

Architecture of the Kernel-Mode Driver Framework

September 12, 2006

原稿位址: <http://www.microsoft.com/whdc/driver/wdf/KMDF-arch.mspix>

歡迎任意複製, 散佈 但必須注意下面幾點:

1. 因為我的知識有限, 所以翻譯的內容可能有誤
2. 請加註來源以及 翻譯者

井民全 翻譯

Email: mqJing@msn.com

程式設計心得網站: <http://mqjing.twbbs.org/~ching/Course/course.htm>

Abstract

這份文章提供了有關 kernel-mode 驅動程式框架 (KMDF) 的資訊, KMDF 是 Windows 系列, 視窗驅動程式基礎 (Windows Driver Foundation, WDF) 的一部份. WDF 是一個新的驅動程式建構模型. 而 KMDF 支援建構遵循它的 kernel-model 驅動程式. 這篇文章描述了 KMDF 的架構以及那些型態的驅動程式可以由 KMDF 來建構的資訊.

這些資訊可以被應用在下面的作業系統:

Microsoft Windows Vista™
Microsoft Windows Server® 2003
Microsoft Windows XP
Microsoft Windows 2000

這份文件目前的版本被管理在下面的位址:

<http://www.microsoft.com/whdc/driver/wdf/KMDF-arch.mspix>

這裡討論的參考與資源被詳細列在這篇文章的後面

內容

簡介.....	4
KMDF 支援的裝置.....	4
KMDF 元件.....	5
KMDF 驅動程式的架構.....	6
KMDF 與 WDM 驅動程式的比較.....	6
裝置物件與驅動程式角色.....	7
Filter 驅動程式和 Filter 裝置物件.....	8
Function 驅動程式與 Functional 裝置物件.....	8
Bus 驅動程式與實體裝置物件.....	9
Legacy 裝置驅動程式與控制裝置物件.....	10
KKMDF 物件模型.....	10
Methods, Properties, and Events.....	10
物件階層架構.....	11

物件屬性.....	14
物件相關內容.....	14
物件的建構以及刪除.....	15
KMDF I/O 模型.....	16
I/O Requests 處理器.....	17
建立 Request, 清除 Request 和 關閉 Request.....	17
讀, 寫, 裝置 I/O 控制, 和 內部裝置 I/O 控制 Requests.....	18
I/O 佇列.....	19
佇列與 電源管理.....	20
Dispatch 型態.....	21
I/O Request 物件.....	21
來自 I/O Requests 的接收 Buffers.....	22
送出 I/O Requests.....	22
I/O Targets.....	23
為 I/O Requests 建立 Buffers.....	24
取消 和 暫停(Suspended) Requests.....	24
Request Cancellation.....	24
Request Suspension.....	25
完整的 I/O Requests.....	26
自我管理的 I/O.....	26
存取 IRPs 和 WDM 架構.....	26
Plug and Play 和 電源管理 Request 處理器.....	27
裝置列舉和啟動.....	28
Function 或 Filter 裝置物件的啟動順序.....	28
實體裝置物件的啟動順序.....	30
裝置關機和移除.....	31
Function 或 Filter 裝置物件的關機和移除順序.....	31
實體裝置的關機和移除順序.....	33
Surprise 移除順序.....	34
WMI Request 處理器.....	36
同步的議題.....	37
同步的範圍.....	38
執行層級.....	39
鎖定.....	40
同步處理機制的互動.....	41
安全性.....	41
預設安全.....	41
參數驗證.....	42
UNICODE 字串.....	42
裝置命名技術.....	42
建構與除錯環境.....	42
安裝.....	43
版本和動態連結.....	43
資源.....	44

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.

© 2006 Microsoft Corporation. All rights reserved.

Microsoft, 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.

簡介

Kernel-mode 驅動程式框架 (KMDF) 為建構一核心模式驅動程式的基礎結構, 他提供了 C 語言的裝置驅動程式介面 (device driver interface DDI) 而且可以被用來建立 Microsoft® Windows® 2000 和以後版本的驅動程式. 本質上, 這個軟體框架為一個裝置驅動程式骨架可以被調整成為特殊用途的裝置驅動程式. KMDF 實作了程式碼處理來自一般驅動程式的 requirements. 驅動程式利用設定物件屬性, 註冊 callbacks 處理特殊事件的處理來調整這個驅動程式框架, 並且還可以包含特殊用途裝置專用的程式碼.

KMDF 提供了一個完整定義的物件模型和物件生命週期和記憶體配置的控制. 物件們以階層式的 parent/child 模型來組織. 重要的驅動程式資料結構由 KMDF 來管理已取代本身驅動程式來管理.

這份文章提供了一個 KMDF 特性與架構上的簡介, 提出了 KMDF 驅動程式的需求. 這篇文章假設你有基本的 Windows 作業系統以及 I/O 模型的知識.

Window 驅動程式基礎 (Windows Driver Foundataion, WDF) 也包含了 user-mode 驅動程式框架 (user-mode driver framework, UMDF). 若妳的裝置不需要處理中斷, 執行直接記憶體存取(DMA), 或需要其他 kernel-model 資源 例如: nonpaged pool memory, 那麼你應該考慮寫一個 user-model 的驅動程式, 而非 kernel-model. 你可以在資源這一節中的 "Introduction to the WDF User-Mode Driver Framework," 這篇文章中找到詳細的資訊.

KMDF 支援的裝置

KMDF 基本上是用來取代 Windows Driver Windows Driver Model (WDM). 除了 miniport models 之外, 最初的 KMDF 支援了大部分的 WDM 支援的裝置以及支援的裝置類別 (device classes). Table 1 列出了 KMDF 支援的項目.

Table 1. KMDF 支援的項目

裝置或裝置型態	存在的驅動程式模型	註解
控制驅動程式 以及 non-Plug and Play 驅動程式	Legacy	支援
IEEE 1394 client 驅動程式	與裝置類別相關	只有支援那些不遵照存在裝置類別規格的硬體(Supported for devices that do not conform to existing device class specifications)
ISA, PCI, PCMCIA, and secure digital (SD) devices	WDM driver	支援, 但是僅限於那些不提供驅動程式 dispatch functions 的裝置.
NDIS protocol drivers	WDM upper edge and NDIS lower edge	支援
NDIS WDM drivers	NDIS upper edge and WDM lower edge	支援
SoftModem drivers	WDM driver with upper-edge support for TAPI interface	支援
Storage class drivers and filter drivers	WDM driver	支援

裝置或裝置型態	存在的驅動程式模型	註解
Transport driver interface (TDI) client drivers	Generic WDM driver	支援
USB client drivers	與裝置類別相關	支援
Winsock client drivers	WDM driver with a callback interface for device-specific requests	支援

只要遵循 WDM 的驅動程式，一般來說，KMDF 都會支援。KMDF 提供 major I/O dispatch 函式群的進入點以及處理 I/O request packets (IRPs)。

以一些裝置型態來說，他們的裝置類別以及 port 驅動程式支援了 driver dispatch functions 以及使用 callback 的方式，回到 miniport 驅動程式來處理特殊的 I/O 細節。對於這些 miniport 驅動程式為主的 callback 程式庫，他們目前並沒有被 KMDF 所支援。另外，WDMDF 並不支援那些使用 Windows image architecture (WIA) 的裝置型態。

KMDF 元件群

KMDF 為視窗驅動程式系列工具(Windows Driver Kit, WDK) 的一部份並擁有一致的 header 檔, libraries, sample drivers, 開發工具組, public debugging symbols, 和 tracing format 檔。在預設的情況下，KMDF 被安裝在 WDK 安裝根目錄中的 WDF 子目錄中。KMDF-based 驅動程式可以被 WDK 建構環境下被建製。Table 2 列出了 被安裝為 WDF 部分的 KMDF 元件群

Table 2. KMDF 元件群

元件	放置位置	描述
Header files	wdf/inc	用來建構 KMDF 驅動程式的必要標頭檔
Libraries	wdf/lib	包含了 x86, x64 以及 Intel Itanium 架構的程式庫
Sample drivers	wdf/src	一堆裝置型態的驅動程式樣本群；其中大部分是從 WDM 樣本群中轉換過來的。
Tools	wdf/bin	包含了測試,除錯,以及安裝驅動程式的工具集合。其中包含了 redistributable KMDF co-installer , WdfCoinstaller nn.dll
Debugging symbols	wdf/symbols	Public symbol database (.pdb) 檔案群，這些檔案將被 KMDF libraries 和 co-installer 將利用這些檔案來做檢查工作與建構使用。
Tracing format files	wdf/tracing	追蹤格式檔案群 將被 KMDF libraries 與 co-installer 用來 產生除錯追蹤訊息

為了除錯方便，KMDF 被免費的方式散佈而且那些建製好的 run-time libraries 與 loader 將包含著相對應的 symbols。然而，Microsoft 並不提供 **checked version** 的 **redistributable co-installer**。

KMDF 驅動程式的結構

一個 KMDF 驅動程式包含了一個 **DriverEntry** 函式，表明了這個驅動程式是以 KMDF 為基礎的所撰寫的。KMDF 會呼叫一系列的 **callback** 函式，使得該驅動程式能夠回應硬體裝置所產生的事件或者是回應其他驅動程式相關的工具函式。幾乎每一個 KMDF 驅動程式都必須遵守下面規則：

- **DriverEntry** 函式是驅動程式的主要進入點。
- 當 Plug and Play manager 在列舉系統中的裝置時，將會呼叫 **EvtDriverDeviceAdd callback** (如果某一個驅動程式支援的硬體為 non-Plug and Play devices, 則不會呼叫這個 callback)。
- **EvtIo*** **callbacks**, 專門負責處理，來自特定佇列中的特殊 I/O requests 型態。

驅動程式基本上會建立一個以上的佇列以提供 KMDF 放置 I/O requests。一個驅動程式能夠藉由指定 **request type** 與 **dispatching type** 來調整他所管理的佇列。更多的細節，請看後面 KMDF I/O 這個章節。

對一個簡單的硬體裝置，最精簡的 **kernel-mode** 驅動程式不多不少也必須擁有上述函式。KMDF 已經包含了處理預設的電源管理程式碼以及 **Plug and Play operations**，所以呢，驅動程式本身不需要去管理實際的硬體，也因此省去了那些管理程式碼。當一個使用預設管理的程式碼後，就可以省去許多共同的工作，例如丟出一個 **power IRP down** 到裝置堆疊中。越多的特殊裝置特徵，則驅動程式就必須提供越多的功能，當然驅動程式所需要的程式碼也就越多。

KMDF 與 WDM 驅動程式的比較

KMDF 模型讓驅動程式比 WDM 驅動程式更簡化和更容易的進行除錯。KMDF 驅動程式需要少量的共用程式碼進行處理預設管理動作。而這些**共用程式碼被放在 KMDF 的軟體 framework 中。這些程式碼被完整的測試過而且可以被全球地更新。**

KMDF events 被定義的較為清楚以及侷限，所以 KMDF 為基礎的驅動程式基本上，會有比較少的程式複雜度。一個 **callback routine** 執行一個特殊的工作。因此相對於 WDM 驅動程式來說，KMDF 為基礎的驅動程式會有較少的程式碼而且基本上沒有記錄狀態的變數或鎖定。

就像之前在 WDF 上的努力，Microsoft 已經將許多樣本驅動程式進行 WDM 至 KMDF 的轉換，一起包在 Windows DDK 中了。沒有意外的，KMDF 驅動程式確實比較小而且比較簡單。

Table 3 顯示了一些 PCIDRV, Serial 和 OSRUSBFX2 等驅動程式，在使用 KMDF 模型後的統計數據

Table 3. WDM-KMDF 對於樣本驅動程式的統計數據

Statistic	PCIDRV ¹		Serial ²		OSRUSBFX2 ³	
	WDM	KMDF	WDM	KMDF	WDM	KMDF
程式碼總行數	13,147	7,271	24,000	17,000	16,350	2,300
用來處理 Plug and Play 和電源管理的程式碼數量 (lines)	7,991	1,795	5,000	2,500	8,700	742
使用 Locks 和 synchronization 資源數量	8	3	10	0	9	0
狀態變數數量 Plug and Play 和 電源管理	30	0	53	0	21	0

¹The PCIDRV sample supports the Intel E100B NIC card. The WDM and KMDF versions are functionally equivalent.

²The Serial sample supports a serial device. In this case, the WDM sample supports a multiport device, but the KMDF sample supports only a single port. However, the statistics for the WDM driver do not include code, locks, or variables that are required solely to support multiport devices, so the statistics are comparable.

³The OSRUSBFX2 sample supports the USB-FX2 board built by OSR. The WDM and KMDF versions are functionally equivalent. The WDM version is available at <http://www.osronline.com>.

從表格看來，將驅動程式由 WDM 轉換至 KMDF 將導致大量的程式碼數量的減少—尤其是 Plug and Play 與 電源管理的部分，特別明顯。而 KMDF 樣本程式也使用較少的 locks, 同步機制 與 狀態變數。

- **程式碼行數**.. KMDF 驅動程式需要較少的程式碼進行實作 Plug and Play 和 電源管理的動作上. 較少的程式碼表示較低的複雜度與較少的錯誤和較小的 executable image.
- **Locks 和 synchronization 機制**. KMDF 驅動程式不止比較小, 而且所有三種 locks 和同步機制都已經被一定程度的降低使用. 這個改變很重要, 因為它消除了一項驅動程式共通的問題來源. WDM 驅動程式使用 lock 來同步化 I/O 佇列 搭配著 Plug and Play 和 電源管理動作 並且常常提供 lock 來管理 I/O 的取消動作. 而 locking 將會發生一個或更多的競賽問題. 這種工作很難被正確地實作. KMDF 驅動程式可以使用較少的這類 locks 因為使用 framework 已經提供了 locking 的功能了.
- **狀態變數**. Plug and Play 和 電源管理所需要狀態變數的數量為計算該驅動程式的複雜度很好的量度方式. 一個 WDM 驅動程式由作業系統接收 Plug and Play 和 電源管理的 requests 形成的 IRPs, 當這類驅動程式接收了一個 Plug and Play 或 電源的 IRP, 則必須判斷決定目前的裝置狀態和系統狀態, 基於這兩個狀態來判斷要如何滿足 IRP 的需求. 由裝置堆疊一路傳來的 IRPs, 驅動程式必須立即進行處理. 重要的是, 一個 WDM 驅動程式必須保持一定數量的狀態細節包含裝置, Plug and Play 和 電源管理 等目前狀態的細節. 例如: 追蹤保持 WDM PCIDRV 樣本驅動程式需要 30 個變數, OSRUSBFX 樣本則需要 21 個變數. 而這三個種 KMDF 版本的驅動程式不需要任何狀態變數. KMDF 不需要自己管理那些資訊, 因為 framework 會幫我們做好它. KMDF Framework 實作了一個擴充的狀態機整合了 Plug and Play 和 電源管理動作 搭配 I/O 動作. 若有管理上的需要, 一個 KMDF 驅動程式提供了 callbacks function 可以被呼叫. 舉一個例子, 一個裝置驅動程式支援了 Wake-up 訊號 可以註冊一個 callback 來 arm 這個訊號. KMDF 可以在適當的時機呼叫這個 callback. 相對的, WDM 驅動程式則必須判斷哪一個 power management IRPs 需要來 arm 這個訊號, 哪些指標用來處理那些 IPRs.

