DMA Support in KMDF Drivers

October 25, 2007

Abstract

The paper describes how Microsoft® Windows® drivers support direct memory access (DMA) devices, using the Windows Driver Foundation (WDF) kernel-mode driver framework. Because understanding Windows DMA architecture is key to properly implementing DMA drivers on Windows, this paper also describes the Windows DMA architecture and the Windows DMA abstraction.

This information applies to drivers that support Microsoft Windows 2000 and later operating systems.

The current version of this paper is maintained on the Web at:   
<http://www.microsoft.com/whdc/driver/kernel/dma.mspx>

For comprehensive information about writing WDF drivers, see *Developing Drivers with the Windows Driver Foundation*, by Penny Orwick and Guy Smith, available at <http://www.microsoft.com/MSPress/books/10512.aspx>.

Contents

[Introduction 4](#_Toc163626758)

[Basic DMA Concepts and Terminology 4](#_Toc163626759)

[DMA Transactions and DMA Transfers 4](#_Toc163626760)

[Packet-Based and Common-Buffer DMA 5](#_Toc163626761)

[Packet-Based DMA Device Design 5](#_Toc163626762)

[Common-Buffer DMA Device Design 5](#_Toc163626763)

[Hybrid Device Designs 5](#_Toc163626764)

[Scatter/Gather Support 6](#_Toc163626765)

[DMA-Specific Device Information 6](#_Toc163626766)

[Device Information and DMA Driver Design 7](#_Toc163626767)

[DMA Design Type 7](#_Toc163626768)

[Device Addressing Capability 7](#_Toc163626769)

[Hardware Scatter/Gather Capability 7](#_Toc163626770)

[Maximum Transfer Length 7](#_Toc163626771)

[Buffer Alignment Requirements 8](#_Toc163626772)

[What Is Not a Consideration 8](#_Toc163626773)

[Windows DMA Abstraction 9](#_Toc163626774)

[DMA Operations and Processor Cache 10](#_Toc163626775)

[Completion of DMA Transfers by Flushing Caches 10](#_Toc163626776)

[Map Registers 10](#_Toc163626777)

[Map Registers: The Concept 10](#_Toc163626778)

[Map Registers: The Implementation 12](#_Toc163626779)

[When Map Registers Are Used 12](#_Toc163626780)

[System Scatter/Gather Support 13](#_Toc163626781)

[System Scatter/Gather: The Concept 13](#_Toc163626782)

[System Scatter/Gather: The Implementation 14](#_Toc163626783)

[DMA Transfer to Any Location in Physical Memory 14](#_Toc163626784)

[DMA Transfer to Any Location: The Concept 14](#_Toc163626785)

[DMA Transfer to Any Location: The Implementation 15](#_Toc163626786)

[Implementing DMA Drivers 16](#_Toc163626787)

[Driver DMA Initialization 17](#_Toc163626788)

[The DMA Enabler Object 17](#_Toc163626789)

[The Common-Buffer Object 18](#_Toc163626790)

[The DMA Transaction Object 18](#_Toc163626791)

[Example: Driver DMA Initialization 19](#_Toc163626792)

[Transaction Initiation 20](#_Toc163626793)

[Transaction Initialization 20](#_Toc163626794)

[Transaction Execution 21](#_Toc163626795)

[Example: Transaction Initiation 21](#_Toc163626796)

[Request Processing 23](#_Toc163626797)

[EvtProgramDma Function Definition 23](#_Toc163626798)

[EvtProgramDma Function Tasks 24](#_Toc163626799)

[Example: Request Processing 25](#_Toc163626800)

[DMA Completion Processing 26](#_Toc163626801)

[Transfer, Transaction, and Request Completion 26](#_Toc163626802)

[Example: DMA Completion Processing 27](#_Toc163626803)

[Testing DMA Drivers 28](#_Toc163626804)

[DMA-Specific Verification 28](#_Toc163626805)

[The !dma Debugger Extension 28](#_Toc163626806)

[KMDF Debugger Extensions for DMA 29](#_Toc163626807)

[Best Practices: Do’s and Don’ts for DMA Drivers 30](#_Toc163626808)

[Resources 30](#_Toc163626809)

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.

© 2006-2007 Microsoft Corporation. All rights reserved.

Microsoft and Windows 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.

# Introduction

Using direct memory access (DMA) for data transfers to or from a DMA-capable device has many advantages, including higher speed transfers and lower overall system CPU usage. The kernel-mode driver framework (KMDF) transparently handles much of the required work to implement DMA in a KMDF driver. Drivers are primarily responsible for specifying the capabilities of their device and initiating DMA operations.

This paper describes the basic concepts and terminology used in writing DMA drivers for Microsoft® Windows®. It also describes the details that you should know about the DMA implementation in a device before you start to write the driver.

The information in this paper applies only to KMDF drivers for DMA-capable devices.

# Basic DMA Concepts and Terminology

DMA is a method of transferring data between a device and main memory without intervention by the CPU. Instead of the CPU copying the data from one location to the other, the CPU initiates the copy operation and then is free to service other requests. Depending on the type of DMA device, either the device or a separate DMA controller actually copies the data. Devices such as network adapters are typically designed to support DMA, which improves both device and system performance.

Windows supports two types of DMA devices: bus-master devices and system devices, which are sometimes called “slave” devices.

Modern buses such as PCI Express typically support bus-master devices, which are now the most common type of DMA devices. A bus-master DMA device contains all of the required electronics and logic to take control of—or “master”—the bus on which it is located and to transfer data between the device’s buffer and the host’s system memory. Drivers for bus-master devices can take advantage of framework support for DMA.

System DMA devices are vestiges of the original IBM PC design and are typically supported by an ISA bus. These devices rely on a DMA controller chip on the motherboard to perform data transfers. KMDF does not support system DMA because modern devices that use system DMA are relatively scarce. Drivers for system DMA devices must use WDM.

**Note**  Unless stated otherwise, the term “DMA” in this paper refers only to bus-master DMA.

## DMA Transactions and DMA Transfers

Two terms are critically important to understanding DMA, DMA devices, and the Windows DMA model:

* **DMA transaction**A complete I/O operation that involves DMA, such as a single read or write request from an application.
* **DMA transfer**A single hardware operation that transfers data from main memory to a device or from the device to main memory.

When a KMDF driver receives an I/O request, it creates a DMA transaction object to represent the request. The DMA transaction might involve one or more transfers, depending on the size of the request. Internally, the framework determines whether the device can satisfy the entire transaction in a single transfer. If the transaction is too large, the framework divides the transaction into multiple transfer operations, each of which transfers a fragment of the requested data.

Thus, a DMA transfer is always associated with a single DMA transaction, but a DMA transaction can involve multiple DMA transfers.

## Packet-Based and Common-Buffer DMA

The two basic types of DMA device design are packet-based DMA and common-buffer DMA. Devices can also incorporate a hybrid design that has features of both packet-based and common-buffer DMA.

### Packet-Based DMA Device Design

Packet-based DMA devices are the most common design. In this design, the driver explicitly sets up and requests each data transfer from the device. Each DMA operation thus transfers a packet of data.

One example of a packet-based DMA device is a mass storage device. To read data from such a device, the driver programs the DMA device to read the required sectors and provides the device with a pointer to a buffer in system memory that will receive the data. The device generates an interrupt after the read operation is complete. When the interrupt occurs, the driver performs any required teardown and completes the request.

### Common-Buffer DMA Device Design

In a common-buffer DMA design, the driver allocates an area in system memory—the common buffer—that it shares with the DMA device. The format of this area is defined by the DMA device and is understood by the driver. The shared area can contain data structures that the driver uses to control the device, and it might also contain one or more data buffers. The device periodically checks the data structures in the shared buffer area and updates them by performing DMA transfers. Because these DMA transfers occur as required by the device without any specific initiating action by the driver, common-buffer DMA is sometimes referred to as “continuous DMA.”

One example of a common-buffer DMA device is an intelligent network adapter. The common buffer for such an adapter might contain a set of data buffers and a pair of circular lists: one for transmit and one for receive. Each entry in the receive list might contain a description of a corresponding data buffer—which can hold an incoming network message—and a status field that indicates whether the data buffer is available or whether it already contains a message.

