent driver has not yet requested the power-down operation.

* With the current USB selective suspend implementation, the USB selective suspend callback function cannot return until the device stack has been powered down. The idle state machine must therefore have a synchronization point for coordination with the USB selective suspend callback function.

Figure 2 shows a state machine for USB selective suspend. The algorithm is similar to the generic runtime idle detection algorithm discussed earlier, but has some additional USB-specific components, which are highlighted in gray. As with Figure 1, this diagram does not include corner cases.



Figure 2. USB selective suspend algorithm for WDM drivers

## Details of the USB Selective Suspend Algorithm

This section describes how a WDM function driver participates in USB selective suspend. “Device” as used in this discussion refers to either a stand-alone device or a function on a multifunction device. The discussion focuses on the USB-specific aspects of the algorithm and omits steps such as canceling the idle timer and arming the device for wake, which are handled in the same way for USB devices as described earlier in this paper.

### Sending the USB Selective Suspend Request

When the device’s idle timer expires, the WDM function driver initiates the USB selective suspend process by creating a USB selective suspend request IRP—sometimes called an “Idle request”—and sending it down the stack to the parent driver. The USB selective suspend request is an internal device I/O control request with an IOCTL\_INTERNAL\_USB\_SUBMIT\_IDLE\_NOTIFICATION device I/O control (IOCTL). The request includes a pointer to the function driver’s USB selective suspend callback function.

When the parent driver receives the USB selective suspend request, it does one of the following:

* Stores the USB selective suspend callback function pointer and context structure and pends the IRP.

—or—

* Fails the IRP.

Failure usually means that USB selective suspend has been disabled at the parent level—either Usbhub.sys or Usbccgp.sys, depending on the device—typically by the user. However, the function driver is not notified if the user decides to reenable USB selective suspend. For that reason, function drivers should retry failed attempts to initiate USB selective suspend. As an example of how long to wait before retrying, the kernel-mode driver framework (KMDF) runtime resets the idle timer and waits for it to expire again.

### Canceling the USB Selective Suspend Request

If the device should remain in D0, the function driver can call **IoCancelIrp** to attempt to cancel the selective suspend request. If a USB selective suspend request completes as canceled, the USB selective suspend operation has been terminated.

**Important:** If **IoCancelIrp** returns TRUE, that response does not necessarily indicate that the USB selective suspend operation has been terminated. To be certain that the USB selective suspend operation is finished, the function driver must wait asynchronously until the parent driver calls the USB selective suspend callback.

### Powering Down the Device

When the parent driver is ready to power down the device, it calls the function driver’s USB selective suspend callback function. The USB selective suspend request remains pended, but now it cannot be canceled. Depending on the version of Windows:

* For Windows Vista®, the parent driver can call the USB selective suspend callback function as soon as the USB selective suspend request arrives.
* For Windows XP, the parent driver pends USB selective suspend IRPs until it has received one from each attached device. Then the parent driver calls each device’s USB selective suspend callback function.

After the devices are powered down, the parent driver powers down the hub.

Although the behavior of the parent driver is different for these two versions of Windows, the function driver should behave identically for either version.

When the parent driver calls the USB selective suspend callback function, the function driver should synchronously power down its device, as follows:

1. If the device can wake itself, the function driver arms the device for wake from S0.

2. The function driver requests a D*x* IRP and sends it to the top of the stack.

3. When the parent driver receives the D*x* IRP, it powers down the device.

4. After the device has transitioned to D*x*, the USB selective suspend callback function returns.

For USB devices, D1 through D3 are software constructs and have basically the same wake latency. For Windows XP, D2 is the level that can be armed for wake, so function drivers for such devices should transition the device to D2. For Windows Vista, a device in D2 or D3 can be armed for wake.

Because of the potential race conditions associated with clearing a cancellation routine, the USB selective suspend callback function might be invoked even though the idle request has been canceled. In this case, the callback function must still power down the device and pend a wait/wake IRP.

**Important:** A function driver should not cancel the USB selective suspend IRP in the USB selective suspend callback function. If it does, the request will not complete until after the power-down process is complete and the USB selective suspend callback has returned.

### Finishing the USB Selective Suspend Operation

After the device returns to D0, the parent driver completes the USB selective suspend request as successful. This notifies the function driver that the USB selective suspend operation is finished. If the device is surprise removed while it is powered down, the parent driver completes the USB selective suspend request as canceled.

Function drivers usually are not required to synchronize USB selective suspend request completion with the device’s power policy state.