裝置物件和驅動程式角色

每一個驅動程式建立一個或多個裝置物件用來表示驅動程式的角色在於管理 I/O requests 和管理它的裝置. KMDF 支援建構建構下面所列的裝置物件:

- **Filter device objects** (filter DOs) 扮演一個 filter driver 的角色. Filter Dos 過濾或者是更改一種或多種目標為該裝置的 I/O request 型態. Filter DOs 被貼到 Plug and Play 裝置堆疊中.
- **Functional device objects** (FDOs) 扮演為一個 function driver 的角色, 為一個裝置的主要驅動程式. FDOs 會被貼到 Plug and Play 裝置的堆疊中.
- **Physical device objects** (PDOs) 扮演著 Bus driver 的角色, 列舉出所有子裝置出來. PDOs 會被貼到 Plug and Play 裝置的堆疊中.

- **Control device objects** 扮演著 legacy non-Plug and Play 裝置或者是一項控制介面。他們並不屬於 Plug and Play 裝置堆疊的任何一部份。

根據硬體裝置和在裝置堆疊中的其他驅動程式的設計考量與關係，一個驅動程式應該被假設為好幾種角色。每一個 Plug and Play 裝置只能擁有 **function driver** 和一個 **bus driver**，但卻可擁有任意數量的 **filter drivers**。在 Plug and Play 裝置堆疊中，一個 driver 對一個裝置而言為 **function driver**，但是同一個 driver 對列舉裝置而言，卻視為 **bus driver**。

舉一個例子，USB hub driver 是 hub 本身的 **function driver**。對於每一個串接在那個 hub 上的硬體而言，USB hub driver 是他們的 **bus driver**。所以 USB hub driver 建立一個 FDO 給 hub，建立一個 PDO 給每一個串接在上面的 USB 硬體。

更多 KMDF 支援的有關驅動程式型態與裝置物件訊息，我們將在下一節詳述

Filter Drivers 和 Filter Device Objects

一個 **filter driver** 接收一個以上的 I/O requests。這些 requests 被目標至它的裝置。基於個別的 **request filter driver** 採取了一些動作，然後傳送該 request 到下一個在堆疊中的 driver 繼續處理。Filter drivers 並不單純地執行 device I/O 而已；他們更改或紀錄一個 request 使得另一個 driver 可以接受。特殊硬體專用的資料加密/資料解密軟體通常被實作為用 **filter driver**。

一個 **filter driver** 增加一個 **filter DO** 到 Plug and Play device stack 中。當它的硬體被加入系統中時，KMDF driver 通知 framework，它是一個 **filter driver**，所以 KMDF 建立了一個 **filter DO** 並且設定給與適當的預設值。

多數的 **filter driver** 對每一個目標至他們的 request 並不感興趣。一個 **filter driver** 可能濾掉唯讀或者是只有建立的 request。為了簡化實作 **filter driver** 的困難度，KMDF 只有分派一些特殊指定的 request 並且把所有其他的 request 直接略過，到下一個 device stack。Filter driver 永遠不會接收到他們，而且因此不需要寫碼專門處理它或者是把他傳到其他的 driver。

範例 Firefly, Kbfiltr, 與 Toaster Filter drivers 建立了 filter DOs 可供參考。

KMDF 並不支援你建立 **bus filter drivers**。這類 driver 立即直接地被分層在一個 bus driver 上面，當 Plug and Play manager 向 bus driver 查詢 bus relations 時，該 bus driver 會建立 PDO 並且把他們的 device objects 加到 stack 中。

Function Drivers 和 Functional Device Objects

Function drivers 是裝置的主要驅動程式。Function driver 藉由執行 I/O 和標準的電源管理機制來進行與裝置的通訊。在 Plug and Play device stack 中，function driver 是以 FDO 的身份出現。

為了支援 function drivers, KMDF 包含了一個 FDO 介面定義一系列的 method, events 與 properties 用來處理 FDO 的初始化動作與一系列的動作。該介面可以使得 driver 可以做下面的工作：

- 註冊一個 event callbacks 於他的裝置資源配置。

- 取出實體裝置的屬性.
- 開啓一個 registry key.
- 當裝置需要列舉 children 時, 進行管理 child devices 的串列.

所以當 driver 建立一個 device object 時, KMDf 自動會建立一個 FDO, 除非 driver 聲明它不要建立.

在預設的情況下, function driver 爲它負責 device 的電源策略管理器. 這個意思是如果目前的 device 支援 wake-up 訊號時, function driver 基本上也會設定電源策略事件 callbacks 功能. 以實現電源管理器的功能.

除了 KbFiltr 和 Firefly driver, 所有的範例 driver 都會建立一個 FDO.

Bus Drivers 和 Physical Device Objects

Bus driver 的工作就像是 function driver 對於 parent device 列舉一個以上的 child devices. Parent device 可能是一個 bus 但也可能是一個多功能的 device 列舉 children 它的功能需要多種型態的 drivers. 在 Plug and Play device stack 中, bus driver 以 PDO 呈現.

KMDf 定義了 methods, events, 和 properties 給 PDOs, 這樣的功能就如同 FDOs 也是一樣的. 我們可以使用 PDO interface 的操作, 來完成下面的工作:

- 註冊 event callbacks. 若它的 children 需要時, 可以使 driver 可以通報所管理的硬體資源.
- 註冊 和 device locking 和 ejection 的 event callbacks
- 註冊 event callbacks 以執行 bus-level 工作使得 child devices 可以觸發一個 wake signal.
- 指定 Plug and Play 相容和實例的 ID 給 child devices.
- 設定, 刪除和 ejection 指定的 child devices.
- 通知系統 child device 已經被 ejected 或者是 surprise-removed.
- 取出或 Retrieve and update the bus address of a child device.
- 指出目前的 driver 控制一個 raw device. (一個 raw device 可以直接被 bus driver 所驅動, 完全不需要 function driver.)

如果要指定一個 bus driver, 則 KMDf driver 必須在建立 device object 之前, 先呼叫 PDO 的初始化 method. 若 driver 指出他正在驅動一個 raw device, 那麼 KMDf 則假設它爲該裝置的 power policy manager.

撰寫一個 KMDf bus drivers 比起 WDM 的複雜度來說, 簡單很多. KMDf 幫你管理了裝置的 PDO 狀態. 使得 driver 只要在新裝置被加入時, 或者是裝置被移除時, 通知 KMDf 即可. 而且 KMDf 支援了靜態與動態兩種模式來列舉 child devices. 假設 child devices 的狀態很少改變, 那麼你應該選擇 靜態模式來撰寫 bus driver. 而像 IEEE 1394 buses 這種 child devices 可能隨時改變的 devices, 則應該使用動態模式來撰寫這種裝置的 bus driver.

對於 bus drivers 而言, KMDf 處理了許多有關列舉方面的細節, 這包含了下列項目:

- Reporting children to WDM.
- 協調合作的掃描 children.
- 管理 children list.

另一方面，因為 KMDF 介面是藉由 drivers 報告需求的資源，也因此比 WDM 提供的機制簡單。

在範例方面，[kbFiltr](#)，[OsrUsbFx2/ EnumSwitches](#)，和 [Toaster Bus drivers](#) 提供了建立 PDO 的範例，其中並且使用了靜態與動態的方法來列舉 child devices。

Legacy Device Drivers 和 Control Device Objects

除了 Plug and Play function 如 bus 和 filter drivers 之外，KMDF 同時也支援了非 Plug and Play 控制的 legacy 裝置。這類 drivers 建立了 *control device objects*，這些 objects 並不存在於 Plug and Play devices stack。

Plug and Play drivers 可以使用 control device objects 來實作控制介面來獨立地操作 device stack。應用程式可以直接發出 requests 給 control device object，藉此跳過任何在 stack 中 driver 的過濾程序。Control device object 基本上有一個唯一的 queue 並且將這個 queue 轉送給 Plug and Play device object。

因為 control device objects 並不屬於 Plug and Play device stack 的成員，所以當初始化動作完成時，你(driver) 必須要自己通知 KMDF 這件事情。還有一件事要注意的，那就是當裝置被移除時，你(driver) 必須要刪除對應的 device object。因為此時，只有你才知道這類裝置的是否存在於系統中。

範例方面：NdisProt, NonPnP, 和 Toaster Filter drivers 提供建立 control device objects 的範例。

KMDF Object Model (物件模型)

KMDF 定義了一個以物件為基礎的程式模型。在這個模型中，物件型態代表著驅動程式的共同建構子。每一個物件提供了 methods 和 property。利用這些物件的屬性，驅動程式可以存取或者藉由 event callbacks 提供物件相關的訊息。The objects themselves are opaque to the driver.

KMDF 建立了一些物件來處理 driver 的基本行為，而 driver 也可以建立其他的物件來提供特殊的需求動作。Driver 必須為 events 提供 callbacks 給 KMDF，藉以提供 KMDF 在預設中沒有設定的功能。在必要時 driver 也可以呼叫物件上的 method 來取得或者是設定物件的屬性，以執行額外的的工作。因此呢，一個由 KMDF 架構所寫成的驅動程式，基本上是一個 **DriverEntry 副程式**，一堆執行裝置特殊工作的 callback 函式和一些有用的函式的集合體。

Framework-based 的 driver 並不會直接存取 framework 物件的 instances。Driver 存取 framework 物件 instances 使用的是 handles。Driver 將 handle 當作參數傳到物件的 method 中而 KMDF 則將 handle 傳到 event callbacks 中。Framework 物件對於 framework 而言是唯一的，也就是說他們不被 Windows object manager 所管理，同時也不被任何系統的 ObXxx 函式所管理。只有 framework 本身以及他的驅動程式可以建構或操作 framework 物件。Windows Object Manager 不會插手管理 framework 物件。

Methods, Properties, and Events

Method 為一個函式，專門用來對一個物件執行某種動作。例如建構或刪除一個物件。KMDF methods 利用下面的規則命名 Method:

WdfObjectOperation

其中 *Object* 標明了這個 KMDF 物件中的 *method* 而 *Operation* 則指出這個 *method* 作了什麼。舉例來說, **WdfDeviceCreate** 建立一個 framework device object.

Properties 為讀寫物件資料成員的函式。因此 *properties* 定義了物件的行為與預設資料。*Properties* 的命名規則如下:

WdfObject{Set|Get}Data

WdfObject{Assign|Retrieve}Data

其中 *Object* 指定了哪一個 KMDF object 上的 *function* 動作, 而 *field* 則指定了這個 *function* 要存取的屬性。某些 *propertyes* 可以順利地讀取或寫入資料, 有些則不一定。使用 *Set/Get* 的命名方式, 則專門用於可以順利存取欄位資料的情況。其中 **Set** functions 傳回 VOID 而 **Get** functions 基本上傳回的是該欄位的資料。有 **Assign** 和 **Retrieve** 字樣的 *function* 則專門用於可以存取欄位資料, 但是卻可能會失敗的情況。這時他們的傳回值是 NTSTATUS.

舉個例子來說, WDFINTERRUPT 物件是一個裝置中斷(interrupt) 物件。中斷物件是由一堆特性所描述。該組特性指出了中斷的型態 (message-signaled 或 IRQ-baed) 和提供相關資訊。你可以呼叫 **WdfInterruptGetInfo** method 傳回中斷資訊。要注意的是: 當你建構某一個中斷物件並設定初始值後, 就無法再更改中斷特性的資料了。因此在 device 運作過程中, driver 只能讀取中斷資料而無法設定資料。這點要特別注意。

事件(event) 可以表達 driver 在執行時(run-time)對外界的反應。Driver 可以針對他感興趣的 event 註冊 callbacks。當 event 發生時, framework 會自動呼叫該 callback, 傳送一個 handle 給註冊該 event 的物件。舉例來說, 若你的裝置提供了 *ejection* 的功能, 而 driver 也註冊了 *EvtDeviceEject* callback routine, 則當 Plug and Play manager 送出了 IRP_MN_EJECT request 時, KMDF 就會呼叫 *EvtDeviceEject* 這個 routine。當然了, 如果裝置不支援 *ejected*, 那麼你就不需要這類的 callback。

對於許多的 events 來說, driver 可以選擇提供自己的 callback routine 或者是讓 KMDF 執行預設的行為來處理這類 events 的對應關係, 但是對於少數特殊的 event, driver 則必須提供特殊的 callback。"增加一個裝置 event" 就是一個例子, 每一個 Plug and Play driver 必須提供這個 callback 來對應這個 event。Driver 的 **EvtDriverDeviceAdd** callback 建立了新增裝置的 device object 以及一系列裝置屬性。這些東西你必須自己給定, 沒辦法由 KMDF 預設幫你設定。

另外, KMDF events 和 Kernel-dispatcher 的 events 是不同的。對於系統提供用來作同步的 event, 同步機制的標準動作如 *create*, *wait event* 則不能用在 KMDF events 上, 你只能註冊 event 的 callback 並且等待 KMDF 呼叫你。這點要分清楚。

物件階層 (Object Hierarchy)

KMDF 物件是有階層地被組織起來的。其中 WDFDRIVER 是最頂端的根物件; 其餘的物件都是它的子物件。許多物件型態可以在建立之初, 就指定它的父物件。如果你沒有指定的話, KMDF 會幫你預設 WDFDRIVER 為父物件。

Figure 1 shows the default KMDF object hierarchy.