### Hybrid Device Designs

Hybrid DMA device designs incorporate both packet-based and common-buffer DMA. These designs typically use both a host-resident shared memory buffer that contains control structures for the device—the common-buffer part of the design—and a set of descriptors that contain information about each data packet to be transmitted. Before each DMA transfer, software sets up the descriptors and then makes an entry in the common buffer for a new packet to transmit. The device then transmits the packet directly from the data buffer—the packet-based part of the design.

Unlike a pure common-buffer design, the driver does not copy the contents of the data buffer to the common buffer. After the device transmits the packet, the device generates an interrupt. After receiving the interrupt, the driver for the device determines which packets are complete, performs any required teardown, and completes the associated requests.

## Scatter/Gather Support

Scatter/gather is a DMA technique in which a single transfer includes data from multiple locations in physical memory. Some devices support scatter/gather DMA in hardware. For devices that do not have such hardware, Windows implements an internal scatter/gather mechanism in software.

Hardware support for scatter/gather DMA, also called DMA chaining, is particularly efficient on Windows systems. Because Windows uses demand-paged virtual memory, virtually contiguous data buffers are often not physically contiguous. That is, a buffer that is contiguous in virtual memory might actually map to multiple locations in physical memory. A device that supports scatter/gather DMA in hardware can complete the request in a single transfer, whereas a device that can perform DMA only from a single, contiguous memory location might be required to make several smaller transfers to complete a request.

The buffer for scatter/gather DMA is described by a scatter/gather list, which is simply a list that contains a base address and length for each physical location in the buffer. The number of entries in the list depends on the device, but in most modern devices the maximum number is unlimited.

KMDF drivers are not required to implement any special handling to support scatter/gather. During initialization, the driver fills in a configuration data structure that describes the capabilities of the device. The framework then uses the device’s hardware scatter/gather capabilities or the Windows software implementation, as appropriate.

# DMA-Specific Device Information

Writing a driver for a DMA device is somewhat more complicated than writing a driver for most other types of devices. The job is easier if you understand the design details of the device and how the driver can control it before you begin to write the driver. Table 1 shows a checklist of the required design information.

Table 1. DMA-Specific Device Information

| Device information | Possible values |
| --- | --- |
| DMA design type | Packet-based, common-buffer, or hybrid |
| Maximum addressing capability | 32-bit, 64-bit, or other |
| Hardware scatter/gather support | Yes or no |
| Maximum transfer length per DMA operation | Number of bytes |
| Buffer alignment requirement | None, word, longword, or quadword |

The information in Table 1 is important in determining overall driver I/O design, selecting a DMA profile for the device, and configuring the DMA enabler and transaction objects that the driver uses to perform DMA.

Because device implementations vary greatly, not all of these issues apply to your driver. Furthermore, it is impossible to discuss every issue you could possibly encounter. Check with the designer of your particular device, or with a hardware engineer who understands the details of your device’s design, for any issues that you should consider that are unique to your device.

## Device Information and DMA Driver Design

This section provides more detail on each of the items in Table 1. In particular, it discusses how the specific details of your device might influence the design of your DMA driver.

### DMA Design Type

The most important fact that you must know about your device is whether it uses a packet-based DMA design, a common-buffer DMA design, or a hybrid of the two. The DMA design type dictates the overall architecture and design of your driver.

It is usually harder to write drivers for common-buffer and hybrid DMA devices than for packet-based devices. The reason is that device designs that use common buffers require a complete understanding and careful coordination of the data structures that the driver and the device share. Some of the issues that you might need to identify include the following:

* How your driver communicates the base address of the common buffer to your device.
* What format is used for pointers that are stored in the common buffer, such as for linked lists.
* What types of synchronization are required between your driver and your device when updating structures in the common buffer area.

Designs that use common buffers can also introduce subtle side effects that a driver might be required to handle. For example, common-buffer devices perform frequent DMA transfers to or from system memory as the device hardware scans for changes to the shared data structures. Although this is not a problem in most desktop systems, on laptop systems such designs can have a negative impact on battery life. If you cannot avoid using a common-buffer DMA device design in a laptop, you should typically ensure that your driver implements an aggressive power-management scheme for the device. For example, you might design your driver to power down the device whenever the device is idle.

Some hybrid designs enable the device to be used as a completely packet-based device. Because drivers for packet-based DMA devices are typically less complex to write than those for common-buffer devices, you might consider first implementing a driver for a hybrid device by using the simpler packet-based interface and later adding support for the common-buffer area.

### Device Addressing Capability

Another important detail is the physical addressing capability of the device—specifically, whether the device is capable of 64-bit addressing. When you select the DMA profile for the device, you indicate whether it supports DMA transfers to 64-bit addresses.

### Hardware Scatter/Gather Capability

Newly designed devices for Windows should include hardware scatter/gather support because this can significantly reduce data transfer overhead and latency. You indicate whether your device supports hardware scatter/gather when you select its DMA profile.

### Maximum Transfer Length

Many devices have a maximum number of bytes that they can transfer in one DMA operation. If your device has such a limit, you must know what the limit is. Some devices have no limit on the maximum length of a DMA transfer. For these devices, you must determine a practical maximum transfer length that your driver will support.

Regardless of whether the limit is set by the device or by the driver, you must supply this value when you configure the DMA enabler object.

### Buffer Alignment Requirements

DMA devices often require that data buffers used in DMA transfers be aligned in particular ways. For example, one relatively common DMA support chipset requires data buffers used for DMA read and write operations to be aligned on a 16-byte boundary. This is because the chipset reserves the lowest 4 bits of each address for its own use. As with support for 64-bit addressing and hardware scatter/gather, the buffer alignment requirement helps to determine the appropriate DMA profile for the device.

## What Is Not a Consideration

Windows implements extensive system support for DMA according to the Windows DMA abstraction described later in this paper. Drivers that use KMDF methods for DMA are guaranteed to conform to the Windows DMA model and thus avoid a considerable amount of complicated code to handle issues such as the following:

* **Windows 64-bit support**Drivers that use the built-in KMDF DMA support are not typically required to implement any special code to work properly in the 64-bit virtual address space that 64-bit editions of Windows provides. Drivers should always use the 64-bit safe data types, such as ULONG\_PTR, to make their data size specification clear and unambiguous. Drivers that must differentiate between 32-bit and 64-bit callers may do so by calling the **WdfRequestIsFrom32BitProcess** method.
* **Amount of physical memory present**The capabilities of the device determine the size of the memory pointers that the framework uses, and not the physical addressing capability or the amount of memory that is available on a given system. The framework ensures that drivers always receive data buffer physical addresses that are within the addressing capabilities of their devices. Thus, a driver for a device that is capable of only 32-bit addressing will not receive a pointer with more than 32 significant bits.
* **Bus addressing capability**Drivers that use the built-in KMDF DMA support are never required to determine the addressing capability of the bus to which their device is attached, because the framework handles all addressing issues. For example, the driver for a PCI bus device that is capable of 64-bit addressing is not required to be aware of whether the device is attached to a 64-bit–capable PCI bus. As with the amount of physical memory present on the machine, the addressing capability of the bus is made transparent by the framework’s DMA implementation.

Bus addressing transparency is maintained for PCI buses with 64 address lines, as well as for PCI buses that support the dual address cycle (DAC) mode of 64-bit addressing.

The above features apply only to drivers that use the KMDF DMA support, which conforms to the implementation model based on the Windows DMA abstraction. Drivers that take shortcuts by not following the model—often in an unnecessary attempt at performance optimization—receive unpredictable levels of support for the features described above, depending on precisely which parts of the DMA model they use.

# Windows DMA Abstraction

The Windows DMA architecture presents an abstract view of the underlying system hardware. This view, called the Windows DMA abstraction, is created by the I/O manager, hardware abstraction layer, bus drivers, and any bus filter drivers that may be present. OEMs can customize some of these components, if required, to support unique features of their hardware; however, most Windows computers use standard hardware designs that are supported by the standard Windows DMA implementation.