## USB Selective Suspend and Power Policy

A key feature of USB selective suspend is that the function driver initiates the operation but relies on the parent driver to decide when to begin powering down the device. To correctly handle this change of ownership, it might be necessary to modify the function driver’s power policy implementation.

Most USB function drivers completely control the power policy, so modifying the implementation to support USB selective suspend is straightforward. The exception is miniport drivers, which rely on the associated port driver to handle power policy. In that case, the port driver must also be modified to support USB selective suspend.

A miniport driver cannot use USB selective suspend if the port driver does not provide corresponding support. For USB selective suspend to work correctly in a port/miniport model, the port driver must be aware of the additional pending state that USB selective suspend requires. Otherwise, the port driver—if it supports runtime idle detection at all—attempts to power down the device and perhaps arm it for wake without properly coordinating its actions with the parent driver.

**Important:** HIDClass is the only system-supplied port driver that supports USB selective suspend. For more information, see “HIDClass Devices” in the WDK documentation.

# Evaluating Driver Support for Runtime Idle Detection

Your first task is to identify the idle-related parts of your driver. This is the code that you might be required to update to correctly support runtime idle detection. This section outlines the questions that you should ask to evaluate your function driver’s support for runtime idle detection and to identify the code that you must modify. For details about how to implement the related support, see “Guidelines for Implementing Runtime Idle Detection” later in this paper.

## Evaluating I/O Request Code Paths

This section discusses how to evaluate the code that handles I/O requests: read, write, and device IOCTL requests.

#### Must the device be in D0 to process an I/O request?

Some types of I/O requests require device access, so the device must be in D0 to process the request. I/O requests that can be processed only if the device is in D0 must be pended until the device has reentered D0.

**Note:** Windows Driver Foundation (WDF) function drivers can use power-managed queues to handle power-dependent requests. These queues automatically queue requests that are received when the device is in D*x* and start dispatching requests only when the device reenters D0. For a discussion of how and when to use WDF power-managed queues, see “When WDF Drivers Can Use Power-Managed I/O Queues” on the WHDC Web site.

#### Does the driver pend requests? If so, for how long?

If a function driver pends any requests—especially if it does so for a long or indefinite time period—you must evaluate how the resource or event that enables the request to be completed will be affected if the device idles out. One useful way to analyze the issue is to ask: what would happen to a pending request if the device must be powered down because the system is transitioning to S*x*? If a request can remain pending while the system is in S*x*, then it can remain pending when the device is idled out.

You should consider the following two primary issues:

* Resource contention.

If a request requires resources that are being used elsewhere in the driver, the driver must pend the request until it has relinquished the resources. You must determine whether the device must be in D0 to handle the request. For example, if the driver uses a direct memory access (DMA) scatter-gather list to handle a request, the device must be in D0.

* Data transfer delays.

Requests are sometimes pended while waiting for data to be transferred to or from the device. Must the device remain in D0 to complete the request? In other words, can the device be powered down and powered up again while the request is pending without causing data loss? Examples include:

If a device such as a keyboard or a network adapter that receives unsolicited input is in the midst of transferring data, the device cannot be idled until the data transfer is complete.

The function driver should direct the device to stop transmitting data before powering down the device. For some devices, arming the device for wake automatically notifies the device to stop transmitting data.

If a function driver has a pended connection-notification request, the device can be idled out if the underlying hardware can wake itself when it receives a connection request from another entity.

For example, if a Bluetooth function driver has a pended server connect request, the function driver can idle out the Bluetooth radio if the radio is capable of waking when, for example, it receives a remote radio connect request.

#### Must the request be handled in the calling application’s context?

Some I/O requests—such as those that contain embedded buffers or use method neither I/O—must be processed in the context of the calling application. If a request of this type requires the device to be in D0, a driver should process the request in two stages:

1. Capture the buffers while in the application’s context and then pend the request.

2. After the device has reentered D0, process the request.

You must be careful about how you handle cancellation between the two stages and ensure that the driver correctly cleans up before completing the request.

#### Does the driver handle only file-based I/O?

If a function driver handles only I/O bound to a file object—that is, file-based I/O—the driver is not required to be locked while the device is idled out against most changes in the Plug and Play state. If the device has open file handles, the PnP manager does not enable the device to be removed. The exception is a Plug and Play Stop request, which requires the function driver to stop touching hardware, even for file-based I/O.