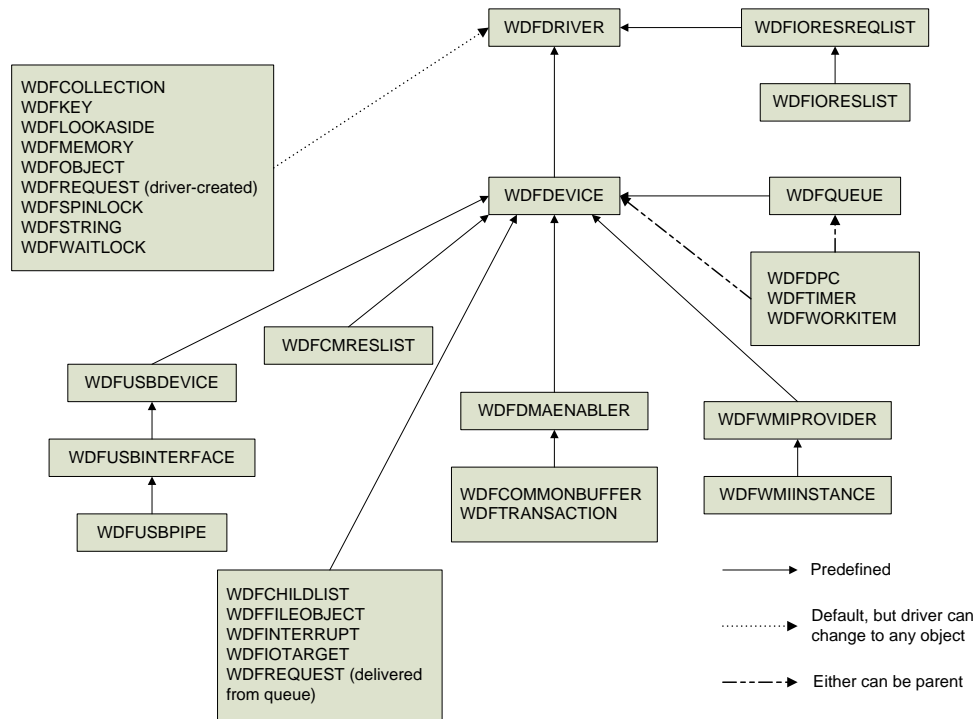


Figure 1. Parent-Child Relationships among the KMDF Objects

For each object, the figure shows which other object(s) must be in its parent chain. These objects are not necessarily the immediate parent but could be the grandparent, great-grandparent, and so forth. For example, the figure shows the WDFDEVICE object as parent of the WDFQUEUE object. However, a WDFQUEUE object could be the child of a WDFIOTARGET object, which in turn is the child of a WDFDEVICE object. Thus, the WDFDEVICE object is in the parent chain for the WDFQUEUE object.

The object hierarchy affects the object’s lifetime. The parent holds a reference count for each child object. When the parent object is deleted, the child objects are deleted and their callbacks are invoked in a defined order. For details, see "Object Creation and Deletion" later in this paper.

Table 4 lists all the KMDF object types.

Table 4. KMDF Object Types

Object	Type	Description
Child list	WDFCHILDLIST	Represents a list of the child devices for a device.
Collection	WDFCOLLECTION	Describes a list of similar objects, such as resources or the devices for which a filter driver filters requests.
Device	WDFDEVICE	Represents an instance of a device. A driver typically has one WDFDEVICE object for each device that it controls.
DMA common buffer	WDFCOMMONBUFFER	Represents a buffer that can be accessed by both the device and the driver to perform DMA.

Object	Type	Description
DMA enabler	WDFDMAENABLER	Enables a driver to use DMA. A driver that handles device I/O operations has one WDFDMAENABLER object for each DMA channel within the device.
DMA transaction	WDFDMATRANSACTION	Represents a single DMA transaction.
Deferred procedure call (DPC)	WDFDPC	Represents a deferred procedure call.
Driver	WDFDRIVER	Represents the driver itself and maintains information about the driver, such as its entry points. Every driver has one WDFDRIVER object.
File	WDFFILEOBJECT	Represents a file object through which external drivers or applications can access the device.
Generic object	WDFOBJECT	Represents a generic object for use as the driver requires.
I/O queue	WDFQUEUE	Represents an I/O queue. A driver can have any number of WDFIOQUEUE objects.
I/O request	WDFREQUEST	Represents a request for device I/O.
I/O target	WDFIOTARGET	Represents a device stack to which the driver is forwarding an I/O request.
Interrupt	WDFINTERRUPT	Represents a device's interrupt object. Any driver that handles device interrupts has one WDFINTERRUPT object for each IRQ or message-signaled interrupt (MSI) that the device can trigger.
Look-aside list	WDFLOOKASIDE	Represents a dynamically sized list of identical buffers that are allocated from the paged or nonpaged pool. Both the WDFLOOKASIDE object and its component memory buffers can have attributes, as described in "Object Attributes" later in this paper.
Memory	WDFMEMORY	Represents memory that the driver uses, typically an input or output buffer that is associated with an I/O request.
Registry key	WDFKEY	Represents a registry key.
Resource list	WDFCMRESLIST	Represents the list of resources that have actually been assigned to the device.
Resource range list	WDFIORESLIST	Represents a possible configuration for a device.
Resource requirements list	WDFIORESREQLIST	Represents a set of I/O resource lists, which comprises all possible configurations for the device. Each element of the list is a WDFIORESLIST object.
String	WDFSTRING	Represents a counted Unicode string.
Synchronization: spin lock	WDFSPINLOCK	Represents a spin lock, which synchronizes access to data DISPATCH_LEVEL.
Synchronization: wait lock	WDFWAITLOCK	Represents a wait lock, which synchronizes access to data at PASSIVE_LEVEL.
Timer	WDFTIMER	Represents a timer that fires either once or periodically and causes a callback routine to run.
USB device	WDFUSBDEVICE	Represents a USB device.

Object	Type	Description
USB interface	WDFUSBINTERFACE	Represents an interface on a USB device.
USB pipe	WDFUSBPIPE	Represents a pipe in a USB interface.
Windows Management Instrumentation (WMI) instance	WDFWMIINSTANCE	Represents an individual WMI data block that is associated with a particular provider.
WMI provider	WDFWMI PROVIDER	Represents the schema for WMI data blocks that the driver provides.
Work item	WDFWORKITEM	Represents a work item, which runs in a system thread at PASSIVE_LEVEL.

Object Attributes

Every KMDF object is associated with a set of attributes. The attributes define information that KMDF requires for objects, as listed in Table 5.

Table 5. KMDF Object Attributes

Field	Description
ContextSizeOverride	Size of the context area; overrides the value in ContextTypeInfo->ContextSize . Useful for context areas that have variable sizes.
ContextTypeInfo	Pointer to the type information for the object context area.
EvtCleanupCallback	Pointer to a callback routine that is invoked to clean up the object before it is deleted; the object might still have references.
EvtDestroyCallback	Pointer to a callback routine that is invoked when the reference count reaches zero for an object that is marked for deletion.
ExecutionLevel	Maximum interrupt request level (IRQL) at which KMDF can invoke certain object callbacks.
ParentObject	Handle to the object's parent.
Size	Size of the object.
SynchronizationScope	Level at which certain callbacks for this object are synchronized; applies only to driver, device, and file objects.

The framework supplies defaults for most attributes. A driver can override these defaults when it creates the object by using the `WDF_OBJECT_ATTRIBUTES_INIT` function.

Object Context

Every instance of a KMDF object can have one or more object context areas. This area is a driver-defined storage area for data that is related to a specific instance of an object, such as a driver-allocated lock or event for the object. The size and layout of the object context area are determined by the driver. When the driver creates the object, it initializes the context area and specifies its size and type. The driver can create additional context areas after the object has been created. For a KMDF device object, the object context area is the equivalent of the WDM device extension.

When KMDF creates the object, it allocates memory for the context areas from the nonpaged pool and initializes them according to the driver's specifications. When KMDF deletes the object, it deletes the context areas, too. The framework provides macros to associate a type and a name with the context area and to create a named accessor function that returns a pointer to the context area.

If you are familiar with WDM, this design might seem unnecessarily complicated. However, it provides flexibility in attaching information to I/O requests as they flow

through the driver. In addition, it enables different libraries to have their own separate context for an object. For example, an IEEE 1394 library could track a WDFDEVICE object at the same time that the device's function driver tracks it, but with separate contexts. Within a driver, the context area enables a design pattern that is similar to inheritance. If the driver uses a request for several different tasks, the request object can have a separate context area to each task. Functions that are related to a specific task can access their own contexts and do not require any information about the existence or contents of any other contexts.

Object Creation and Deletion

To create an object, KMDF:

- Allocates memory from the nonpaged pool for the object and its context areas.
- Initializes the object's attributes with default values and the driver's specifications (if any).
- Zeroes the object's context areas.
- Configures the object by storing pointers to its event callbacks and setting other object-specific characteristics.

If object initialization fails, KMDF deletes the object and any children that have already been created.

To initialize object attributes and configuration structures, a driver invokes KMDF initialization functions before it calls the object-creation methods. KMDF uses the initialized attributes and structures when it creates the object.

KMDF maintains a reference count for each object and ensures that the object persists until all references to it have been released. If the driver explicitly deletes an object (by calling a deletion method), KMDF marks the object for deletion but does not physically delete it until its reference count reaches zero.

Drivers do not typically take out references on the objects that they create, but in some cases (such as when escaping directly to WDM) such references are necessary to ensure that the object's handle remains valid. For example, a driver that sends asynchronous I/O requests might take out a reference on the request object to guard against race conditions during cancellation. Before the request object can be deleted, the driver must release this reference.

Object deletion starts from the object farthest from the parent and works up the object hierarchy towards the root. KMDF takes the following steps to delete an object:

1. Starting with the child object farthest from the parent, calls the object's *EvtCleanupCallback*. In this routine, drivers should perform any clean-up tasks that must be done before the object's parent is deleted. Such tasks might include releasing explicit references on the object or a parent object. Note that when the *EvtCleanupCallback* function runs, the object's children still exist, even though their *EvtCleanupCallback* functions have already been invoked.
2. When the object's reference count reaches zero, calls the object's *EvtDestroyCallback*, if the driver has registered one.
3. Deallocates the memory that was allocated to the object and its context area.

KMDF always calls the *EvtCleanupCallback* routines of child objects before calling those of their parent objects, so drivers are guaranteed that the parent object still exists when a child's *EvtCleanupCallback* routine runs. This guarantee does not

apply to *EvtDestroyCallbacks*, however; KMDF can call the *EvtDestroyCallback* routines in any order, so that the *EvtDestroyCallback* for a parent might be called before that of one of its children.

Drivers can change the parent of most KMDF objects by setting the **ParentObject** attribute. By setting the parent/child relationships appropriately, a driver can avoid taking out explicit references on related objects and can instead use the hierarchy and the associated callbacks to manage the object's lifetime.

KMDF I/O Model

KMDF establishes its own dispatch routines that intercept all IRPs that are sent to the driver.

Figure 2 shows the overall flow of I/O through the KMDF library and driver. For a detailed explanation, see "I/O Request Flow in WDF Kernel-Mode Drivers", which is listed in the Resources section.

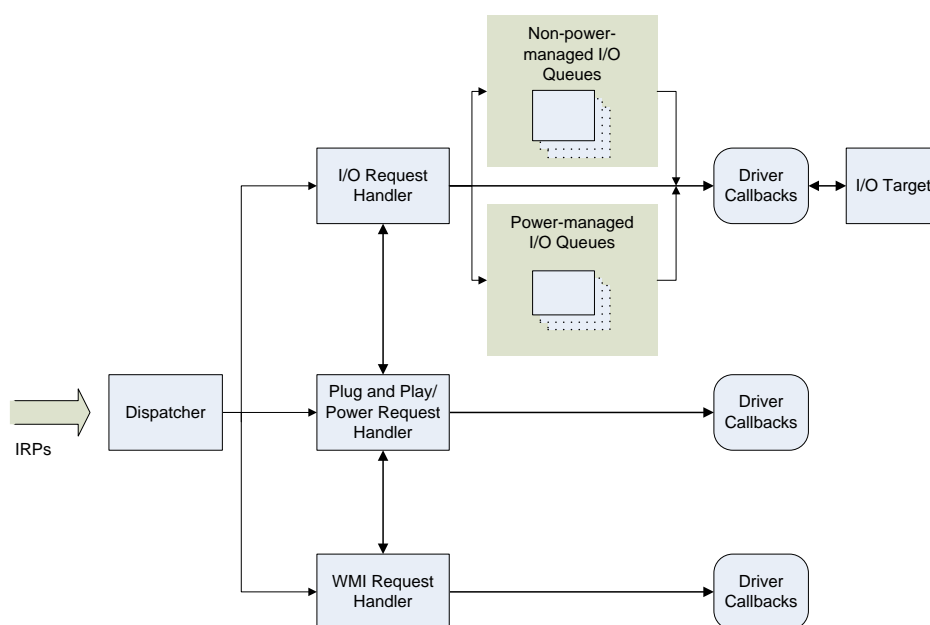


Figure 2. KMDF I/O Flow

When an IRP arrives, KMDF directs it to one of the following components for processing:

- I/O request handler, which handles requests that involve device I/O.
- Plug and Play/power request handler, which handles Plug and Play and power requests (IRP_MJ_PNP and IRP_MJ_POWER requests) and notifies other components of changes in device status.
- WMI handler, which handles WMI and event-tracing requests (IRP_MJ_SYSTEM_CONTROL requests).

Each component takes one or more of the following actions for each request:

- Raising one or more events to the driver.
- Forwarding the request to another internal handler or I/O target for further processing.
- Completing the request based on its own action.

- Completing the request as a result of a driver call.

If the request has not been processed when it reaches the end of frameworks processing, KMDF takes an action that is appropriate for the type of driver. For function and bus drivers, KMDF completes the request with the status `STATUS_INVALID_DEVICE_REQUEST`. For filter drivers, KMDF automatically forwards the request to the default I/O target (the next lower driver).

The next three sections describe how each of the three components processes I/O requests.

I/O Request Handler

The I/O request handler dispatches I/O requests to the driver, manages I/O cancellation and completion, and works with the Plug and Play/power handler to ensure that the device state is compatible with performing device I/O.