The DMA support in KMDF is based upon the Windows DMA abstraction. By using the Windows DMA abstraction, the framework is not required to deal with the unique capabilities and organization of the underlying hardware platform. KMDF and the abstraction define a stable hardware environment that drivers can rely on. As a result, drivers do not require conditional compilation or runtime checks to support different underlying hardware platforms. The Windows DMA abstraction describes a system with the following major characteristics:

* DMA operations occur directly from system memory and cannot be assumed to be coordinated with the contents of the processor cache.

Thus, the framework is responsible for flushing any changed data residing in processor cache back to system memory before each DMA operation.

* At the end of each DMA operation, part of the transferred data may remain cached either by Windows or by one of the supporting hardware chipsets.

As a result, the framework is responsible for flushing this internal adapter cache after each DMA operation.

* The address space on any given device bus is separate from the address space of system memory.

Map registers convert between device-bus logical addresses and system memory physical addresses. This mapping allows the Windows DMA abstraction to support software scatter/gather.

In addition, the use of map registers ensures that DMA operations on any device bus can reach any location in system memory. For example, the framework is not required to perform any special processing to ensure that DMA operations on a PCI bus with 32-bit addressing can reach data buffers that are located in system memory above the 4-GB (0xFFFFFFFF) physical address mark.

The Windows DMA abstraction involves several key areas:

* DMA operations and processor cache
* Completion of DMA transfers by flushing caches
* Map registers
* System scatter/gather support
* DMA transfer to any location in physical memory

The remainder of this section describes the Windows DMA abstraction more fully.

For KMDF drivers, most of the details for DMA support are handled by the framework, so the information in this section is not critical for your driver development tasks. However, although you are not required to master all of the details of the abstraction to write a KMDF driver that supports DMA, a good understanding of the basic concepts can help you in designing and debugging your driver.

## DMA Operations and Processor Cache

In the Windows DMA abstraction, DMA operations bypass the system hardware cache and take place directly in system memory. Therefore, before performing each DMA operation, the framework is responsible for updating system memory by flushing any data back from the system hardware cache to system memory. When the driver requests a DMA transfer, the framework flushes the data before executing the transfer.

To implement the abstraction, Windows updates system memory only when necessary. On many systems, DMA operations properly reflect the contents of the system hardware cache when it is different from the contents in system memory. As a result, performing a write-back operation from the cache to system memory before a DMA operation is unnecessary. On these systems, the standard Windows DMA implementation does not actually perform any operation in response to a flush request before a DMA operation.

On machines or bus configurations in which the hardware does not ensure cache coherence for DMA operations—such as certain Intel Itanium systems—the standard Windows DMA implementation does the process-specific work that is necessary to ensure such coherency when the driver calls **WdfDmaTransactionExecute**.

## Completion of DMA Transfers by Flushing Caches

According to the Windows DMA abstraction, data can remain cached by Windows or by another system hardware component after a DMA transfer is complete. This caching is entirely transparent to both the framework and the driver. Therefore, after performing each DMA operation, the framework is responsible for flushing this data from the Windows internal cache to complete the DMA operation. The framework flushes this cache when the driver calls **WdfDmaTransactionDmaCompleted.**

To implement the abstraction, the framework notifies Windows when each DMA transfer is complete. This notification enables the operating system to properly complete any transfers that use map registers. The next section discusses map registers in detail.

## Map Registers

One of the major features of the Windows DMA abstraction is the use of map registers to translate addresses between the system address space and the logical address space that a device bus uses. This section describes the conceptual abstraction of map registers that the Windows DMA model uses and briefly discusses how map registers are realized in the standard Windows DMA implementation.

### Map Registers: The Concept

Figure 1 shows how system memory and device buses are connected in the Windows DMA abstraction. The device buses in the diagram might be two PCI buses, for example. Remember that Figure 1 is a conceptual diagram and is not intended to depict the physical hardware layout of any specific machine.

The figure shows two device buses, each of which is separate from the memory bus. In the Windows DMA abstraction, each device bus has an address space that is separate from the address space of all other device buses and that is also separate from the address space of system memory. The address space for a device bus is referred to as the device-bus logical address space for that bus.



Figure 1. Conceptually connecting device and memory bus by using map registers

A box labeled “Map Registers” connects each device bus to the memory bus in Figure 1. Because device-bus logical address space and system memory address space are separate, some component is required to translate addresses between them. That component is the group of map registers. Map registers translate addresses between the memory bus and the device bus so that data can flow between those two buses.

Map registers translate addresses in much the same way that the processor’s memory management registers—the page directory entries and page table entries—translate between processor virtual addresses and physical memory addresses. Each map register can translate up to a page of addresses in one direction. A page is 4K on x86 and x64 systems and 8K on Itanium systems and is defined as the PAGE\_SIZE constant.

Map registers are a shared resource that the Windows operating system manages. The system reserves map registers based on the maximum transfer length that the driver specifies when it creates the DMA enabler object. The framework allocates the map registers immediately before executing the transaction and frees them after the transaction is complete. Whenever a driver for a DMA device transfers data to or from system memory, the framework allocates and programs the map registers for the transfer.

Because each map register can translate a page of addresses, the number of map registers that a transfer requires depends on the transfer size, as shown by the following equation:

Number of Required Map Registers = (Transfer Size / PAGE\_SIZE) + 1

In the preceding equation, the additional map register added by the “+1” accounts for the case in which a transfer does not start at a page-aligned address.

Map registers are allocated in a contiguous block. Each block of map registers represents a contiguous set of device-bus logical addresses that can translate an equally sized set of system memory physical addresses in either transfer direction. Windows represents a block of map registers to the framework by a map register base value and a length. These values are not available to the KMDF driver.

### Map Registers: The Implementation

Because map registers are a conceptual abstraction that Windows uses, system hardware designers and the kernel-mode developers working with them can implement them in any way they choose. Over the history of Windows, a number of interesting implementations of hardware-based map registers have existed. Some of these designs used memory management logic to implement address mapping between the device bus and the memory bus in a manner similar to that shown in Figure 1.

Most modern computer systems running Windows use standard hardware designs and so use the standard Windows DMA implementation. In this implementation, map registers are realized entirely in software.

The standard Windows DMA components implement each map register as one PAGE\_SIZE buffer located in physical memory below 4 GB. Depending on the bus being supported, the map registers might even be located in physical memory below the 16-MB physical address mark. When a driver prepares to perform a DMA transfer, Windows determines whether any map registers are required to support the transfer and, if so, how many. Map registers might be needed to support transfer of an entire data buffer, or they might be needed only to support the transfer of some fragments of a data buffer.

If no map registers are required to support the transfer, Windows provides the framework with the physical memory address of a buffer fragment as the fragment’s device-bus logical address.

If the transfer requires map registers, Windows allocates buffer space for the duration of the transfer. If map registers are required but not enough buffer space is available, the operating system delays the request until sufficient buffer space becomes free. When a DMA transfer that uses map registers is complete, Windows frees the map registers, thus making the buffer space available to support another transfer.

### When Map Registers Are Used

Conceptually, the Windows DMA abstraction always uses map registers to translate between device-bus logical addresses and physical memory addresses.

In implementation, the standard Windows DMA components use map registers if either of the following is true:

* The driver for the DMA device indicates that the device does not support hardware scatter/gather.
* Any part of the buffer used for a given transfer exceeds the device’s addressing capability.

For example, if a DMA device can perform only 32-bit transfers, but part of the buffer being transferred is located above the 4-GB physical address mark, the system uses map registers.

## System Scatter/Gather Support

Map registers make possible special support to drivers for devices that do not implement hardware scatter/gather. This section describes the concepts behind the Windows system scatter/gather support and how the standard Windows DMA components implement system scatter/gather support.

### System Scatter/Gather: The Concept

On Windows systems, data buffers are rarely contiguous. If the buffers are not contiguous, performing DMA for a device that has no hardware scatter/gather support could easily require a lot of extra processing because a separate DMA transfer would be required for each physical fragment of a user buffer. However, system scatter/gather support makes such extra processing unnecessary.

Map registers, which convert device-bus logical addresses to system memory physical addresses and vice versa, map all data transfers between the device bus and the memory bus. To support scatter/gather in software, Windows allocates contiguous map registers for a given transfer. As a result, transfers that use map registers always appear to the device as a contiguous set of device-bus logical addresses, even if the corresponding pages in system memory are not physically contiguous.