If a function driver handles I/O that is not bound to a file object, the driver might require more locking to correctly handle runtime idle detection. The amount of additional locking logic depends on the I/O contract between the driver and the kernel-mode client that issued the request. For example, a driver could use a device-wide lock to guard the Plug and Play state, as follows:

1. For each I/O request, the driver acquires the lock and checks the Plug and Play state.

2. Depending on the state, the driver either processes or pends the request.

#### What is the affect of an upper filter driver?

If the upper edge of your function driver is public, a filter driver could be installed above it in the stack. Adding an upper filter driver usually does not affect the function driver. However, you should consider two issues:

* How would an upper filter driver interact with your function driver in a runtime idle detection scenario?
* What assumptions might the filter driver make about device power state when it sends requests down the stack?

Specific examples of issues to consider include the following:

* Would the presence of an upper filter driver affect how the function driver handles device power-up latency?
* How should the function driver handle I/O requests that were completed synchronously but—with the addition of an upper filter driver—might be pended and completed later?

## Evaluating Non-I/O Request Code Paths

This section provides some guidelines for evaluating the code that handles non-I/O requests—including Plug and Play, power, and WMI requests—to determine whether the code must be modified to support runtime idle detection.

#### How does the driver handle an S*x* request when the device is in D*x*?

If a function driver receives an S*x* request while the device is in D*x*:

* If the device is in D*x* but is not armed for wake, the function driver can leave the device in that state until the system returns to S0.

One exception to this rule is when the D*x* state that is used for runtime idle detection is different from the state used for S*x*. In that case, the function driver must wake the device and then put it in the correct D*x* state for S*x*.

* If the device is in D*x* and armed for wake, the function driver might be required to power up the device and disarm it.

The function driver then arms the device for wake from S*x*—if the device supports that feature—and transitions the device to the appropriate D*x* state.

## Disabling Runtime Idle Detection for Stop and Remove Requests

The stack’s function driver should disable runtime idle detection support when it processes the Query Remove or Query Stop request. This simplifies the remove or stop operation for the entire stack. The function driver can reenable runtime idle detection if the Query Stop or Query Remove request is canceled or if the device is powered up again.

The following example shows the consequences of not disabling runtime idle detection when the function driver handles the Query Stop or Query Remove requests. The example assumes a stack that consists of an upper filter driver above a function driver that is the PPO:

1. Both drivers have received and processed the Query Remove or Query Stop request, but runtime idle detection remains enabled.

2. The filter driver receives a Remove or Stop request and begins to fail all subsequent IRPs instead of passing them down the stack.

3. Just before the filter driver passes the Remove or Stop request down the stack to the function driver, the idle timer expires and the function driver requests a D*x* IRP to initiate the power-down process.

4. The D*x* IRP is first sent to the filter driver, which fails it. The IRP returns to the power manager and never reaches the function driver.

5. The function driver’s power policy state machine—which is waiting for the arrival of the D*x* IRP—stalls indefinitely.

If the function driver disables runtime idle detection support when it processes the Query Remove or Query Stop request, the drivers above the PPO passes a D*x* IRP or an optional wait/wake IRP down the stack to the function driver instead of failing the IRP and stalling the stack.

#### How does the driver handle surprise removal requests?

If the driver requested a wait/wake IRP, it must cancel the IRP when it receives a surprise-removal request. Because a surprise-removal request does not necessarily mean that the device has been removed, the function driver should try to power up the device, disarm wake, and power down the device again.

#### Does your bus driver serve as a PPO?

For bus drivers that also serve as PPOs, implementing runtime idle detection is similar in principle to implementing runtime idle detection in a function driver. However, bus drivers must manage one or more child physical device objects (PDOs)—one for each attached device. Therefore, the runtime idle detection implementation must consider the power state of the child devices before powering down the parent device.

Coordinating the power state of the parent device with the power states of all the child devices can be quite difficult and should be done carefully. In general:

* Keep the bus in D0 as long as even one of the child devices is powered up.
* If all the child devices are in D*x*, the bus driver can power down the bus.
* The idle time-out value for the parent device should be very short, perhaps zero.
* When a child device asks to be powered up by sending a D0 IRP to the bus driver, the bus driver must immediately power up the parent device.

The bus driver should pend the D0 IRP from the child device until the parent device is fully powered up and has reentered D0.

# Determining the Runtime Idle Detection Policy

This section contains questions that you should ask to help determine the settings—such as the idle time-out value—to use for your runtime idle detection implementation.

## Issues to Consider for All Devices

The questions in this section apply to all device classes.