Depending on the type of I/O request, the I/O request handler either queues the request or invokes an event callback that the driver registered for the request.

Create, Cleanup, and Close Requests

To handle create events, a driver can either configure a queue to receive the events or can supply an event callback that is invoked immediately. The driver's options are the following:

- To be called immediately, the driver supplies an *EvtDeviceFileCreate* callback and registers it from the *EvtDriverDeviceAdd* callback by calling **WdfDeviceInitSetFileObjectConfig**.
- To configure a queue to receive the requests, the driver calls **WdfDeviceConfigureRequestDispatching** and specifies **WdfRequestTypeCreate**. If the queue is not manual, the driver must register an *EvtIoDefault* callback, which is called when a create request arrives.

Queuing takes precedence over the *EvtDeviceFileCreate* callback; that is, if the driver both registers for *EvtDeviceFileCreate* events and configures a queue to receive such requests, KMDF queues the requests and does not invoke the callback. KMDF does not queue create requests to a default queue; the driver must explicitly configure a queue to receive them.

In a bus or function driver, if a create request arrives for which the driver has neither registered an *EvtDeviceFileCreate* callback function nor configured a queue to receive create requests, KMDF opens a file object to represent the device and completes the request with `STATUS_SUCCESS`. Therefore, any bus or function driver that does not accept create or open requests from user-mode applications—and thus does not register a device interface—must register an *EvtDeviceFileCreate* callback that explicitly fails such requests. Supplying a callback to fail create requests ensures that a rogue user-mode application cannot gain access to the device.

If a filter driver does not handle create requests, KMDF by default forwards all create, cleanup, and close requests to the default I/O target (the next lower driver). Filter drivers that handle create requests should perform whatever filtering tasks are required and then forward such requests to the default I/O target. If the filter driver completes a create request for a file object, it should set **AutoForwardCleanupClose** to **WdfFalse** in the file object configuration so that

KMDF completes cleanup and close requests for the file object instead of forwarding them.

To handle file cleanup and close requests, a driver registers the *EvtFileCleanup* and *EvtFileClose* event callbacks. If a bus or function driver does not register such a callback, KMDF closes the file object and completes the request with `STATUS_SUCCESS`. In a filter driver that does not register cleanup and close callbacks, KMDF forwards these requests to the default I/O target unless the driver has explicitly set **AutoForwardCleanupClose** to **WdfFalse** in the file object configuration.

Read, Write, Device I/O Control, and Internal Device I/O Control Requests

For read, write, device I/O control, and internal device I/O control requests, the driver creates one or more queues and configures each queue to receive one or more types of I/O requests. When such a request arrives, the I/O request handler:

- Determines whether the driver has configured a queue for this type of request. If not, the handler fails a read, write, device I/O control, or internal device I/O control request if this is a function or bus driver. If this is a filter driver, the handler passes such a request to the default I/O target.
- Determines whether the queue is accepting requests and the device is powered on. If both are true, the handler creates a `WDFREQUEST` object to represent the request and adds it to the queue. If the queue is not accepting requests, the handler fails the request.
- If the device is not in the D0 state, notifies the Plug and Play/power handler to power up the device.
- Queues the request.

Figure 3 shows the flow of a read, write, device I/O control, or internal device I/O control request through the I/O request handler to the driver.

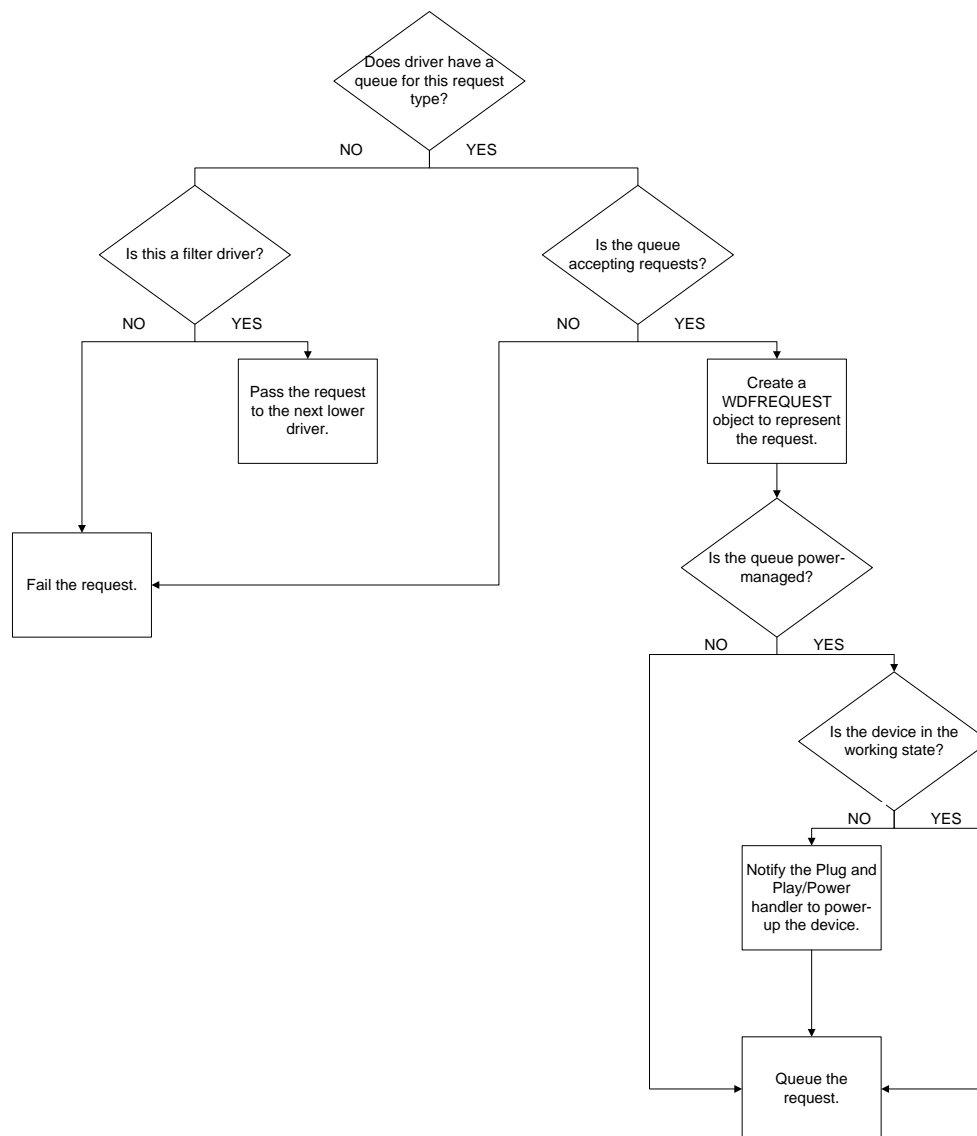


Figure 3. Flow of I/O Request through I/O Request Handler

I/O Queues

A WDFQUEUE object represents a queue that presents requests from KMDF to the driver. A WDFQUEUE is more than just a list of pending requests, however; it tracks requests that are active in the driver, supports request cancellation, manages the concurrency of requests, and can optionally synchronize calls to the driver's I/O event callback functions.

A driver typically creates one or more queues, each of which can accept one or more types of requests. The driver configures the queues when it creates them. For each queue, the driver can specify:

- The types of requests that are placed in the queue.
- The event callback functions that are registered to handle I/O requests from the queue.

- The power management options for the queue.
- The dispatch method for the queue, which determines the number of requests that are serviced at a given time.
- Whether the queue accepts requests that have a zero-length buffer.

A driver can have any number of queues, and they can all be configured differently. For example, a driver might have a parallel queue for read requests and a sequential queue for write requests.

While a request is in a queue and has not yet been presented to the driver, the queue is considered the "owner" of the request. After the request has been dispatched to the driver, it is "owned" by the driver and is considered an *in-flight* request. Internally, each WDFQUEUE object keeps track of which requests it owns and which requests are pending. A driver can forward a request from one queue to another by calling a method on the request object.

Queues and Power Management

KMDF provides rich control of queues. The framework can manage the queues for the driver, or the driver can manage queues on its own. Power management is configurable on a per-queue basis. A driver can use both power-managed and non-power-managed queues and can sort requests based on the requirements for its power model.

Power-Managed Queues

By default, queues for FDOs and PDOs are power managed, which means that the state of the queue can trigger power-management activities. Such queues have several advantages:

- If an I/O request arrives while the system is in the working state (S0) but the device is not, KMDF notifies the Plug and Play/power handler so that it can restore device power.
- When a queue becomes empty, KMDF notifies the Plug and Play/power handler so that it can track device usage through its idle timer.
- If the device power state begins to change while the driver "owns" an I/O request, KMDF can notify the driver through the *EvtIoStop* callback. The driver must complete, cancel, or acknowledge all the I/O requests that it owns before the device can leave the working state.

For power-managed queues, KMDF pauses the delivery of requests when the device leaves the working state (D0) and resumes delivery when the device returns to the working state. Although delivery stops while the queue is paused, queuing does not. If KMDF receives a request while the queue is paused, KMDF adds the request to the queue for delivery after the queue resumes. If an I/O request arrives while the device is idle and the system is in the working state, KMDF returns the device to the working state so that it can handle the request. If an I/O request arrives while the system is transitioning to a sleep state, however, KMDF does not return the device to the working state until the system returns to the working state.

For requests to be delivered, both the driver and the device power state must allow processing. The driver can pause delivery manually by calling **WdfIoQueueStop** and resume delivery by calling **WdfIoQueueStart**.

Non-Power-Managed Queues

If a queue is not power managed, the state of the queue has no effect on power management, and conversely. KMDF delivers requests to the driver any time the system is in the working state, regardless of the power state of the device. KMDF does not start an idle timer when the queue becomes empty, and it does not power up a sleeping device when I/O arrives for the queue.

Drivers should use non-power-managed queues to hold requests that the driver can handle even while its device is not in the working state.

Dispatch Type

A queue's dispatch type determines how and when I/O requests are delivered to the driver and, as a result, whether multiple I/O requests from a queue are active in the driver at one time. Drivers can control the concurrency of in-flight requests by configuring the dispatching method for their queues. KMDF supports three dispatch types:

- **Sequential.** A queue that is configured for sequential dispatching delivers I/O requests to the driver one at a time. The queue does not deliver another request to the driver until the previous request has been completed. (Sequential dispatching is similar to the start-I/O technique in WDM.)
- **Parallel.** A queue that is configured for parallel dispatching delivers I/O requests to the driver as soon as possible, whether or not another request is already active in the driver.
- **Manual.** A queue that is configured for manual dispatching does not deliver I/O requests to the driver. Instead, the driver retrieves requests at its own pace by calling a method on the queue.

Important

The dispatch type controls only the number of requests that are active within a driver at one time. It has no effect on whether the queue's I/O event callbacks are invoked sequentially or concurrently; instead, the concurrency of callbacks is controlled by the synchronization scope of the device object. Even if the synchronization scope for a parallel queue does not allow concurrent callbacks, the queue nevertheless might have many in-flight requests. For more information about synchronization scope for queues, see "Synchronization Issues" later in this paper.

All I/O requests that a driver receives from a queue are inherently asynchronous. The driver can complete the request within the event callback or sometime later, after returning from the callback.

I/O Request Objects

The WDFREQUEST object is the KMDF representation of an IRP. When an I/O request arrives, the I/O handler creates a WDFREQUEST object, queues the object, and eventually passes the object to the driver in its I/O callback function.

The properties of the WDFREQUEST object represent the fields of an IRP. The object also contains additional information. Like all other KMDF objects, the WDFREQUEST object has a reference count and can have its own object context area. When the driver completes the I/O request that the object represents, KMDF automatically frees the object and any child resources such as associated memory buffers or memory descriptor lists (MDLs). After the driver has called **WdfRequestComplete**, the driver must not attempt to access the handle to the WDFREQUEST object or any of its child resources.

A driver can create its own WDFREQUEST objects to request I/O from another device or to split an I/O request into multiple, smaller requests before completing it.

Retrieving Buffers from I/O Requests

The WDFMEMORY object encapsulates the I/O buffers that are supplied for an I/O request. To enable device drivers to handle complicated requests with widely scattered buffers, any number of WDFMEMORY objects may be associated with a WDFREQUEST.

The WDFMEMORY object represents a buffer that the framework manages. The object can be used to copy memory to and from the driver and the buffer represented by the WDFMEMORY handle. In addition, the driver can use the underlying buffer pointer and its length for complex access, such as casting to a known data structure.

Like other KMDF objects, WDFMEMORY objects have reference counts and persist until all references to them have been removed. The buffer that underlies the WDFMEMORY object, however, might not be "owned" by the object itself. For example, if the issuer of the I/O request allocated the buffer or if the driver called **WdfMemoryCreatePreallocated** to assign an existing buffer to the object, the WDFMEMORY object does not "own" the buffer. In this case, the buffer pointer becomes invalid when the associated I/O request has been completed, even if the WDFMEMORY object still exists.

Each WDFMEMORY object contains the length of the buffer that it represents. KMDF methods that copy data to and from the buffer validate the length of every transfer to prevent buffer overruns and underruns, which can result in corrupt data or security breaches.

Depending on the type of I/O that the device and driver support, the underlying buffer might be any of the following:

- For buffered I/O, a system-allocated buffer from the nonpaged pool.
- For direct I/O, a system-allocated MDL that points to the physical pages for DMA.
- For neither buffered nor direct I/O, an unmapped and unverified user-mode memory address.

The WDFMEMORY object supports methods that return each type of buffer from the object and methods to read and write the buffers. For device I/O control requests (IOCTLs), KMDF provides methods to probe and lock user-mode buffers. The driver must be running in the context of the process that sent the I/O request to probe and lock a user-mode buffer, so KMDF also defines a callback that drivers can register to be called in the context of the sending component.

Each WDFMEMORY object also controls access to the buffer and allows the driver to write only to buffers that support I/O from the device to the buffer. A buffer that is used to receive data from the device (as in a read request) is writable. The WDFMEMORY object does not allow write access to a buffer that only supplies data (as in a write request).

Sending I/O Requests

Drivers send I/O requests by creating or reusing an I/O request object, creating an I/O target, and sending the request to the target. Drivers can send requests either synchronously or asynchronously. A driver can specify a time-out value for either type of request.