A picture might help clarify this concept. Figure 2 shows how a series of contiguous map registers describes a fragmented data buffer. The standard Windows DMA components have programmed the map registers to point to various locations in the host’s physical memory. However, the device-bus logical addresses represented by the map registers are contiguous, so the system can use a single device-bus logical base address and length to describe them for the DMA device.



Figure 2. How a fragmented buffer is translated by using map registers

If this abstraction seems confusing, remember that map registers conceptually translate physical addresses in system memory to device-bus logical addresses for DMA transfers in precisely the same way as the system’s memory management hardware translates virtual addresses to physical addresses for program execution.

### System Scatter/Gather: The Implementation

Map registers support system scatter/gather by intermediate buffering of DMA transfers. During initialization, if a driver indicates that its device does not support hardware scatter/gather, the Windows DMA implementation uses map registers for all DMA transfers to or from the device. Thus, for such devices, a DMA transfer proceeds as follows:

1. The Windows DMA implementation allocates enough contiguous map registers—that is, low-memory buffers—to contain the data for the entire transfer.

If adequate buffer space is not available, the system delays the request until sufficient buffer space becomes free.

2. For a write operation—a transfer from system memory to the device—the implementation copies the data from the original data buffer to the map register buffer.

3. The Windows DMA implementation provides the framework with the physical memory address of the map register buffer as the buffer’s device-bus logical address.

The framework then passes this address to the driver in a scatter/gather list when it calls the driver’s *EvtProgramDma* callback function to program the device for DMA. Because the buffer is physically contiguous, the scatter/gather list contains only one element: the base address and length of the buffer. The driver uses that length and address—not the actual address of the data buffer fragments—to program the device for the DMA operation. In addition, because the map register buffers are located below the 4-GB physical address mark, such buffers are always within the addressing capability of any bus-master DMA device.

When the driver notifies the framework that the DMA transfer is complete, the framework in turn notifies the standard Windows DMA implementation, which determines whether map registers were used for the transfer and whether the operation was a read from the device to system memory. If so, Windows copies the contents of the map register buffers that were used to the original data buffer. Then Windows frees the map registers, making them available for another DMA transfer.

## DMA Transfer to Any Location in Physical Memory

By using map registers, the Windows DMA abstraction can support any transfer, regardless of the addressing capability of the device bus and the amount of system memory. As a result, even devices with only limited addressing capabilities can access all of physical memory under Windows.

This section describes how map registers make it possible for a DMA device to transfer data to any location in physical memory.

### DMA Transfer to Any Location: The Concept

Suppose DMA Device 1 in Figure 1 is a 32-bit bus-master DMA device on Device Bus A. Because Device 1 is only 32-bit capable, it can present only addresses from 0x00000000 to 0xFFFFFFFF on the device bus. This represents 4 GB of addressing capability. If Device Bus A were directly connected to system memory, it could only transfer data to or from data buffers in the low 4 GB of the system’s physical address space. Because of the way in which the Windows virtual memory system works, it is impossible to prevent a program that uses Device 1 from having its data buffers located above the 4-GB mark on a machine that has more than 4 GB of physical memory—or on any system that has memory located above the 4-GB physical address point. To solve this problem, every DMA transfer for Device 1 would be required to either use buffers that are specifically allocated below the 4 GB mark or implement special processing for the physical addresses of any fragments of user data buffers that were located out of its device’s addressing range.

The Windows DMA abstraction makes such special processing unnecessary. As part of setting up the DMA transfer, the Windows DMA functions allocate and program the map registers between Device Bus A and the memory bus to perform the required relocation. Map registers translate 32-bit device-bus logical addresses to 36-bit memory bus physical addresses in the same way in which memory management registers in certain x86 machines translate 32-bit virtual addresses to 36-bit physical addresses.

### DMA Transfer to Any Location: The Implementation

In the standard Windows DMA implementation, map registers are implemented entirely in software as contiguous PAGE\_SIZE buffers in system memory with a physical address less than 4 GB.

Each time a KMDF driver sets up a DMA transfer, the Windows DMA components determine whether map registers are required by checking the addressing capability of the device and the location of the data buffer that is being transferred. If any part of the data buffer is located outside the physical addressing capability of the device, the system uses map registers to perform the transfer. Thus, if a 32-bit DMA device sets up a transfer by using a data buffer that includes one or more fragments located above the 4-GB address mark, the transfer requires map registers.

Windows uses map registers only for the fragments of the data buffer that lie outside the device’s addressing range. If the entire fragment resides in memory that the device can address, Windows provides the physical memory address of the fragment as that fragment’s device-bus logical address.

If a fragment does not lie within the device’s addressing capability, the standard Windows DMA implementation performs the following actions:

1. Allocates a map register to contain the data within the fragment.

If sufficient buffer space is not available, the system delays the request until sufficient buffer space becomes free.

2. Copies the fragment’s data from the original data buffer to the map register buffer if the transfer is a write operation—that is, a transfer from system memory to the device.

3. Provides the physical memory address of the map register buffer as the fragment’s device-bus logical address.

The framework uses the physical memory addresses that the system provides to build a scatter/gather list. Each element of the list thus represents the physical memory address of either a data buffer fragment or a map register. When the framework calls the driver’s *EvtProgramDma* callback function, it passes the list as a parameter. The driver uses the elements of the scatter/gather list to program its device for the DMA operation.

When the driver and framework indicate that the DMA transfer is complete, the standard Windows DMA implementation determines whether it used map registers to support the transfer. If it did, and if the operation was a read from the device to system memory, the standard Windows DMA implementation copies the contents of any map register buffers that were used to the original data buffer. Windows then frees the map registers that were used for the transfer, making them available for another DMA transfer.

# Implementing DMA Drivers

Supporting DMA in a KMDF driver requires code in several of the driver’s event callback functions, as Figure 3 shows.



Figure 3. DMA implementation in KMDF drivers

As the figure shows, DMA-related processing takes place in four phases:

1. During driver initialization, typically in the *EvtDriverDeviceAdd* callback, the driver initializes and creates the DMA enabler object and common-buffer object that are required to support the DMA device.

2. When an I/O request arrives that requires DMA, the driver creates a transaction object if it has not already done so and initiates the DMA transaction.

This code typically is in the *EvtIoRead*, *EvtIoWrite*, or other I/O event callback, but might be in a different driver function if the driver has set up its queue for manual dispatching.

3. When the framework has set up the buffers that are required for the transfer, it calls the driver’s *EvtProgramDma* callback function.

This function programs the device hardware to perform a DMA transfer.

4. Each time the hardware completes a DMA transfer, the driver determines whether the entire transaction is complete, typically during the *EvtInterruptDpc* callback function.

If so, the driver completes the I/O request. If not, the framework prepares the next transfer and repeats phase 3.

The following sections describe each of these processing phases in detail, using sample code that is based on the PLX9x5x sample KMDF driver provided with the WDK. The sample driver supports a PCI device that has port, memory, interrupt and DMA resources. The device can be stopped and started at runtime and supports low-power states. The hardware has two DMA channels, so the driver uses one channel for reads and the other for writes. The driver configures two sequential queues: one for read requests and the other for write requests.

## Driver DMA Initialization

During initialization, typically in the *EvtDriverDeviceAdd* callback, the driver configures and creates the DMA enabler object and, if the device supports common-buffer DMA, the common buffer object.

If the driver has specified a packet-based DMA profile, it must serialize all of its DMA transactions because the framework allows only one packet-based DMA transaction to execute at any given time. To implement the serialization, the driver should either dispatch all of the I/O requests that require DMA from the same sequential queue or implement manual dispatching and call the framework to get the next I/O request only after the previous request is complete.

If the driver configures a sequential queue to dispatch requests for DMA I/O, it should also create the WDF DMA transaction object during initialization. In this case, only one such request is ever active at a given time, so the driver can create a single DMA transaction object during initialization and can reuse that object for each sequential DMA request.

### The DMA Enabler Object

The driver uses the DMA enabler object (that is, WDFDMAENABLER) to communicate with the framework about DMA transfers for a specific device object. The DMA enabler object maintains information about the DMA capabilities of the device.