#### When should the device wake itself?

Some possible scenarios include:

* The device does not wake itself at all, but depends on the function driver to decide when to wake the device and to initiate the wake process.
* The device wakes itself if the system is in S*x*, but not when the system is in S0.
* The device wakes itself only under certain device-specific circumstances.

For example, a network adapter could be configured to wake itself from S0 when any packet arrives. However, if the system is in S*x*, the network adapter wakes itself only when it receives a Magic Packet.

The function driver might be required to determine the device’s wake support at runtime, before it powers down the device. A driver can obtain some generic power capabilities by querying the associated bus driver. However, for device-specific wake capabilities, the function driver might be required to query the device. Using the network adapter discussed earlier as an example, the function driver can query the bus to try to determine whether the network adapter supports wake from S0 or S*x*. However, the driver might be required to query the network adapter itself to determine what types of wake packets it responds to.

#### Can the device idle itself out?

For example, some laptop internal hard drives power themselves down—perhaps with help from the firmware—when they determine that they have been idle for too long. If a device can idle itself out, issues to consider include the following:

* Is the function driver required to enable the idle-out capability?
* Is the idle-out operation transparent to the function driver, or must the driver participate in the process?
* Does the device idle itself out by notifying the function driver, but let the driver handle the power-down operation?

#### Is wake from S0 different from wake from S*x*?

For example, a network adapter might wake itself when a cable is inserted or when it receives a Magic Packet. However, a Magic Packet wakes the network adapter from any system power state, but cable insertion wakes the network adapter only from S0.

If a device that has separate wake scenarios is in the S0 D*x* state when an S*x* IRP arrives, the function driver must do the following:

1. Wake the device.

2. Disarm wake from S0.

3. Rearm the device for wake from S*x*.

4. Transition the device to the D*x* state that is appropriate for S*x*.

#### What is the correct D*x* state when the device idles out?

Generally, the lower the D*x* state, the longer it takes for the device to wake, but this behavior depends heavily on the underlying bus. Guidelines include the following:

* Unless a short wake time is critical or the user is likely to notice the difference, use the lowest state from which the device can wake itself.

The bus driver reports the lowest state from which the device can wake itself in the DEVICE\_CAPABILITIES structure’s **WakeFromDx** members.

* If a short wake time is important, carefully consider which D*x* state is most appropriate.

The wake latencies are reported in the DEVICE\_CAPABILITIES structure’s **DxLatency** members.

For USB devices, D1 through D3 are software constructs and have essentially the same wake latency. For Windows XP, D2 is the level that can be armed for wake. However, for Windows Vista, a device in D3 can also be armed for wake.

#### Should the user be able to enable or disable runtime idle detection?

Several issues might make it desirable to let the user control the runtime idle detection setting, including the following:

* The device firmware might have bugs in the power-up/power-down path.
* The wake time is long enough to be perceptible to the user.
* Powering down the device could have undesirable side-effects.

For example, a USB-connected cellular phone might use current from the USB cable to charge the battery. Turning the device off after a short interval of inactivity prevents the phone from charging correctly.

To let the user control runtime idle detection for a device, the function driver must implement support for GUID\_POWER\_DEVICE\_ENABLE, which is the WMI globally unique identifier (GUID). This adds the **Power Management** tab to the device’s **Properties** dialog box, with a check box that the user can set to enable or disable runtime idle detection.

#### Should the device have multiple runtime idle detection policies?

It might be useful to have different runtime idle detection policies, depending on the circumstances. For example:

* When the device is connected to a laptop that is running on AC power, the function driver disables runtime idle detection.
* When the laptop is running on battery power, the function driver enables runtime idle detection and aggressively sets the idle time-out, especially when battery power is low.

To support multiple runtime idle detection policies, a function driver must be able to detect when the computer’s power state has changed and use that information to enable the appropriate policy. Power state changes can be detected in one of these ways:

* Use **ExCreateCallback** to create a \Callback\PowerState object. This object can notify a driver of some basic power state changes, including:

The system has switched from AC to battery power, or vice versa.

A user application has changed the system power policy.

The system is about to transition to S*x*.

**Note:** The \Callback\PowerState notification is sent before the S*x* IRP.

* Use Windows Vista power profile change notifications.

Windows Vista provides some additional power profile change notifications. For information, see “Managing Device Performance States” in the WDK documentation.