I/O Targets

An I/O target represents a device object to which an I/O request is directed. If a driver cannot complete an I/O request by itself, it typically forwards the request to an I/O target. An I/O target can be a KMDF driver, a WDM driver, or any other kernel-mode driver.

Before a driver forwards an existing I/O request or sends a new request, it must create a WDFIOTARGET object to represent either a local or remote target for the I/O request. The local I/O target is the next lower driver in the device stack and is the default target for a filter or FDO device object. A remote I/O target is any other driver that might be the target of an I/O request. A driver might use a remote I/O target if it requires data from another device to complete an I/O request. A function driver might also use a remote I/O target to send a device I/O control request to its bus driver. In this case, the I/O request originates with the function driver itself, rather than originating with some other process.

The WDFIOTARGET object formats I/O requests to send to other drivers, handles changes in device state, and defines callbacks through which a driver can request notification about target device removal. A driver can call methods on the WDFIOTARGET to:

- Open a device object or device stack by name.
- Format read, write, and device I/O control requests to send to the target. Some types of targets, such as WDFUSBDEVICE and WDFUSBPIPE, can format bus-specific requests in addition to the standard request types.
- Send read, write, and device I/O control requests synchronously or asynchronously.
- Determine the Plug and Play state of the target.

Internally, KMDF calls **IoCallDriver** to send the request. It takes out a reference on the WDFREQUEST object to prevent the freeing of associated resources while the request is pending for the target device object.

The WDFIOTARGET object tracks queued and sent requests and can cancel them when the state of the target device or of the issuing driver changes. From the driver's perspective, the I/O target object behaves like a cancel-safe queue that retains forwarded requests until KMDF can deliver them. KMDF does not free the WDFIOTARGET object until all the I/O requests that have been sent to it are complete.

By default, KMDF sends a request only when the target is in the proper state to receive it. However, a driver can also request that KMDF ignore the state of the target and send the request anyway. If the target device has been stopped (but not removed), KMDF queues the request to send later after the target device resumes. If the issuing driver specifies a time-out value, the timer starts when the request is added to the queue.

If the device that is associated with a remote I/O target is removed, KMDF stops and closes the I/O target object, but does not notify the driver unless the driver has registered an *EvtIoTargetXxx* callback. If the driver must perform any special processing of I/O requests that it sent to the I/O target, it should register one or more such callbacks. When the removal of the target device is queried, canceled, or completed, KMDF calls the corresponding functions and then processes the target state changes on its own.

For local I/O targets, no such callbacks are defined. Because the driver and the target device are in the same device stack, the driver is notified of device removal requests through its Plug and Play and power management callbacks.

Creating Buffers for I/O Requests

Drivers that issue I/O requests must supply buffers for the results of those requests. The buffers in a synchronous request can be allocated from any type of memory, such as the nonpaged pool or an MDL, as well as a WDFMEMORY object. Asynchronous requests must use WDFMEMORY objects so that KMDF can ensure that the buffers persist until the I/O request has completed back to the issuing driver.

If the driver uses a WDFMEMORY object, the I/O target object takes out a reference on the WDFMEMORY object when it formats the object to send to the I/O target. The target object retains this reference until one of the following occurs:

- The request has been completed.
- The driver reformats the WDFREQUEST object.
- The driver calls **WdfRequestReuse** to send a request to another target.

A driver can retrieve a WDFMEMORY object from an incoming WDFREQUEST and reuse it later in a new request to a different target. However, if the driver has not yet completed the original request, the original I/O target still has a reference on the WDFMEMORY object. To avoid a bug check, the driver must call **WdfRequestReuse** in its I/O completion routine before it completes the original request.

Canceled and Suspended Requests

Windows I/O is inherently asynchronous. The system can request that a driver stop processing an I/O request at any time for many reasons, of which these are the most common:

- The thread or process that issued the request cancels it or exits.
- A system Plug and Play or power event such as hibernation occurs.
- The device is being, or has been, removed.

The actions that a driver takes to stop processing an I/O request depend on the reason for suspension or cancellation. In general, the driver can either cancel the request or complete it with an error. In some situations, the system might request that a driver suspend (temporarily pause) processing; the system notifies the driver later when to resume processing.

To provide a good user experience, drivers should provide callbacks to handle cancellation and suspension of any I/O request that might take a long time to complete or that might not complete, such as a request for asynchronous input.

Request Cancellation

How KMDF proceeds to cancel an I/O request depends on whether the request has already been delivered to the target driver.

- If the request has never been delivered—either because KMDF has not yet queued it or because it is still in a queue—KMDF cancels or suspends it automatically. If the original IRP has been canceled, KMDF completes the request with a cancellation status.
- If the request has been delivered and then requeued, KMDF notifies the driver of cancellation only if the driver has registered an *EvtIoCanceledOnQueue* callback for the queue.

After a request has been delivered, it cannot be canceled unless the driver that owns it explicitly marks it cancelable by calling the **WdfRequestMarkCancelable** method on the request and registering a cancellation callback (*EvtRequestCancel*) for the request.

A driver should mark a request cancelable and register an I/O cancellation callback if either of the following is true:

- The request involves a long-term operation.
- The request might never succeed; for example, the request is waiting for synchronous input.

Drivers should follow the guidelines that are described in "I/O Completion/Cancellation Guidelines," which is listed in the Resources section.

In the *EvtRequestCancel* callback, the driver must perform any tasks that are required to cancel the request, such as stopping any device I/O operations that are in progress and canceling any related requests that it has already forwarded to an I/O target. Eventually, the driver must complete the request with the status `STATUS_CANCELLED`.

Requests that are marked cancelable cannot be forwarded to another queue. Before requeuing a request, the driver must first make it noncancelable by calling **WdfRequestUnmarkCancelable**. After the request has been added to the new queue, KMDF once again considers it cancelable until that queue dispatches it to the driver.

If the driver does not mark a request cancelable, it can call **WdfRequestIsCanceled** to determine whether the I/O manager or original requester has attempted to cancel the request. A driver that processes data on a periodic basis might use this approach. For example, a driver involved in image processing might complete a transfer request in small chunks and poll for cancellation after processing each chunk. In this case, the driver supports cancellation of the I/O request, but only after each discrete chunk of processing is complete. If the driver determines that the request has been canceled, it performs any required cleanup and completes the request with the status `STATUS_CANCELLED`.

Request Suspension

When the system transitions to a sleep state—typically because the user has requested hibernation or closed the lid on a laptop—a driver can complete, requeue, or continue to hold any in-flight requests. KMDF notifies the driver of the impending power change by calling the *EvtIoStop* callback for each in-flight request. Each call includes flags that indicate the reason for stopping the queue and whether the I/O request is currently cancelable.

Depending on the value of the flags, the driver can complete the request, requeue the request, acknowledge the event but continue to hold the request, or ignore the event if the current request will complete in a timely manner. If the queue is stopping because the device is being removed (either by an orderly removal or a surprise removal), the driver must complete the request immediately.

Drivers should handle *EvtIoStop* events for any request that might take a long time to complete or that might not complete, such as a request for asynchronous input. Handling *EvtIoStop* provides a good user experience for laptops and other power-managed systems.

Completing I/O Requests

To complete an I/O request, a driver calls **WdfRequestComplete**. In response, KMDF completes the underlying IRP and then deletes the WDFREQUEST object and any child objects. If the driver has set an *EvtCleanupCallback* for the WDFREQUEST object, KMDF invokes the callback before completing the underlying IRP, so that the IRP itself is still valid when the callback runs.

After **WdfRequestComplete** returns, the WDFREQUEST object's handle is invalid and its resources have been released. The driver must not attempt to access the handle or any of its resources, such as parameters and memory buffers that were passed in the request.

If the request was dispatched from a sequential queue, the driver's call to complete the IRP might cause KMDF to deliver the next request in the queue. (If the queue is configured for parallel dispatching, KMDF can deliver another request at any time.) If the driver holds any locks while it calls **WdfRequestComplete**, it must ensure that its event callbacks for the queue do not use the same locks because a deadlock might occur. In practice, this is difficult to ensure, so the best practice is not to call **WdfRequestComplete** while holding a lock.

Self-Managed I/O

Although the I/O support that is built into KMDF is recommended for most drivers, some drivers have I/O paths that do not pass through queues or are not subject to power management. KMDF provides self-managed I/O features for this purpose. For example, the PCIDRV sample uses self-managed I/O callbacks to start and stop a watchdog timer DPC.

The self-managed I/O callbacks correspond directly to WDM Plug and Play and power management state changes. These routines are called with a handle to the device object and no other parameters. If a driver registers these callbacks, KMDF calls them at the designated times so that the driver can perform whatever actions it requires.

Accessing IRPs and WDM Structures

KMDF includes a mechanism nicknamed "the great escape" through which a driver can access the underlying WDM structures and the I/O request packet as it was delivered from the operating system. Although this mechanism exposes the driver to all the complexity of the WDM model, it can often be useful in converting a WDM driver to KMDF. In addition, some drivers require WDM features that are not available in KMDF, such as processing for some types of IRPs. Such drivers can use KMDF for most features but can rely on the "great escape" to gain access to the WDM features that they require.

To use the "great escape," a driver calls **WdfDeviceInitAssignWdmIrpPreprocessCallback** to register an *EvtDeviceWdmIrpPreprocess* event callback function for an IRP major function code. When KMDF receives an IRP with that function code, it invokes the callback. The driver must then handle the request just as a WDM driver would, by using I/O manager functions such as **IoCallDriver** to forward the request and **IoCompleteRequest** to complete it. The Serial driver sample shows how to use this feature.

In addition to the "great escape," KMDF provides methods with which a driver can access the WDM objects that the KMDF objects represent. For example, a driver can access the IRP that underlies a WDFREQUEST object, the WDM device object that underlies a WDFDEVICE object, and so forth.

Plug and Play and Power Management Request Handler

KMDF implements integrated Plug and Play and power management support as an internal state machine. An event is associated with the transition to each state, and a driver can supply callback routines that are invoked at each such state change.

If you are familiar with WDM drivers, you probably remember that any time the system power state changes, the WDM driver must determine the correct power state for its device and then issue power management requests to put the device in that state at the appropriate time. The KMDF state machine automatically handles the translation of system power events to device power events. For example, KMDF notifies the driver to:

- Transition the device to low power when the system hibernates or goes to sleep.
- Enable the device's wake signal so that it can be triggered while the system is running, if the device is idle.
- Enable the device's wake signal so that it can be triggered while the system is in a sleep state.

KMDF automatically provides for the correct behavior in device parent/child relationships. If both a parent and a child device are powered down and the child must power up, KMDF automatically returns the parent to full power and then powers up the child.

To manage idle devices, the KMDF state machine notifies the driver to remove the device from the working state and put it in the designated low-power state when the device is idle and to return the device to the working state when there are requests to process.

To accomplish these power transitions, a driver includes a set of callback routines. These routines are called in a defined order, and each conforms to a "contract," so that both the device and the system are guaranteed to be in a particular state when the driver is called to perform an action. This support makes it much easier for drivers to power down idle devices. The driver simply sets an appropriate time-out value and low-power state for its device and notifies KMDF of these values; KMDF calls the driver to power down the device at the correct times.

In addition, requests received by the framework and not yet delivered to the device driver can affect the power state of the device. If the driver has configured a queue for power management, the framework can automatically restore device power before it delivers the request to the driver. It can also automatically stop and start the queue in response to Plug and Play and power events.

Finally, the driver that manages power policy for the device can specify whether a user can control both the behavior of the device while it is idle and the ability of the device to wake up the system. All the driver must do is specify the appropriate enumerator value when it initializes certain power policy settings. KMDF enables the necessary property sheet through WMI, and Device Manager displays it.

Device Enumeration and Startup

To prepare the device for operation, KMDF calls the driver's callback routines in a fixed sequence. The sequence varies somewhat depending on the driver's role in the device stack.

Startup Sequence for a Function or Filter Device Object

Figure 4 shows the callbacks for an FDO or filter DO that is involved in bringing a device to the fully operational state, starting from the Device Inserted state at the bottom of the figure.

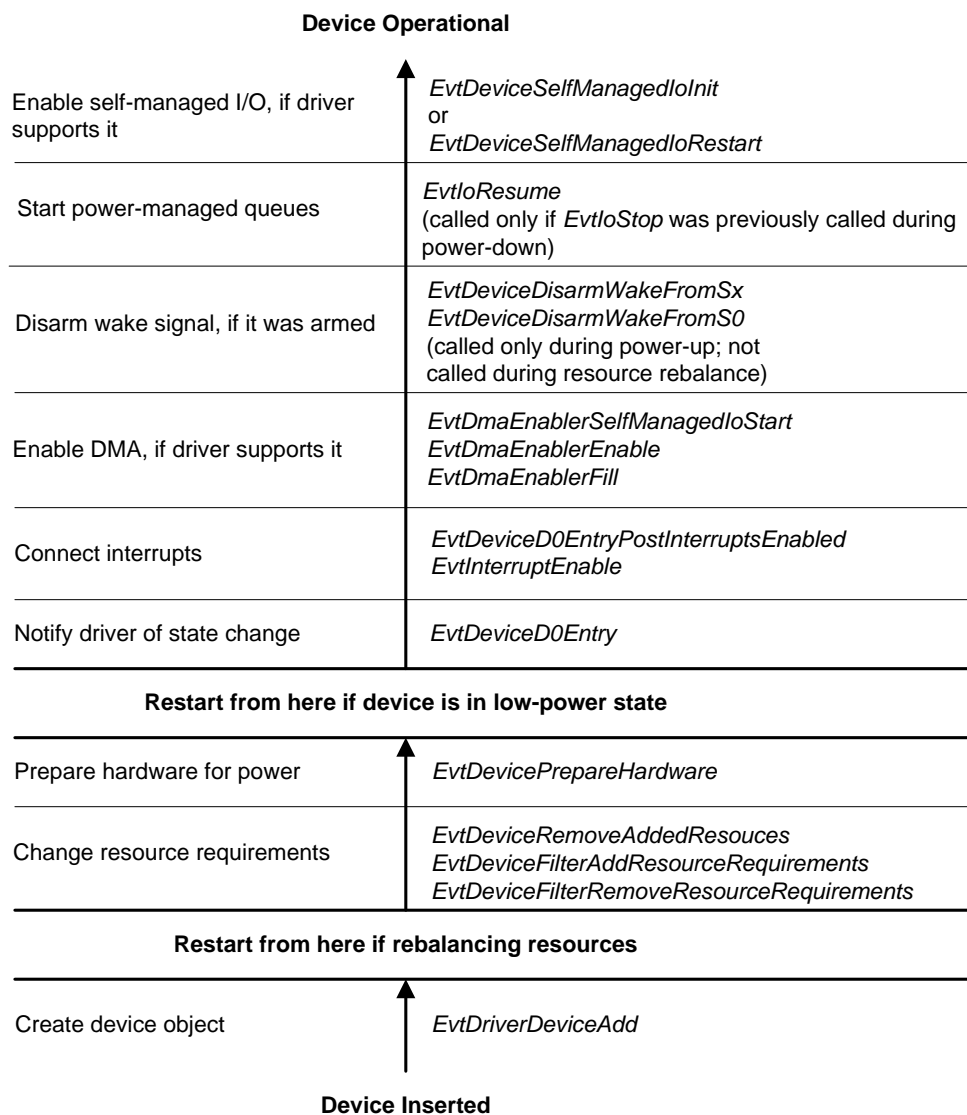


Figure 4. Device Enumeration and Startup Sequence for FDO or Filter DO

The broad horizontal lines mark the steps that are involved in starting a device. The column on the left side of the figure describes the step, and the column on the right lists the event callbacks that accomplish it.