Before creating the object, the driver must first initialize a WDF\_DMA\_ENABLER\_CONFIG structure by calling the WDF\_DMA\_ENABLER\_CONFIG\_INIT macro. The macro initializes the structure with a DMA profile and the device’s maximum transfer length.

The DMA profile indicates the device’s basic DMA characteristics, such as whether it supports 64-bit addressing and hardware scatter/gather. Most of the names of the available DMA profiles are self explanatory, but for completeness, Table 2 lists all of the profiles and their associated attributes.

Table 2. KMDF DMA Profiles and Their Meanings

| Profile name | Capable of 64-bit addressing | Supports hardware scatter/gather | Supports simultaneous read and write operations |
| --- | --- | --- | --- |
| **WdfDmaProfilePacket** | No | No | No |
| **WdfDmaProfileScatterGather** | No | Yes | No |
| **WdfDmaProfileScatterGatherDuplex** | No | Yes | Yes |
| **WdfDmaProfilePacket64** | Yes | No | No |
| **WdfDmaProfileScatterGather64** | Yes | Yes | No |
| **WdfDmaProfileScatterGather64Duplex** | Yes | Yes | Yes |

After initializing the structure, the driver creates the DMA enabler object by calling **WdfDmaEnablerCreate**.

### The Common-Buffer Object

If the device performs common-buffer DMA, the driver also must create a common-buffer object (that is, WDFCOMMONBUFFER) by calling **WdfCommonBufferCreate** or **WdfCommonBufferCreateWithConfig**. These methods are identical, except that **WdfCommonBufferCreateWithConfig** takes as an additional input parameter a WDF\_COMMON\_BUFFER\_CONFIG structure, which includes the alignment requirement for the buffer.

Next, the driver must get the system virtual address and the device-bus logical address of the common buffer by calling two additional WDF methods:

* **WdfCommonBufferGetAlignedVirtualAddress**, which returns the system virtual address of the common buffer.
* **WdfCommonBufferGetAlignedLogicalAddress**, which returns the device-bus logical address of the common buffer.

These addresses are required to program the device in the *EvtProgramDma* callback function.

### The DMA Transaction Object

The I/O requests for which the sample driver performs DMA are dispatched from a sequential queue. Therefore, the driver creates a DMA transaction object during initialization. Instead of creating and deleting a DMA transaction object for each request, the driver can reuse this object for each additional DMA transaction.

To create the object, the driver calls **WdfDmaTransactionCreate**,passing in a handle to the previously created DMA enabler object and receiving a handle to the newly created DMA transaction object. Both the framework and the driver use the DMA transaction object to manage the DMA operations for a given request.

### Example: Driver DMA Initialization

The example in Listing 1 supports a hybrid device, so it demonstrates the required initialization for both common-buffer and packet-based DMA support. This example is from the Sys\Init.c file in the PLX9x5x sample driver.

Listing 1. DMA initialization

WDF\_DMA\_ENABLER\_CONFIG dmaConfig;

WdfDeviceSetAlignmentRequirement( DevExt->Device, PCI9656\_DTE\_ALIGNMENT\_16 );

WDF\_DMA\_ENABLER\_CONFIG\_INIT( &dmaConfig,

WdfDmaProfileScatterGather64Duplex,

DevExt->MaximumTransferLength );

status = WdfDmaEnablerCreate( DevExt->Device,

&dmaConfig,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&DevExt->DmaEnabler );

if (!NT\_SUCCESS (status)) {

. . . //Error-handling code omitted

}

// Allocate common buffer for building writes

DevExt->WriteCommonBufferSize =

sizeof( DMA\_TRANSFER\_ELEMENT) \* DevExt->WriteTransferElements;

status = WdfCommonBufferCreate( DevExt->DmaEnabler,

DevExt->WriteCommonBufferSize,

WDF\_NO\_OBJECT\_ATTRIBUTES,

&DevExt->WriteCommonBuffer );

if (!NT\_SUCCESS(status)) {

. . . //Error-handling code omitted

}

DevExt->WriteCommonBufferBase =

WdfCommonBufferGetAlignedVirtualAddress(

DevExt->WriteCommonBuffer);

DevExt->WriteCommonBufferBaseLA =

WdfCommonBufferGetAlignedLogicalAddress(

DevExt->WriteCommonBuffer);

RtlZeroMemory( DevExt->WriteCommonBufferBase, DevExt->WriteCommonBufferSize);

WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE(&attributes, TRANSACTION\_CONTEXT);

status = WdfDmaTransactionCreate( DevExt->DmaEnabler,

&attributes,

&DevExt->ReadDmaTransaction);

if(!NT\_SUCCESS(status)) {

. . . //Error-handling code omitted

}

The example in Listing 1 performs the following tasks:

1. Sets the required alignment for the device object.

2. Initializes and creates a DMA enabler object.

3. Creates a common buffer.

4. Gets the addresses of the common buffer.

5. Creates a DMA transaction object.

The sample driver starts by setting the required alignment for the device object. The framework uses this value as the alignment for DMA if the driver does not specify an alignment requirement when it creates the common buffer.

Next, the driver initializes a WDF\_DMA\_ENABLER\_CONFIG structure by calling WDF\_DMA\_ENABLER\_CONFIG\_INIT. It specifies the DMA profile that best describes the device and the maximum transfer length that the device supports in a single DMA operation. The driver selects the **WdfDmaProfileScatterGather64** profile to indicate that the device supports both 64-bit DMA transfers and hardware scatter/gather.

With the WDF\_DMA\_ENABLER\_CONFIG structure initialized, the driver creates a new DMA enabler object by calling **WdfDmaEnablerCreate**. The driver stores the handle to the created object for later use.

This device uses a hybrid design—that is, it supports a combination of packet-based and common-buffer DMA, so the sample driver now creates a common buffer. It does this by calling **WdfCommonBufferCreate**, passing the length in bytes of the required common buffer. The allocated common-buffer area is contiguous in the device-bus logical address space, although not necessarily contiguous in the physical address space. By default, the common buffer has the same alignment that was specified earlier in the call to **WdfDeviceSetAlignmentRequirement**. Alternatively, the driver could call **WdfCommonBufferCreateWithConfig** to create the buffer and set the alignment requirement.

In addition to allocating the common-buffer space, **WdfCommonBufferCreate** and **WdfCommonBufferCreateWithConfig** allocate enough contiguous map registers to translate the physical addresses spanned by the common buffer to device-bus logical addresses. These methods also program those map registers to perform the necessary translations between logical and physical device bus addresses.

Next, the driver calls **WdfCommonBufferGetAlignedVirtualAddress** to get the kernel virtual address of the common buffer that it just created. The driver uses this address to manipulate the data structures in the common-buffer area that it shares with the device. The driver completes its DMA-specific initialization by calling **WdfCommonBufferGetAlignedLogicalAddress** to get the device-bus logical address of the common buffer.

Finally, the driver creates a DMA transaction object by calling **WdfDmaTransactionCreate**, passing a handle to the DMA enabler object. The driver uses this transaction object for all DMA read requests.

## Transaction Initiation

When the driver receives an I/O request that requires DMA, it initiates the DMA transaction. Typically, this code appears in the *EvtIoRead*, *EvtIoWrite*, or other I/O event callback, but if the driver manually retrieves I/O requests from a queue, the code might appear elsewhere.

### Transaction Initialization

Before the driver can initiate a DMA transaction, it must initialize the DMA transaction object with information about the requested transfer. The framework uses this information—along with the DMA profile that the driver previously supplied in the DMA enabler object—to calculate the number of required map registers for the transfer and to create the scatter/gather list that the driver uses to program the device.

If the driver has not already created a DMA transaction object to use for this transaction, it must first create a new DMA transaction object by calling **WdfDmaTransactionCreate**.

The driver can then initialize the transaction by calling either **WdfDmaTransactionInitializeUsingRequest** or **WdfDmaTransactionInitialize**.

If the driver has received an I/O request from the framework, it uses **WdfDmaTransactionInitializeUsingRequest** to initialize the transaction object with data from the request object. This method takes as input a pointer to the WDFREQUEST object to be processed, an enumeration constant that indicates whether the transfer moves data to or from the device, and a pointer to the driver’s *EvtProgramDma* callback.