Windows does not have a generic idle time-out value that applies to all device classes. You must determine the appropriate time-out value for your device, depending on its usage. Typically, the time-out value for a particular runtime idle detection policy is fixed. However, drivers sometimes modify the time-out value, based on factors such as the following:

* Usage history
* Remaining battery life
* The current power source

For example, you might want to reduce the idle time-out value when the battery power begins to be low.

## Issues to Consider for USB Devices

The questions in this section are specific to USB devices.

#### Should a USB function driver support runtime idle detection?

If your device connects to a USB port or if you have implemented a class driver that supports USB-connected devices, you should consider whether the driver should support runtime idle detection by implementing USB selective suspend. If your device or driver does not fall into either of these categories, you can skip this section.

The complete answer depends on the device and the version of Windows, as follows:

* If the device must be armed for wake, the function driver must support USB selective suspend, regardless of the Windows version.
* If the device is not required to be armed for wake, the answer depends on the Windows version, as follows:

For Windows versions earlier than Windows Vista, the function driver must support USB selective suspend. This requirement derives from how the USB core turns off devices, not from hardware limitations.

For Windows Vista and later versions, the function driver is not required to support USB selective suspend. The driver can simply power down the device.

#### Is a wait/wake request sufficient? Or must the device be explicitly armed for wake?

With some USB devices, the WDM function driver must explicitly arm the device for wake before it sends the wait/wake IRP down the stack.

# Guidelines for Implementing Runtime Idle Detection

This section provides guidelines for implementing runtime idle detection. Optimizing power-up code is the primary issue, because that code is more likely than the power-down code to affect the user. Questions to consider include the following:

* Is the power-up code a linear sequence that proceeds synchronously, except perhaps for queuing a work item in the D0 IRP’s completion routine?
* Is the power-up code broken into multiple pieces—completion routines, interrupt service routines (ISRs), deferred procedure calls (DPCs), and so on—that proceed asynchronously?
* Is the device marked as POWER\_PAGABLE? This does not affect runtime idle detection functionality.

All these factors affect how a function driver powers up a device and processes pended I/O requests. The following sections discuss how a function driver should implement runtime idle detection for two driver models: a KMDF driver and a WDM driver. The final section discusses how to test your runtime idle detection implementation.

## Adding Runtime Idle Detection to a KMDF Function Driver

To implement runtime idle detection in a KMDF function driver:

* Create and configure framework queue objects (WDFQUEUE) to appropriately handle power requirements.

Typically, a KMDF function driver uses a non–power-managed queue to handle incoming device I/O control requests and forwards any power-dependent requests to a power-managed queue.

* In the *EvtDriverDeviceAdd* callback, initialize a WDF\_DEVICE\_POWER\_POLICY  
  \_IDLE\_SETTINGS structure and use it to specify idle settings, including the following:

Whether the device can wake from S0.

Whether the device supports USB selective suspend.

Which D*x* state the device enters when it is powered down.

The idle time-out value.

Whether the user can modify the device’s idle settings.

Whether the device supports runtime idle detection.

Pass the structure to **WdfDeviceAssignS0IdleSettings** to specify the idle policy.

* Implement *EvtDeviceD0Entry* and *EvtDeviceD0Exit* to handle device power up and power down, respectively.
* For KMDF bus drivers, configure idle settings for the parent and rely on the child stacks to handle their own idle operation. All children must be powered down before the parent can do so.

**Note:** If a function driver uses a continuous reader on a USB endpoint, the reader’s repeated polling does not count as activity as far as the KMDF idle timer is concerned. If no data is actually read, the device is considered idle and is powered down when the idle timer expires.

Several KMDF samples provided in the WDK implement runtime idle detection, including:

* Featured Toaster

This sample shows how to implement runtime idle detection in a KMDF function driver. Featured toaster is located at %WinDDK%\*BuildNumber*\src\kmdf\toaster  
\func\featured.

* Osrusbfx2

This sample shows to implement USB selective suspend in a KMDF function driver. This sample is located at %WinDDK%\*BuildNumber*\src\kmdf\osrusbfx2  
\sys\final.

## Adding Runtime Idle Detection to a WDM Function Driver

If a WDM function driver does not currently support runtime idle detection, the following parts of the existing code might require modification:

* I/O dispatch routines.

Separate the incoming I/O requests into two categories based on whether the device must be in D0 to process the request. For those requests that require D0, the function driver must initiate the wake process and possibly pend the requests until the device has reentered D0.

* Power dispatch routine.

Function drivers that do not support runtime idle detection typically expect wait/wake and D*x* IRPs to arrive only when the driver has a pended S*x* IRP. Modify the logic to also handle these two IRPs if they arrive when the system is in S0.