At the bottom of the figure, the device is not present on the system. When the user inserts it, KMDF begins by calling the driver's *EvtDriverDeviceAdd* callback so that the driver can create a device object to represent the device. KMDF continues calling the driver's callback routines by progressing up through the sequence until the device is operational. Remember that KMDF invokes the event callbacks in bottom-up order as shown in the figure, so *EvtDeviceFilterRemoveResourceRequirements* is called before *EvtDeviceFilterAddResourceRequirements* and so forth.

If the device was stopped to rebalance resources or was physically present but not in the working state, not all of the steps are required, as the figure shows.

Startup Sequence for a Physical Device Object

Figure 5 shows the callbacks for a bus driver (PDO) that are involved in bringing a device to the fully operational state, starting from the Device Inserted state at the bottom of the figure.

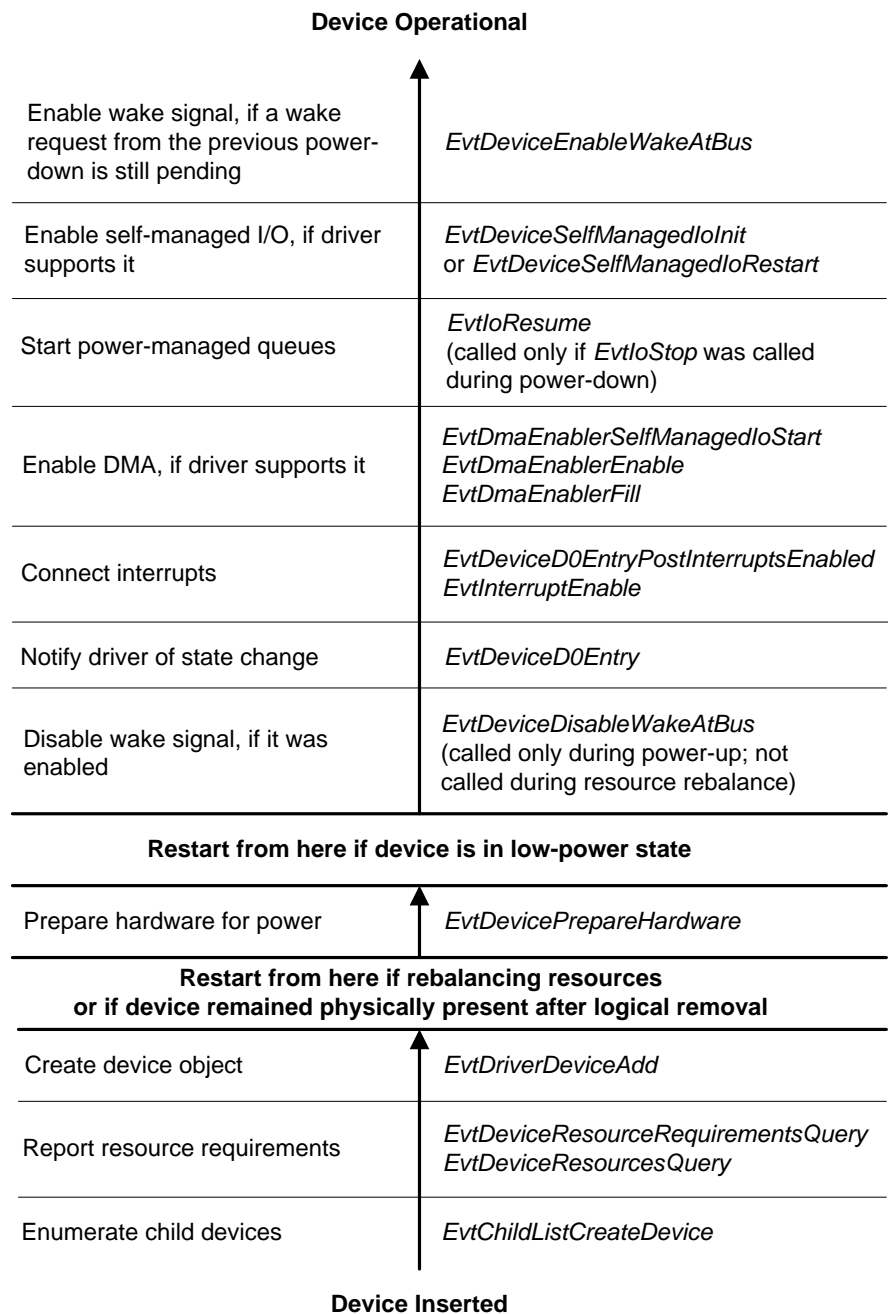


Figure 5. Device Addition/Startup Sequence for PDO

KMDF does not physically delete a PDO until the corresponding device is physically removed from the system. For example, if a user disables the device in Device Manager but does not physically remove it, KMDF retains its device object. Thus, the three steps at the bottom of the figure occur only during Plug and Play enumeration—that is, during initial boot or when the user plugs in a new device.

If the device was previously disabled but not physically removed, KMDF starts by calling the *EvtDevicePrepareHardware* callback.

Device Power Down and Removal

KMDF can remove a device from the operational state for several reasons:

- To put the device in a low-power state because it is idle or the system is entering a sleep state.
- To rebalance resources.
- To remove the device after the user has requested an orderly removal.
- To disable the device in response to the user's request in Device Manager.

As in enumeration and power-up, the sequence of callbacks depends on the driver's role in device management.

Power-Down and Removal Sequence for a Function or Filter Device Object

Figure 6 shows the sequence of callbacks that are involved in power down and removal for an FDO or filter DO. The sequence starts at the top of the figure with an operational device that is in the working power state (D0).

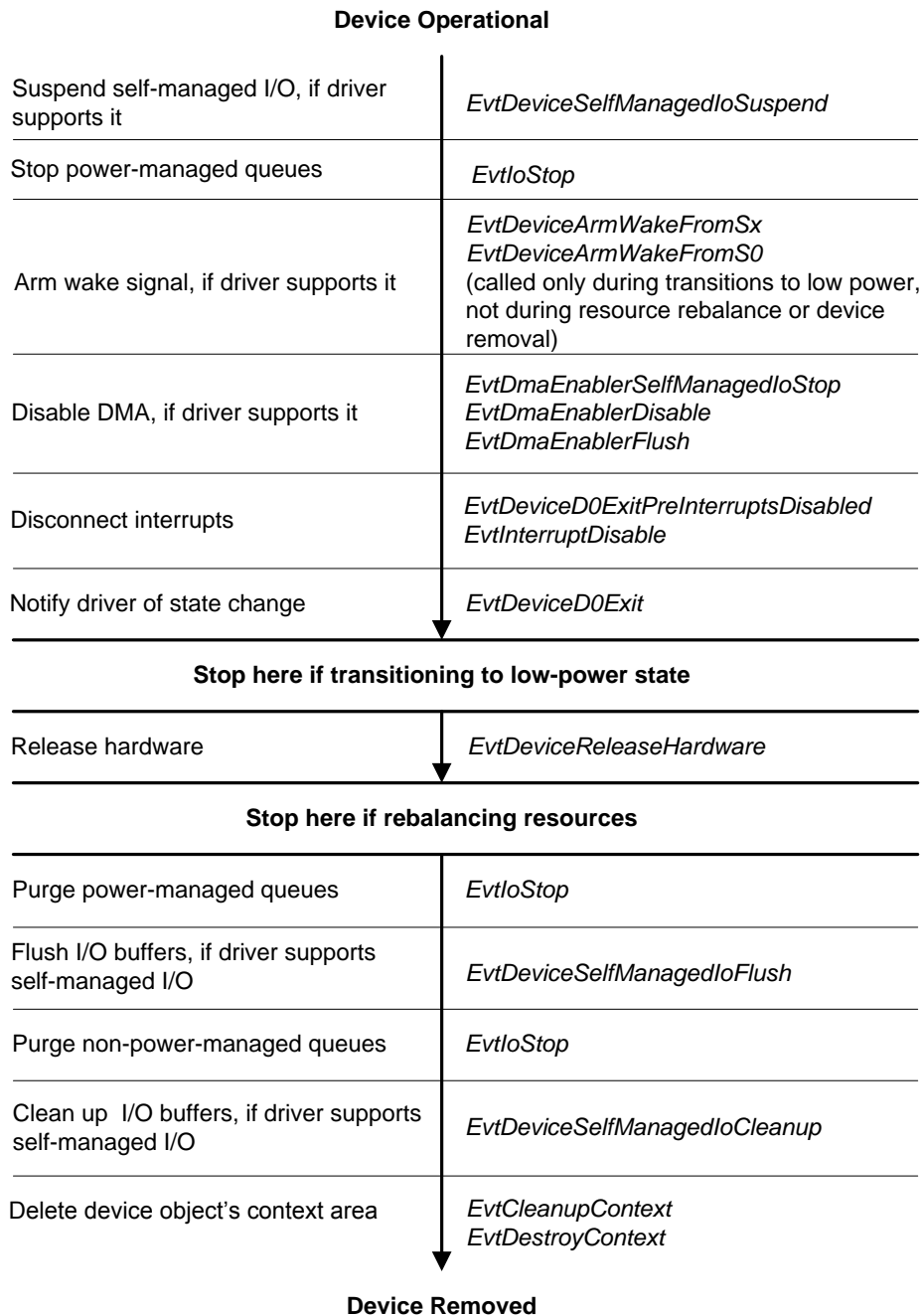


Figure 6. Device Power-Down and Orderly Removal Sequence for FDO and Filter DO

As the figure shows, the KMDF power-down and removal sequence involves calling the corresponding "undo" callbacks in the reverse order from which KMDF called the functions that are involved in making the device operational.

Power-Down and Removal Sequence for a Physical Device Object

Figure 7 shows the callbacks involved in power down and removal for a PDO.

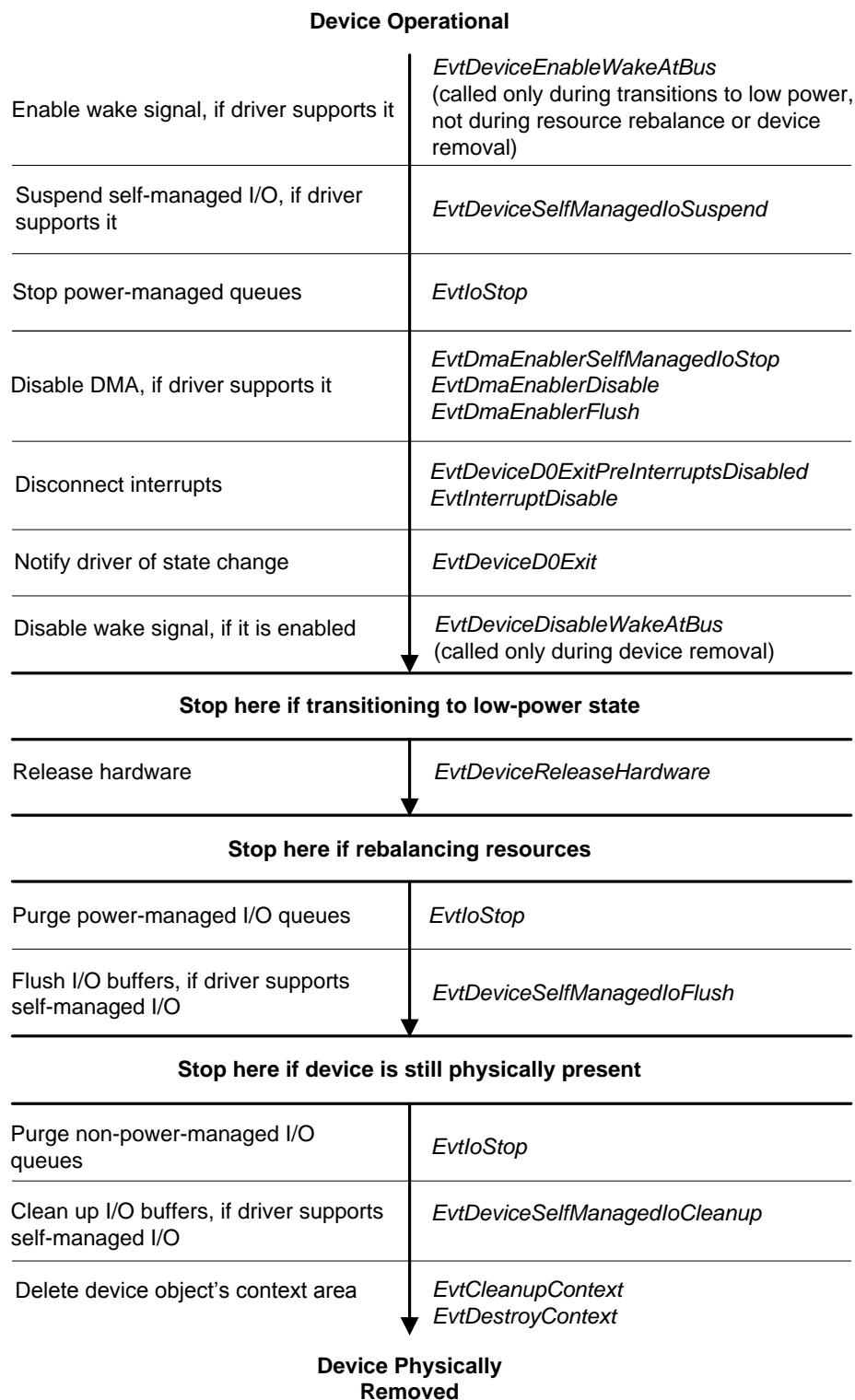


Figure 7. Device Power-Down and Orderly Removal Sequence for PDO

KMDF does not physically delete the PDO until the device is physically removed from the system. For example, if a user disables the device in Device Manager or uses the Safely Remove Hardware utility to stop the device but does not physically remove it, KMDF retains the PDO. If the device is later re-enabled, KMDF uses the same PDO and begins the startup sequence by calling the *EvtDevicePrepareHardware* callback, as previously shown in Figure 5.