If the driver performs common-buffer DMA or performs DMA transactions that are not based on an I/O request, it calls **WdfDmaTransactionInitialize** to initialize the transaction object. In addition to the direction of the transfer and a pointer to the *EvtProgramDma* callback, this method takes as input a pointer to an MDL that describes the buffer to use for the transfer, the virtual address of the buffer, and the buffer length. The driver calls **WdfRequestRetrieveInputWdmMdl** for a write request or **WdfRequestRetrieveOutputWdmMdl** for a read request to get a pointer to the MDL, and then calls kernel memory manager functions to get its virtual address and length.

### Transaction Execution

After it initializes the DMA transaction object, the driver can start processing the DMA transaction by calling **WdfDmaTransactionExecute**. Before beginning the DMA transaction, this method flushes any changed data in the processor cache back to system memory. It then calls the driver’s *EvtProgramDma* callback to request that the driver program the device for this DMA transfer.

### Example: Transaction Initiation

The following example draws again from the PLX9x5x sample driver. The code in Listing 2 shows the steps that a typical KMDF driver performs to initiate a DMA transfer. This code appears in the Read.c file.

Listing 2. DMA initiation

VOID PLxEvtIoRead(

IN WDFQUEUE Queue,

IN WDFREQUEST Request,

IN size\_t Length

)

{

NTSTATUS status = STATUS\_UNSUCCESSFUL;

PDEVICE\_EXTENSION devExt;

// Get the DevExt from the queue handle

devExt = PLxGetDeviceContext(WdfIoQueueGetDevice(Queue));

do {

// Validate the Length parameter.

if (Length > PCI9656\_SRAM\_SIZE) {

status = STATUS\_INVALID\_BUFFER\_SIZE;

break;

}

// Initialize the DmaTransaction.

status =   
 WdfDmaTransactionInitializeUsingRequest(

devExt->ReadDmaTransaction,

Request,

PLxEvtProgramReadDma,

WdfDmaDirectionReadFromDevice

);

if(!NT\_SUCCESS(status)) {

. . . //Error-handling code omitted

break;

}

// Execute this DmaTransaction.

status = WdfDmaTransactionExecute( devExt->ReadDmaTransaction,   
 WDF\_NO\_CONTEXT);

if(!NT\_SUCCESS(status)) {

. . . //Error-handling code omitted

break;

}

// Indicate that the DMA transaction started successfully.

// The DPC routine will complete the request when the DMA

// transaction is complete.

status = STATUS\_SUCCESS;

} while (0);

// If there are errors, clean up and complete the request.

if (!NT\_SUCCESS(status )) {

WdfDmaTransactionRelease(devExt->ReadDmaTransaction);

WdfRequestComplete(Request, status);

}

return;

}

The example in Listing 2 shows the driver’s *EvtIoRead* callback, which performs the following tasks to initiate a DMA transaction:

1. Gets a pointer to the device context area, which contains the handle to the DMA transaction object.

2. Validates the transfer length.

3. Initializes the transaction.

4. Starts the transaction.

The driver starts by calling the PLxGetDeviceContext accessor function to get a pointer to its WDFDEVICE context area. It then validates the length of the transfer.

The driver next calls **WdfDmaTransactionInitializeUsingRequest** to associate the request that the framework passed to its *EvtIoRead* callback with the DMA transaction object that it previously created. As input parameters, this function takes handles to both the I/O request object and the DMA transaction object. It also takes as input a transfer direction indicator—**WdfDmaDirectionReadFromDevice** or **WdfDmaDirectionWriteToDevice**—and a pointer to the driver’s *EvtProgramDma* callback, which is named PLxEvtProgramDma. **WdfDmaTransactionInitializeUsingRequest** validates the parameters for the request and sets up as much of the internal infrastructure as possible.

If **WdfDmaTransactionInitializeUsingRequest** completed successfully, the driver calls **WdfDmaTransactionExecute**. This method:

* Determines the length of the DMA transfer.

The length of the DMA transfer depends on whether the current I/O request can be satisfied with one transfer or whether, because of size constraints imposed by the device or constraints on the availability of mapping registers, the transaction must be divided into multiple transfers. If the framework can process the entire request in a single DMA transfer, then it does so. If not, the framework divides the transaction into multiple DMA transfers and processes them serially.

* Requests that Windows make the processor cache coherent with system memory for the purposes of a DMA request.
* Allocates and initializes the necessary resources to perform the transfer.

This step includes allocating and programming any necessary map registers and building the scatter/gather list that will be passed to the driver.

* Calls the driver’s *EvtProgramDma* callback, passing a pointer to the created list of device-bus logical base address and length pairs, so that the driver can program the device to initiate the DMA operation.

If an error occurs during initiation, the driver calls **WdfDmaTransactionRelease** to free the resources that the framework set up for the transaction without deleting the transaction object. The driver then completes the I/O request with an error status in the usual way.

If **WdfDmaTransactionExecute** determines that multiple DMA transfers are necessary to fulfill the DMA transaction, the framework performs these transfers serially. That is, the framework determines the length for the first transfer and calls the driver’s *EvtProgramDma* callback to program the device for that transfer. Later, after the first transfer is complete and the driver calls **WdfDmaTransactionDmaCompleted**, typically from its *EvtInterruptDpc* function, the framework determines whether the entire transaction has been completed. If not, the framework calculates the length of the next transfer and calls the driver’s *EvtProgramDma* callback again to perform the transfer. This cycle repeats until the entire DMA transaction is complete. The driver can then complete the associated I/O request.

## Request Processing

A driver’s *EvtProgramDma* callback function programs the DMA device to perform a transfer. The framework passes the driver a pointer to a scatter/gather list, which contains one or more pairs of a device-bus logical address and a length that together describe the transfer. The driver uses these address/length pairs to program the device for the DMA transfer.

### EvtProgramDma Function Definition

The following is the prototype for this callback function:

typedef

BOOLEAN

(\*PFN\_WDF\_PROGRAM\_DMA) (

IN WDFDMATRANSACTION Transaction,

IN WDFDEVICE Device,

IN WDFCONTEXT Context,

IN WDF\_DMA\_DIRECTION Direction,

IN PSCATTER\_GATHER\_LIST SgList

);

where:

*Transaction*

A handle to the DMA transaction object that represents the current DMA transaction.

*Device*

A handle to a framework device object.

*Context*

The context pointer that the driver specified in a previous call to **WdfDmaTransactionExecute**.

*Direction*

An enumeration constant of the WDF\_DMA\_DIRECTION type that indicates the direction of the DMA transfer operation.

*SgList*

A pointer to a SCATTER\_GATHER\_LIST structure.

The *EvtProgramDma* function should return TRUE if it successfully starts the DMA transfer, and FALSE otherwise.

### EvtProgramDma Function Tasks

After the framework sets up a transaction, it calls the driver’s *EvtProgramDma* callback function to perform a DMA transfer. This function should perform the following steps:

1. Determine the offset into the buffer at which to start the transfer.

2. Set up the addresses and lengths to use in programming the device.

3. Program the device and start the transfer.

4. Release or delete the transaction if errors occur.

Remember that a single DMA transaction can involve more than one DMA transfer operation. This could happen if the I/O request involves a large amount of data, if the device has a limited capacity to transfer data, or if system resources are so constrained that the size of the buffer or the number of map registers is limited.

If this is the first transfer to be performed for this request, the driver programs the device to start transferring data from the beginning of the buffer. However, if one or more transfers have already been performed for this transaction, the transfer typically must start at some offset from the beginning of the buffer. To determine the offset and, by inference, find out whether this is the first transfer, the driver calls **WdfDmaTransactionGetBytesTransferred**, passing a handle to the current DMA transaction object. The method returns the number of bytes that have already been transferred for the transaction or, if no transfers have been performed yet, it returns zero. The driver can use the returned value as the offset into the buffer.