* The driver’s lower edge.

The lower edge of some function drivers must communicate with the device outside the context of I/O request processing:

If the device is in D0, the driver should mark the device as not idle, so that the driver does not start the idle timer.

If the device is in D*x*, the driver should wake the device and perhaps wait until the device has reentered D0 before completing the action.

In addition to modifying existing code, you must add code to do the following:

* Maintain the current idle state.
* Manage the idle timer state, including starting and perhaps canceling the idle timer as required.
* Request the appropriate power IRPs when the idle timer expires.
* Power up the device when it is time to return it to D0.
* Pend power-dependent I/O requests when the device is in D*x* and process the requests when the device reenters D0.

Generally, USB selective suspend is quite difficult to implement correctly in a WDM driver and no samples exist that show how to do this. A much better solution is to convert your WDM driver to KMDF so that you can take advantage of KMDF’s USB selective suspend support.

## Testing Runtime Idle Detection Implementations

As discussed earlier, adding support for runtime idle detection to a function driver creates the potential for several new race conditions that must be properly handled. The following are some suggestions to help you test and validate your runtime idle detection implementation:

* Set the idle time-out to a very small value, perhaps 50 milliseconds or so. Then send power-dependent I/O requests at approximately the same interval, plus or minus some random variability. The requests should arrive so that they bracket the expiration of the idle timer.

This test should cover two cases:

The attempt to cancel the timer succeeds.

The attempt to cancel the timer occurs during the power-down process.

* Set the idle time-out to a very small value, perhaps 50 milliseconds or so. Write a Microsoft® Visual Basic® Script Edition (VBScript) application that uses WMI to repeatedly enable and disable runtime idle detection at about the same time interval, plus or minus some random variability.

This test covers the same two cases as the previous example, but exercises a different code path.

* Suspend and resume the computer while the device is idled out.
* If the device can be unplugged while the computer is running—such as a USB device—unplug the device in these cases:

When it is idle.

After the idle timer has expired but while the function driver remains in the power-down process.

* Use Device Manager to disable the device when it is idle.
* Run the Driver Test Manager (DTM) Plug and Play and power tests—especially Pnpdtest—against the function driver. The DTM tests are provided in the Windows Logo Kit.
* If the function driver must support USB selective suspend, use the device simulation framework (DSF) to create an emulator for the device.

You can use that emulator to control when the USB selective suspend callback is called. For information about DSF, see “Device Simulation Framework for USB Devices” in the WDK documentation.

# Resources

The following links provide more information on runtime idle detection and related topics.

#### Windows Driver Kit Documentation

Device Simulation Framework for USB Devices

<http://msdn2.microsoft.com/en-us/library/aa972916.aspx>

ExCreateCallback

<http://msdn2.microsoft.com/en-us/library/aa489845.aspx>

HIDClass Devices

<http://msdn2.microsoft.com/en-us/library/ms789872.aspx>

Managing Device Performance States

<http://msdn2.microsoft.com/en-us/library/aa489894.aspx>

Implementing Power Management

<http://msdn2.microsoft.com/en-us/library/ms798306.aspx>

#### WHDC Web White Papers and Driver Tips

Plug and Play and Power Management in WDF Drivers

<http://www.microsoft.com/whdc/driver/wdf/WDF_pnpPower.mspx>

Power Management Best Practices

[http://www.microsoft.com/whdc/system/pnppwr/powermgmt/default.mspx](http://www.microsoft.com/whdc/system/pnppwr/powermgmt/default.mspx%20)

Windows Vista and Power Development home page

<http://www.microsoft.com/whdc/DevTools/WDK/vista_dev.mspx>

Selective Suspend in USB Drivers

<http://www.microsoft.com/whdc/driver/wdf/USB_select-susp.mspx>

When WDF Drivers Can Use Power-Managed I/O Queues

<http://www.microsoft.com/whdc/driver/tips/pow-mnged_queues.mspx>

Windows Vista Energy Conservation

<http://www.microsoft.com/whdc/system/pnppwr/powermgmt/VistaEnergyConserv.mspx>

Writing USB Drivers with WDF

<http://www.microsoft.com/whdc/driver/wdf/USB_WDF.mspx>

#### UMDF Blog Articles

How to enable USB selective suspend and system wake in the UMDF driver for a USB device (Peter Wieland’s Blog)

<http://blogs.msdn.com/peterwie/default.aspx>