Surprise Removal Sequence

If the user removes the device without warning, by simply unplugging it without using Device Manager or the Safely Remove Hardware utility, the device is considered "surprise-removed." When this occurs, KMDF follows a slightly different removal sequence. It also follows the surprise-removal sequence if another driver calls **IoInvalidateDeviceState** on the device, even if the device is still physically present.

In the surprise-removal sequence, KMDF calls the *EvtDeviceSurpriseRemoval* callback before calling any of the other callbacks in the removal sequence. When the sequence is complete, KMDF destroys the device object.

Drivers for all removable devices must ensure that the callbacks in both the shutdown and startup paths can handle failure, particularly failures caused by the removal of the hardware. Any attempts to access the hardware should not wait indefinitely, but should be subject to time-outs or a watchdog timer.

Figure 8 shows the surprise-removal sequence.

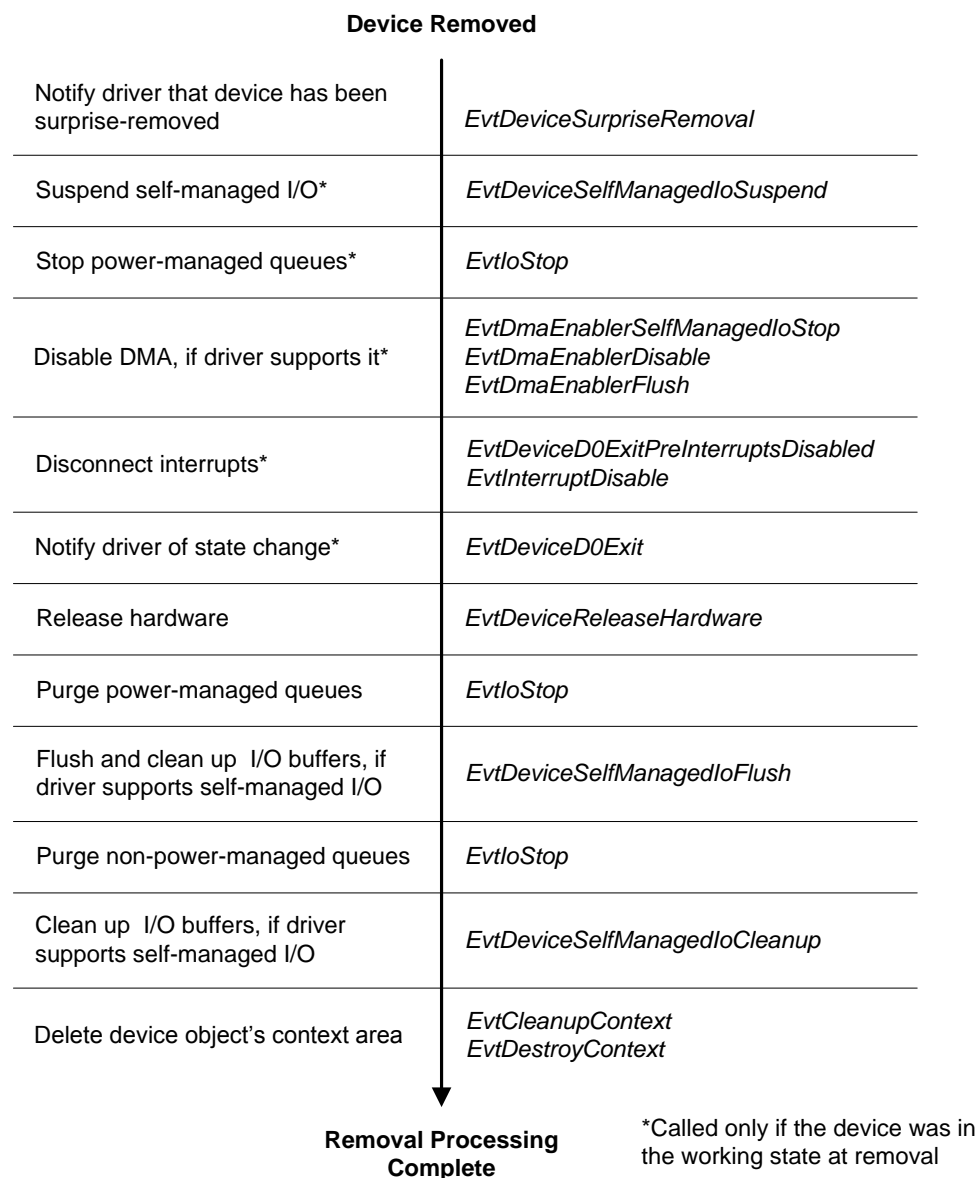


Figure 8. Surprise Removal Sequence

If the device was not in the working state when it was removed, KMDF calls the *EvtDeviceReleaseHardware* event callback immediately after *EvtDeviceSurpriseRemoval*. It omits the intervening steps, which were already performed when the device exited from the working state.

WMI Request Handler

WMI provides a way for drivers to export information to other components. Drivers typically use WMI to:

- Enable user-mode applications to query and set device-related information, such as time-out values.
- Enable an administrator with the necessary privileges to control a device by running an application on a remote system.

A driver that supports WMI registers as a provider of information and registers one or more instances of that information. Each WMI provider is associated with a particular globally unique identifier (GUID). Another component can register with the same GUID to consume the data from the instances. User-mode components request WMI instance data by calling COM functions, which the system translates into IRP_MJ_SYSTEM_CONTROL requests and sends to the target providers.

KMDF supports WMI requests through its WMI request handler, which provides the following features for drivers:

- A default WMI implementation. Drivers that do not provide WMI data are not required to register as WMI data providers; KMDF handles all IRP_MJ_SYSTEM_CONTROL requests.
- Callbacks on individual instances, rather than just at the device object level, so that different instances can behave differently.
- Validation of buffer sizes to ensure that buffers that are used in WMI queries meet the size requirements of the associated provider and instance.

The default WMI implementation includes support for the check boxes on the **Power Management** tab of Device Manager. These check boxes enable a user to control whether the device can wake the system and whether the system can power down the device when it is idle. WDM drivers must include code to support the WMI controls that map to these check boxes, but KMDF drivers do not require such code. If the driver enables this feature in its power policy options, KMDF handles these requests automatically.

The driver enables buffer size validation when it configures a WMI provider object (WDFWMI_PROVIDER). In the WDF_WMI_PROVIDER_CONFIG structure, the driver can specify the minimum size of the buffer that is required for the provider's *EvtWmiInstanceQueryInstance* and *EvtWmiInstanceSetInstance* callbacks. If the driver specifies such a value, KMDF validates the buffer size when the IRP_MJ_SYSTEM_CONTROL request arrives and calls the callbacks only if the supplied buffer is large enough. If the driver does not configure a buffer size—because the instance size is either dynamic or is not available when the provider is created—the driver should specify zero for this field and the callbacks themselves should validate the buffer sizes.

When KMDF receives an IRP_MJ_SYSTEM_CONTROL request that is targeted at a KMDF driver, it proceeds as follows:

- If the driver has registered as a WMI provider and registered one or more instances, the WMI handler invokes the callbacks for those instances as appropriate.
- If the driver has not registered any WMI instances, the WMI handler responds to the request by providing the requested data (if it can), passing the request to the next lower driver, or failing the request.

Like all KMDF objects, WMI instance objects (WDFWMIINSTANCE) have a context area. A driver can use the context area of a WDFWMIINSTANCE object as a source of read-only data, thus enabling easy data collection with minimal effort. A driver can delete WDFWMIINSTANCE objects any time after their creation.

WMI callbacks are not synchronized with the Plug and Play and power management state of the device. Therefore, when WMI events occur, KMDF calls a driver's WMI callbacks even if the device is not in the working state.

Synchronization Issues

Because Windows is a pre-emptive, multitasking operating system, multiple threads can try to access shared data structures or resources concurrently and multiple driver routines can run concurrently. To ensure data integrity, all drivers must synchronize access to shared data structures. Correctly implementing such synchronization can be difficult in WDM drivers.

For KMDF drivers, ensuring proper synchronization requires attention to several areas:

- The number of concurrently active requests that are dispatched from a particular queue.
- The number of concurrently active callbacks for a particular object.
- The driver utility functions that access object-specific data.
- The IRQL at which an object's callbacks run.

The dispatch method for an I/O queue controls the number of requests from the queue that can be concurrently active in the driver, as described previously in "Dispatch Type." Limiting concurrent requests does not, however, resolve all potential synchronization issues. Concurrently active callbacks on the same object might require access to shared object-specific data, such as the information that is stored in the object context area. Similarly, driver utility functions might share object-specific data with callbacks. Furthermore, a driver must be aware of the IRQL at which its callbacks can be invoked. At DISPATCH_LEVEL and above, drivers must not access pageable data and thread pre-emption does not occur.

KMDF simplifies synchronization for drivers by providing automatic synchronization of many callbacks. Calls to most PDO, FDO, Plug and Play, and power event callback functions are synchronized so that only one such callback function is invoked at a time for each device. These callback functions are called at IRQL PASSIVE_LEVEL. Note, however, that calls to the *EvtDeviceSurpriseRemoval*, *EvtDeviceQueryRemove*, and *EvtDeviceQueryStop* callbacks are not synchronized with the other callbacks and so can occur while the device is changing power state or is not in the working state.

For other types of callbacks—primarily I/O-related callbacks—the driver can specify the *synchronization scope* (degree of concurrency) and the maximum execution level (IRQL).

KMDF provides the following configurable synchronization features:

- Synchronization scope
- Execution level
- Locks

Although implementing synchronization is much less complicated in KMDF drivers than in WDM drivers, you should nevertheless be familiar with the basics of Windows IRQL, synchronization, and locking, as described in the papers "Scheduling, Thread Context, and IRQL" and "Locks, Deadlocks, and Synchronization," which are listed in the Resources at the end of this paper.

Synchronization Scope

KMDF provides configurable concurrency control, called *synchronization scope*, for the callbacks of several types of objects. An object's synchronization scope determines whether KMDF invokes certain event callbacks on the object concurrently.

KMDF defines the following synchronization scopes:

- *Device scope* means that KMDF does not call certain I/O event callbacks concurrently for an individual device object or any file objects or queues that are children of the device object. Specifically, device scope applies to the following event callbacks: *EvtDeviceFileCreate*, *EvtFileCleanup*, *EvtFileClose*, *EvtIoDefault*, *EvtIoRead*, *EvtIoWrite*, *EvtIoDeviceControl*, *EvtIoInternalDeviceControl*, *EvtIoStop*, *EvtIoResume*, *EvtIoQueueState*, *EvtIoCanceledOnQueue*, and *EvtRequestCancel*.

However, callbacks for different device objects that were created by the same driver object can be called concurrently. Internally, KMDF creates a synchronization lock for each device object. To implement device synchronization scope, KMDF acquires this lock before invoking any of the device object's callbacks.

- *Queue scope* means that KMDF does not call certain I/O callbacks concurrently on a per-queue basis. If a kernel-mode driver specifies queue scope for a device object, some callbacks for the device object and its queues can run concurrently. However, the following callbacks for an individual queue object are not called concurrently: *EvtIoDefault*, *EvtIoRead*, *EvtIoWrite*, *EvtIoDeviceControl*, *EvtIoInternalDeviceControl*, *EvtIoStop*, *EvtIoResume*, *EvtIoQueueState*, *EvtIoCanceledOnQueue*, and *EvtRequestCancel*. If the driver specifies queue scope, KMDF creates a synchronization lock for each queue object and acquires this lock before invoking any of the listed callbacks.
- *No scope* means that KMDF does not acquire any locks and can call any event callback concurrently with any other event callback. The driver must create and acquire all its own locks. By default, KMDF uses no scope. A driver must "opt in" to synchronization for its objects by setting device scope explicitly.

Each KMDF object inherits its scope from its parent object (**WdfSynchronizationScopeInheritFromParent**). The parent of each WDFDEVICE object is the WDFDRIVER object, and the default value of the synchronization scope for the WDFDRIVER object is **WdfSynchronizationScopeNone**. Thus, a driver must explicitly set the synchronization scope on its objects to use frameworks synchronization.

A driver can change the scope by setting a value in the WDF_OBJECT_ATTRIBUTES structure when it creates the object. Because scope is inherited, a driver can easily set synchronization for most of its objects by setting the scope for the device object, which is the parent to most KMDF objects. (For the complete hierarchy, see Figure 1.)

For example, to set the concurrency for its I/O callback functions, a driver sets the **SynchronizationScope** in the `WDF_OBJECT_ATTRIBUTES` for the device object that is the parent to the I/O queues. If the driver sets device scope (**WdfSynchronizationScopeDevice**), KMDf calls only one I/O callback function at a time across all the queues. To use queue scope, the driver sets **WdfSynchronizationScopeQueue** for the device object and **WdfSynchronizationScopeInheritFromParent** for the queue object. Queue scope means that only one of the listed callback functions can be active for the queue at any time. A driver cannot set concurrency separately for each queue. Restricting the concurrency of I/O callbacks can help to manage access to shared data in the `WDFQUEUE` context memory.

By default, a file object inherits its scope from its parent device object. Attempting to set queue scope for a file object causes an error. Therefore, drivers that set queue scope for a device object must manually set the synchronization scope for any file objects that are its children. The best practice for file objects is to use no scope and to acquire locks in the event callback functions when they are required to synchronize access.