After it determines the offset, the driver should set up the required data structures to program the device. The specific details vary from one device to another, but a typical DMA device requires a base address and a length for each component of the transfer. The *EvtProgramDma* callback receives the base/address length pairs in the scatter/gather list parameter. The number of elements in the scatter/gather list depends on the type of DMA being performed and the type of device. For packet-based DMA, the list contains a single pair. For a device that supports hardware scatter/gather, the list contains multiple pairs. The driver translates these pairs into a form that the device can understand.

Next, the driver programs the device and starts the transfer. Before accessing the device registers, the driver acquires the interrupt spin lock for the device. This spin lock raises IRQL to DIRQL for the device and thus ensures that the device does not attempt to interrupt while the driver is changing the register values. Because code that is protected by this lock runs at a high IRQL, the driver should hold the lock for a minimal length of time. The lock should only protect code that physically accesses the device registers. All calculations and setup should occur outside the lock.

If the driver successfully starts the DMA transfer, the *EvtProgramDma* callback returns TRUE. If errors occur, the driver should cancel the current transaction and return FALSE.

### Example: Request Processing

The example in Listing 3 is from the PLx5x9x sample’s *EvtProgramDma* callback function for read requests in the Read.c file. Much of this function is device-specific code that is not reproduced here.

Listing 3. Sample *EvtProgramDma* callback

BOOLEAN PLxEvtProgramReadDma(

IN WDFDMATRANSACTION Transaction,

IN WDFDEVICE Device,

IN WDFCONTEXT Context,

IN WDF\_DMA\_DIRECTION Direction,

IN PSCATTER\_GATHER\_LIST SgList

)

{

PDEVICE\_EXTENSION devExt;

size\_t offset;

PDMA\_TRANSFER\_ELEMENT dteVA;

ULONG\_PTR dteLA;

BOOLEAN errors;

ULONG i;

devExt = PLxGetDeviceContext(Device);

errors = FALSE;

// Get the number of bytes already transferred for this transaction.

offset = WdfDmaTransactionGetBytesTransferred(Transaction);

// Set up the addresses to use in programming the device

... //Device-specific code omitted

// Acquire the interrupt spin lock for the device and start DMA.

WdfInterruptAcquireLock( devExt->Interrupt );

... //Device-specific code that programs device registers for DMA.

WdfInterruptReleaseLock( devExt->Interrupt );

// If errors occur in the EvtProgramDma callback,

// release the DMA transaction object and complete the request.

if (errors) {

NTSTATUS status;

WDFREQUEST request;

(VOID) WdfDmaTransactionDmaCompletedFinal(Transaction,

offset,

&status);

// Get the associated request from the transaction.

request = WdfDmaTransactionGetRequest(Transaction);

WdfDmaTransactionRelease(Transaction);

WdfRequestCompleteWithInformation( request,

STATUS\_INVALID\_DEVICE\_STATE,

0);

return FALSE;

}

return TRUE;

}

First, the driver initializes its local variables and then calls **WdfDmaTransactionGetBytesTransferred** to determine how many bytes of data have already been transferred for this transaction. It uses the returned value to determine the offset into the buffer at which to start the transfer.

Next, the driver translates the address/length pairs from the scatter/gather list into a form that the device can use and sets up the data structures that it requires to program the device. Then the driver calls **WdfInterruptAcquireLock** to acquire the interrupt spin lock for the device, accesses the device registers to program the device, and calls **WdfInterruptReleaseLock** to release the spin lock.

If errors occur during device programming, the driver calls **WdfDmaTransactionDmaCompletedFinal**, which indicates to the framework that the transaction is complete but all of the data was not transferred. This method takes a handle to the transaction object, the number of bytes that were successfully transferred, and a pointer to a location to receive a status value. Although the method is defined as a Boolean, it always returns TRUE, so the sample driver casts it to VOID. The driver then releases the DMA transaction object for later reuse, completes the I/O request with the STATUS\_INVALID\_DEVICE\_STATE failure status, and returns FALSE from the callback function.

If the driver programs the device successfully, the callback function returns TRUE. The device interrupts to indicate that the transfer is complete.

## DMA Completion Processing

Typically, a DMA device signals an interrupt when it completes a transfer. The driver performs minimal processing in the *EvtInterruptIsr* callback function and queues an *EvtInterruptDpc* callback function. The *EvtInterruptDpc* callback is generally responsible for processing the completed DMA transfer.

### Transfer, Transaction, and Request Completion

When the device signals the completion of a DMA transfer, the driver must determine whether the entire transaction is complete. If so, it completes the I/O request and deletes or releases the DMA transaction object.

The framework sets up the individual DMA transfers and keeps track of the number of bytes of data that each transfer involves. When a transfer is complete, the driver notifies the framework by calling **WdfDmaTransactionDmaCompleted** or **WdfDmaTransactionDmaCompletedWithLength**. The driver passes a handle to the DMA transaction object and receives an NTSTATUS value from both methods. The only difference between the two methods is that **WdfDmaTransactionDmaCompletedWithLength** also takes an input parameter that supplies the number of bytes that the device transferred in the just-completed operation, which is useful for devices that report this information.

Both of these methods do the following:

* Flush any remaining data from the Windows cache.
* Free the shared resources, such as map registers, that it allocated to support the transfer.
* Determine whether the completion of this transfer also completes the entire DMA transaction.
* If so, the method returns TRUE; if not, the method returns FALSE and the STATUS\_MORE\_PROCESSING\_REQUIRED status.

If the entire transaction is complete, the driver completes the associated I/O request.

If the entire DMA transaction is not complete, one or more additional transfers are required to complete the transaction. The framework then allocates the necessary resources for the next transfer and calls the *EvtProgramDma* callback again to perform another transfer.

### Example: DMA Completion Processing

Once again using the PLX9x5x sample driver as a general guide, the code example in Listing 4 illustrates the steps that a typical KMDF driver performs to complete a DMA transfer. The sample code is based on the read completion processing in the driver’s *EvtInterruptDpc* callback in the Isrdpc.c file.

Listing 4. DMA completion processing

if (readComplete) {  
 BOOLEAN transactionComplete;

WDFDMATRANSACTION dmaTransaction;

size\_t bytesTransferred;

// Get the current Read DmaTransaction.

dmaTransaction = devExt->CurrentReadDmaTransaction;

// Indicate that this DMA operation has completed:  
 // This may start the transfer on the next packet if   
 // there is still data to be transferred.  
 transactionComplete =   
 WdfDmaTransactionDmaCompleted( dmaTransaction, &status );   
 if (transactionComplete) {  
 // Complete the DmaTransaction and the request.  
 devExt->CurrentReadDmaTransaction = NULL;  
 bytesTransferred =   
 ((NT\_SUCCESS(status)) ?   
 WdfDmaTransactionGetBytesTransferred(dmaTransaction): 0 );  
 WdfDmaTransactionRelease(dmaTransaction);

WdfRequestCompleteWithInformation(request, status, bytesTransferred);

}

}

This sample code fragment shows how a typical KMDF driver handles DMA transfer completion. The driver executes this code after the device interrupts to indicate that the read operation is complete.

The example begins by getting a handle to the DMA transaction object for the current read operation. The driver uses this handle to call **WdfDmaTransactionDmaCompleted**. This method notifies the framework that the current transfer for the DMA transaction is complete. It returns a Boolean value that indicates whether entire transaction is now complete and an NTSTATUS value that indicates success or failure.

If **WdfDmaTransactionDmaCompleted** returns TRUE, the driver completes the current request by setting the location that holds the handle for the current DMA transaction object in its device object context area to NULL. If the transfer completed successfully, the driver retrieves the number of bytes that the DMA transaction transferred by calling **WdfDmaTransactionGetBytesTransferred**. Now that the entire transaction is complete, the driver releases the DMA transaction object by calling **WdfDmaTransactionRelease**. Finally, the driver completes the I/O request in the usual manner, by calling **WdfRequestCompleteWithInformation** and passing the status and number of bytes transferred.

If **WdfDmaTransactionDmaCompleted** returns FALSE, the *EvtInterruptDpc* callback performs no more processing for this DMA transaction because the framework immediately calls the driver’s *EvtProgramDma* callback to process the next transfer associated with the transaction.

# Testing DMA Drivers

Three features can be helpful in testing drivers that support DMA: Driver Verifier, the **!dma** debugger extension, and the KMDF-specific debugger extensions for DMA.

## DMA-Specific Verification

In Windows XP and later Windows versions, Driver Verifier includes specific verification tests to detect improper use of various DMA operations. Driver Verifier includes checks for the following DMA-specific errors, which involve the underlying WDM structures rather than the WDF objects:

* Overrunning or underrunning the DMA memory buffer.

These errors can be made by the hardware or by the driver.

* Freeing the same common buffer, adapter channel, map register, or scatter/gather list more than once.
* Leaking memory by failing to free common buffers, adapter channels, map registers, scatter/gather lists, or adapters.
* Attempting to use an adapter that has already been freed and no longer exists.
* Failing to flush an adapter buffer.
* Performing DMA on a pageable buffer.
* Allocating too many map registers at one time or allocating more map registers than the maximum allowed number.
* Attempting to free map registers while some are still mapped.
* Attempting to flush a map register that has not been mapped.
* Calling DMA routines at an improper IRQL.

In addition to the above, Driver Verifier includes several minor consistency checks, For more information on the consistency checking in Driver Verifier, see the WDK, which is listed in the Resources section.

Many of these errors pertain to actions that the framework, rather than the KMDF driver, performs. However, some such errors can indicate problems in driver code as well.

Driver Verifier implements one additional feature to facilitate testing of drivers for DMA devices. When Driver Verifier runs, it double-buffers all DMA transfers for verified drivers. Double-buffering helps to ensure that the driver uses the correct addresses for DMA operations. Depending on the kind of error it discovers, Driver Verifier reports DMA-specific errors either by generating an ASSERT or by issuing Bug Check 0x6E (that is, DRIVER\_VERIFIER\_DMA\_VIOLATION).

When you test your driver, remember that Driver Verifier is not an automated test utility. Rather, enabling verification for your driver causes Driver Verifier to watch the operations that your driver performs and warn you if it detects any operations that your driver performs incorrectly. Therefore, you must ensure that your driver is thoroughly exercised with a wide variety of I/O requests while Driver Verifier is enabled, including read and write requests of various lengths.

## The !dma Debugger Extension

Another debugging aid for drivers that support DMA devices is the **!dma** kernel debugger extension. This extension displays information about the DMA subsystem and DMA device drivers that are being verified by Driver Verifier.

If DMA verification is not enabled for a driver, the **!dma** debugger extension can list all of the DMA adapters in the system. The DMA adapter object is the WDM representation of the DMA capabilities of the device. A WDF DMA enabler object represents one or more underlying DMA adapters for a device. When DMA verification is enabled, the **!dma** debugger extension can also list the following information:

* Device object, map registers, scatter/gather lists, and common buffers that are associated with each DMA adapter.
* Map register usage for a particular DMA adapter, including whether transfers to and from the device are being double-buffered.
* Length, virtual and physical addresses of any common-buffer segments.

The information pertains to the underlying framework data structures, not to the objects that the KMDF driver itself creates. Nevertheless, it can be useful in debugging to help you determine whether DMA is being performed as you expect.

For more information on the **!dma** debugger extension, see the documentation that accompanies the Debugging Tools for Windows package, which is listed in the Resources section.

## KMDF Debugger Extensions for DMA

The KMDF debugger extensions that are provided with the WDK include several commands that are specifically designed to help you debug drivers for DMA drivers. Table 3 lists these extensions.

Table 3. KMDF Debugger Extensions for DMA Verification

| Extension | Related object | Description |
| --- | --- | --- |
| **!wdfdmaenablers** | WDFDEVICE | Lists all of the DMA enablers and their transactions and common buffer objects. |
| **!wdfdmaenabler** | WDFDMAENABLER | Dumps information about a specific DMA enabler object and its transactions and common buffer objects. |
| **!wdftransaction** | WDFDMATRANSACTION | Dumps information about a given transaction object. |
| **!wdfcommonbuffer** | WDFCOMMONBUFFER | Dumps information about a given common buffer object. |

Listing 5 shows the output of the DMA debugger extensions.

Listing 5. Output of KMDF debugger extensions for DMA

0: kd> !wdfdmaenablers 0x7e6128a8

Dumping WDFDMAENABLERS 0x7e6128a8

===============================

1) !WDFDMAENABLER 0x7e7ac630:

---------------------------

List of Transaction objects:

1)!WDFDMATRANSACTION 0x016f5fb0

List of CommonBuffer objects:

1)!WDFCOMMONBUFFER 0x7e887c00

2)!WDFCOMMONBUFFER 0x7e780cd0

3)!WDFCOMMONBUFFER 0x7ec6b338

0: kd> !WDFDMAENABLER 0x7e7ac630

Dumping WDFDMAENABLER 0x7e7ac630

===============================

Profile : WdfDmaProfileScatterGather64Duplex

Read DMA Description:

WDM AdapterObject (!dma): 0x81b37938

Maximum fragment length : 32768

Number of map registers : 9

Write AdapterObject Description:

WDM AdapterObject (!dma): 0xff0b92d0

Maximum fragment length : 32768

Number of map registers : 9

Default common buffer alignment: 1

Max SG-elements per transaction: 0xffffffff

List of Transaction objects:

1)!WDFDMATRANSACTION 0x016f5fb0

State: FxDmaTransactionStateTransfer

Direction: Read

Associated WDFREQUEST: 0x0108b2d0

Input Mdl: 0x81b042b8

Starting Addr: 0x81893000

Mdl of current fragment being transferred: 0x81b042b8

VA of current fragment being transferred: 0x81893000

Length of fragment being transferred: 0x8c

Max length of fragment: 0x8000

Bytes transferred : 0x0

Bytes remaining to be transferred: 0x0

ProgramDmaFunction: (0xf7cc62e0) testdma!EvtProgramDma

List of CommonBuffer objects:

1)!WDFCOMMONBUFFER 0x7e887c00

Virtual address allocated: 0xff8ca560 Aligned: 0xff8ca560

Logical address allocated: 0x18b79560 Aligned: 0x18b79560

Length allocated: 0x1a Length requested: 0xa

Alignment Mask: 0x10

# Best Practices: Do’s and Don’ts for DMA Drivers

Here are some reminders and tips regarding writing DMA drivers for Windows:

* Strive to understand the Windows DMA abstraction.

Understanding the abstraction helps you to write and debug your DMA driver.

* Do not bypass the KMDF DMA support or the Windows DMA abstraction in an attempt to create your own solution.

For example, it is never acceptable to build the contents of a scatter/gather list manually by interpreting the contents of an MDL and to use the resulting data to program your DMA device, because this does not account for the possible use of map registers.

* If your driver gets DMA requests from a sequential queue, and thus can ensure that only one such transaction is active at any time, design the driver to create a single DMA transaction object during driver DMA initialization and reuse this object for each DMA transaction.
* Test your driver with Driver Verifier enabled in general and with DMA verification enabled specifically.

In addition to designing your driver to properly use the DMA support built into KMDF, thorough testing with Driver Verifier is probably the single most important thing you can do to ensure that your driver functions properly in Windows.

# Resources

Windows Driver Foundation (WDF) on the WHDC Web site  
<http://www.microsoft.com/whdc/driver/wdf/default.mspx>

Windows Driver Kit (WDK)  
<http://www.microsoft.com/whdc/DevTools/ddk/default.mspx>

Debugging Tools for Windows  
<http://www.microsoft.com/whdc/devtools/debugging/default.mspx>

#### Books

*Developing Drivers with the Windows Driver Foundation*, by Penny Orwick and Guy Smith  
 <http://www.microsoft.com/MSPress/books/10512.aspx>

#### White Papers

Architecture of the Kernel-Mode Driver Framework  
[http://www.microsoft.com/whdc/driver/wdf/kmdf-arch.mspx](http://go.microsoft.com/fwlink/?LinkId=55233)

#### Samples in the WDK

PLX9x5x sample

%wdk%\src\kmdf\Plx9x5x

#### WDK Documentation

Handling DMA Operations in Framework-Based Drivers <http://go.microsoft.com/fwlink/?LinkId=80073>

DMA Verification (Driver Verifier) <http://go.microsoft.com/fwlink/?LinkId=80070>