If a driver sets device scope for a file object, it must also set the passive execution level for the object, as described in "Execution Level" later in this paper. The reason is that the framework uses spin locks (which raise IRQL to `DISPATCH_LEVEL`) to synchronize access to objects with device scope. However, the *EvtDeviceFileCreate*, *EvtFileClose*, and *EvtFileCleanup* callbacks run in the caller's thread context and use pageable data, so they must be called at `PASSIVE_LEVEL`. At `PASSIVE_LEVEL`, the framework uses a `FAST_MUTEX` instead of a spin lock for synchronization.

Interrupt objects are the children of device objects. KMDf acquires the interrupt object's spin lock at device interrupt request level (`DIRQL`) to synchronize calls to the *EvtInterruptEnable*, *EvtInterruptDisable*, and *EvtInterruptIsr* callbacks. A driver can also ensure that calls to its interrupt object's *EvtInterruptDpc* callback are serialized with other callbacks on the parent device object.

DPC, timer, and work item objects can be the children of device objects or of queue objects. To simplify a driver's implementation of callbacks for DPCs, timers, and work items, KMDf enables the driver to synchronize their callbacks with those of either the associated queue object or the device object (which might be the parent or the grandparent of the DPC, timer, or work item).

A driver sets callback synchronization on interrupt, DPC, timer, and work item objects by setting **AutomaticSerialization** in the object's configuration structure during object creation.

Execution Level

KMDf drivers can specify the maximum IRQL at which the callbacks for driver, device, file, and general objects are invoked. Like synchronization scope, execution level is an attribute that the driver can configure when it creates the object. KMDf supports the following execution levels:

- *Default execution level* indicates that the driver has placed no particular constraints on the IRQL at which the callbacks for the object can be invoked. For most objects, this is the default.

- *Passive execution level (**WdfExecutionLevelPassive**)* means that all event callbacks for the object occur at `PASSIVE_LEVEL`. If necessary, KMDF invokes the callback from a system worker thread. Drivers can set this level only for device and file objects. Typically, a driver should set passive execution level only if the callbacks access pageable code or data or call other functions that must be called at `PASSIVE_LEVEL`.

Callbacks for events on file objects (`WDFFILEOBJECT` type) are always called at `PASSIVE_LEVEL` because these functions must be able to access pageable code and data.

- *Dispatch execution level (**WdfExecutionLevelDispatch**)* means that KMDF can invoke the callbacks from any IRQL up to and including `DISPATCH_LEVEL`. This setting does not force all callbacks to occur at `DISPATCH_LEVEL`. However, if a callback requires synchronization, KMDF uses a spin lock, which raises IRQL to `DISPATCH_LEVEL`. Drivers can set dispatch execution level but nevertheless ensure that some tasks are performed at `PASSIVE_LEVEL` by using work items (`WDFWORKITEM` objects). Work item callbacks are always invoked at `PASSIVE_LEVEL` in the context of a system thread.

By default, an object inherits its execution level from its parent object. The default execution level for the `WDFDRIVER` object is **WdfExecutionLevelDispatch**.

Locks

In addition to internal synchronization, synchronization scope, and execution level, KMDF provides the following additional ways for a driver to synchronize operations:

- Acquire the lock that is associated with a device or queue object.
- Create and use additional, KMDF-defined, driver-created lock objects.

Driver code that runs outside an event callback sometimes must synchronize with code that runs inside an event callback. To accomplish this synchronization, KMDF provides methods (**WdfObjectAcquireLock** and **WdfObjectReleaseLock**) through which the driver can acquire and release the internal framework lock that is associated with a particular device or queue object.

Given the handle to a device or queue object, **WdfObjectAcquireLock** acquires the lock that protects that object. After acquiring the lock, the driver can safely access the object context data or properties and can perform other actions that affect the object. If the driver has set **WdfExecutionLevelPassive** for the object (or if the object has inherited this value from its parent), KMDF uses a `PASSIVE_LEVEL` synchronization primitive (a fast mutex) for the lock. If the object does not have this constraint, use of the lock raises IRQL to `DISPATCH_LEVEL` and, while the driver holds the lock, it must not touch pageable code or data or call functions that must run at `PASSIVE_LEVEL`.

KMDF also defines two types of lock objects:

- *Wait locks (`WDFWAITLOCK`)* synchronize access from code that runs at IRQL `PASSIVE_LEVEL` or `APC_LEVEL`. Such locks prevent thread suspension. Internally, KMDF implements wait locks by using kernel dispatcher events, so each wait lock is associated with an optional time-out value (as are the kernel dispatcher events). If the time-out value is zero, the driver can acquire the lock at `DISPATCH_LEVEL`.

- *Spin locks* (WDFSPINLOCK) synchronize access from code that runs at any IRQL up to DISPATCH_LEVEL. Because code that holds a spin lock runs at DISPATCH_LEVEL, it cannot take a page fault and therefore must not access any pageable data. The WDFSPINLOCK object keeps track of its acquisition history and ensures that deadlocks cannot occur. Internally, KMDF uses the system's spin lock mechanisms to implement spin lock objects.

As with all other KMDF objects, each instance of a lock object can have its own context area that holds lock-specific information.

Drivers that do not use the built-in frameworks locking (synchronization scope, execution level, and **AutomaticSerialization**) can implement their own locking schemes by using KMDF wait locks and spin locks. Drivers that use frameworks locking can use KMDF wait locks and spin locks to synchronize access to data that is not associated with a particular device or queue object. In general, drivers can rely on frameworks locking while communicating with their own hardware and calling within their own code. Drivers that communicate with other drivers generally must implement their own locking schemes.

Interaction of Synchronization Mechanisms

Synchronization scope and execution level interact because of the way in which KMDF implements synchronization. By default, KMDF uses spin locks, which raise IRQL to DISPATCH_LEVEL, to synchronize callbacks. Therefore, if the driver specifies device or queue synchronization scope, its device and queue callbacks must be able to run at DISPATCH_LEVEL.

If the driver sets the **WdfExecutionLevelPassive** constraint for a parent device or queue object, KMDF uses a fast mutex instead of a spin lock. In this case, however, KMDF cannot automatically synchronize callbacks for timer and DPC child objects (including the DPC object that is associated with the interrupt object) because DPC and timer callbacks, by definition, always run at DISPATCH_LEVEL. Trying to create any of these objects with **AutomaticSerialization** fails if the **WdfExecutionLevelPassive** constraint is set for the parent object.

Instead, the driver can synchronize the event callbacks for these objects by using a WDFSPINLOCK object. The driver acquires and releases the lock manually by the KMDF locking methods **WdfSpinLockAcquire** and **WdfSpinLockRelease**. Alternatively, the driver can perform whatever processing is required within the DPC or timer callback and then queue a work item that is synchronized with the callbacks at PASSIVE_LEVEL to perform further detailed processing.

Security

KMDF is designed to enhance the creation of secure drivers by providing:

- Safe defaults
- Parameter validation
- Counted Unicode strings
- Safe device naming techniques

Safe Defaults

Unless the driver specifies otherwise, KMDF provides access control lists (ACLs) that require Administrator privileges for access to any exposed driver constructs, such as names, device IDs, WMI management interfaces, and so forth. In addition,

KMDF automatically handles I/O requests for which a driver has not registered by completing them with STATUS_INVALID_DEVICE_REQUEST.

Parameter Validation

One of the most common driver security problems involves improper handling of buffers in IOCTL requests, particularly requests that specify neither buffered nor direct I/O (METHOD_NEITHER). By default, KMDF does not grant drivers direct access to user-mode buffer pointers, which is inherently unsafe. Instead, it provides methods for accessing the user-mode buffer pointer that require probing and locking, and it provides methods to probe and lock the buffer for reading and writing.

All KMDF DDIs that require a buffer take a length parameter that specifies a required minimum buffer size. I/O buffers use the WDFMEMORY object, which provides data access methods that automatically validate length and determine whether the buffer permissions allow write access to the underlying memory.

Counted UNICODE Strings

To help prevent string handling errors, KMDF DDIs use only counted PUNICODE_STRING values. To aid drivers in using and formatting UNICODE_STRING values, the safe string routines in ntstrsafe.h have been updated to take PUNICODE_STRING parameters.

Device Naming Techniques

KMDF device objects do not have fixed names. KMDF sets FILE_AUTOGENERATED_DEVICE_NAME in the device's characteristics for PDOs, according to the WDM requirements.

KMDF also supports the creation and registration of device interfaces on all Plug and Play devices and manages device interfaces for its drivers. Whenever possible, you should use device interfaces instead of the older fixed name/symbolic link techniques.

However, if legacy applications require that a device has a name, KMDF enables you to name a device and to specify its security descriptor definition language (SDDL). The SDDL controls which users can open the device.

By convention, a fixed device name is associated with a fixed symbolic link name (such as \DosDevices\MyDeviceName). KMDF supports the creation and management of a symbolic link and automatically deletes the link when the device is destroyed. KMDF also enables the creation of a symbolic link name for an unnamed Plug and Play device.

Build and Debug Environment

KMDF drivers, like WDM drivers, are built in the WDK build environment. KMDF drivers include the header files Wdf.h (shipped with KMDF) and ntddk.h.

To build a KMDF driver, you must set the /GS flag on the compiler and the KMDF_VERSION environment variable in the Sources file. Setting KMDF_VERSION=1 indicates that the driver should be built with the first version of KMDF.

KMDF is distributed with the following libraries:

- WdfDriverEntry.lib defines the framework's FxDriverEntry function, which intercepts calls to the driver's **DriverEntry** function and calls the driver's

own **DriverEntry** function. Drivers bind statically with this library at build time.

- `WdfMM000.sys` supports the KMDF DDIs. The name includes the KMDF major version; thus, `Wdf01000.sys` is a version 1 library, `Wdf02000.sys` is a version 2 library, and so forth. The co-installer installs this run-time library at driver installation, and drivers bind dynamically with it at load time.
- `Wdfldr.sys` is the KMDF loader, which loads the appropriate version of `WdfMM000.sys` and binds the client driver to it. The co-installer installs this library at driver installation.

KMDF is also distributed with a set of debugger extensions and a symbol file to aid in debugging.

Installation

KMDF drivers are installed by using an INF file and the redistributable KMDF co-installer. A hardware vendor's installation package thus includes:

- An INF file for the driver.
- The redistributable co-installer `WdfCoinstallerMMmmm.dll` (where *MM* is the major KMDF version number and *mmm* is the minor version number).
- A driver binary.
- An optional custom installation application.

The INF file includes a [**wdf**] section and references the co-installer. The co-installer includes the redistributable installation package (a .cab file) as a resource. The .cab file, in turn, includes the KMDF run-time library (`WdfMM000.sys`) and the KMDF loader (`WdfLdr.sys`). The co-installer is available to hardware vendors in the WDK and in general distribution releases (GDRs).

The `WdfCoinstallerMMmmm.dll`, the extracted .cab file, and the contents of the cabinet file are all signed components. Driver installation fails if the certificate with which the co-installer was signed is not available on the target system.

The Components variable in the **Wdf** section of the driver's INF file specifies the driver service name and the version of KMDF with which the driver was built.

Versioning and Dynamic Binding

When Windows loads a KMDF driver, the driver is dynamically bound to the KMDF run-time library (`WdfMM000.sys`). Multiple drivers can share the same run-time library (DLL) image, and the run-time libraries for two major versions can co-exist side by side.

When you build a KMDF driver, you link it with `WdfDriverEntry.lib`. This library contains information about the KMDF version in a static data structure that becomes part of the driver binary. The internal `FxDriverEntry` function in `WdfDriverEntry.lib` wraps the driver's **DriverEntry** routine, so that when the driver is loaded, `FxDriverEntry` becomes the driver's entry point. At load time, the following occurs:

1. `FxDriverEntry` calls the internal function `WdfVersionBind` (defined in `wdfldr.sys`) and passes the version number of the KMDF run-time library with which to bind.
2. The loader determines whether the specified major version of the framework library is already loaded. If not, it starts the service that represents the framework library and loads the library and the driver. If so, it adds the driver as

a client of the service and returns the relevant information to the FxDriverEntry function. If the driver requires a newer version of the run-time library than the one already loaded, the loader fails and logs the failed attempt in the system event log.

3. FxDriverEntry calls the driver's **DriverEntry** function, which in turn calls back to KMDF to create the KMDF driver object.

Although two major versions of KMDF can run side-by-side simultaneously, two minor versions of the same major version cannot. At installation, a more recent minor version of the KMDF run-time library overwrites an existing, older minor version. If the older version is already loaded when a user attempts to install a driver with a newer version, the user must reboot the system.

For a boot driver, the loading scenario is different because the KMDF run-time library must be loaded before the driver. At installation, the co-installer reads the INF (or the registry) to determine whether the driver is a boot driver. If so, the co-installer both changes the start type of the KMDF service so that the Windows loader starts it at boot time and sets its load order so that it is loaded before the client driver.

Resources

Windows Driver Foundation on the WHDC Web site

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

Current White Papers

Architecture of the Windows Driver Foundation

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

Introduction to the WDF User-Mode Driver Framework

http://www.microsoft.com/whdc/driver/wdf/UMDF_intro.mspx

Introduction to the Windows Driver Foundation

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

Introduction to Plug and Play and Power Management in the Windows Driver Foundation

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

DMA Support in Windows Drivers

<http://www.microsoft.com/whdc/driver/kernel/dma.mspx>

I/O Request Flow in WDF Kernel Mode Drivers

http://www.microsoft.com/whdc/driver/wdf/ioreq_flow.mspx

Sample Drivers for the Kernel Mode Driver Framework

<http://www.microsoft.com/whdc/driver/wdf/KMDF-samp.mspx>

I/O Completion/Cancellation Guidelines

<http://www.microsoft.com/whdc/driver/kernel/locancel.mspx>

Scheduling, Thread Context, and IRQL

<http://www.microsoft.com/whdc/driver/kernel/IRQL.mspx>

Locks, Deadlocks, and Synchronization

<http://www.microsoft.com/whdc/driver/kernel/locks.mspx>

Windows Driver Kit

<http://www.microsoft.com/whdc/driver/WDK/default.mspx>

WDK Documentation

Windows Driver Foundation

Kernel-Mode Driver Framework

Kernel-Mode Driver Architecture

Design Guide ("Windows Management Instrumentation")

Reference (In "Driver Support Routines," see "WMI Library Support Routines")

Driver Development Tools

Tools for Software Tracing ("WPP Software Tracing" and "Software Tracing
FAQ")