A COM QuickStart for UMDF Developers

Updated November 12, 2007

Abstract

This paper is a brief and very basic introduction to COM programming, as used by user-mode driver framework (UMDF) drivers. It provides the minimum information about COM that a developer needs to know to implement a UMDF driver.

This information applies for the following operating systems:
 Windows Vista
 Microsoft Windows XP

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

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

Contents

[Introduction 3](#_Toc146351823)

[COM Basics 3](#_Toc146351824)

[How to Use UMDF COM Objects 5](#_Toc146351825)

[How to Call an Interface's Methods 5](#_Toc146351826)

[How to Obtain an Interface on a UMDF Object 6](#_Toc146351827)

[Receive an Interface through a Driver Method 6](#_Toc146351828)

[Call a UMDF Object Creation Method 6](#_Toc146351829)

[Call QueryInterface to Request a New Interface 7](#_Toc146351830)

[Reference Counting 7](#_Toc146351831)

[How to Implement the Basic Infrastructure 8](#_Toc146351832)

[DllMain 8](#_Toc146351833)

[DllGetClassObject 9](#_Toc146351834)

[The Driver Object's Class Factory 9](#_Toc146351835)

[How to Implement a UMDF Callback Object 10](#_Toc146351836)

[How to Implement the Class 10](#_Toc146351837)

[How to Implement IUnknown 10](#_Toc146351838)

[How to Implement AddRef and Release 10](#_Toc146351839)

[How to Implement QueryInterface 11](#_Toc146351840)

[How to Implement UMDF Interfaces 11](#_Toc146351841)

[Resources 11](#_Toc146351842)

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, MSDN, Windows, 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.

#### Revision History

|  |  |
| --- | --- |
| **Date** | **Change** |
| 10/30/2007 | * Correct URL stutter error
 |
| 08/31/2006 | Created |

# Introduction

The Microsoft® Windows® Driver Foundation (WDF) user-mode driver framework (UMDF) is based on the Component Object Model (COM), a user-mode object-oriented programming model that has been used extensively by Windows applications and Windows itself for over a decade. COM-based applications have a reputation for being large, complex, and difficult to implement. However, much of this complexity is imposed by COM run time and the requirements of the applications; it is not intrinsic to COM. UMDF does not use COM run time and uses only the essential core of the COM programming model. It takes advantage of the key features of COM while keeping UMDF drivers lightweight and relatively straightforward to implement.

This paper is intended to equip you with the minimum that you need to know about using and implementing COM objects to understand the UMDF documentation and samples and start implementing UMDF drivers. For a more thorough introduction to COM, see the white paper titled *An Introduction to COM for UMDF Developers*. This paper does not cover the details of implementing actual UMDF drivers, except in passing. For more information, see "Resources" at the end of this paper.

This paper discusses the three basic aspects of COM programming that you must understand to implement a UMDF driver:

* How to use the COM objects that the UMDF run time provides.
* How to implement the required basic infrastructure to load the UMDF driver.
* How to implement a UMDF *callback object*.

UMDF drivers are typically written with C++, so this paper assumes a reasonable familiarity with object-oriented programming (OOP) in general and C++ programming in particular, including:

* Class structure, including the **struct** and **class** keywords, public and private members, static methods, constructors, destructors, and pure abstract classes.
* Object creation and destruction that use the **new** and **delete** operators.
* Inheritance, including base and derived classes, multiple inheritance, and pure virtual methods.

It is also possible—although significantly less convenient—to use and implement COM objects with C. For further discussion, see the book entitled *Inside OLE*.

If you are unfamiliar with C++ programming, you should familiarize yourself with these topics. Other aspects of C++—such as operator overloading or templates—are not necessary for UMDF drivers. UMDF developers can use the Active Template Library (ATL) to simplify some aspects of driver implementation. However, ATL is not required and is not used in this paper. For more information on ATL, see the book titled *Inside ATL*, or the ATL documentation in the MSDN® library.

# COM Basics

Before starting, consider the following essential information about COM objects:

* There are no “raw” object pointers in COM, as there are in other OOP models. Instead, COM objects expose interfaces, which are groups of related methods. An object typically exposes at least two and sometimes many interfaces. When you obtain a COM object, you are given a pointer to one of the object's interfaces, not the object itself.
* All COM interfaces derive from **IUnknown,** the core COM interface. It is exposed by every COM object and is essential to the object's operation.
* COM interfaces are identified by globally unique identifiers (GUIDs), referred to as IIDs. They are used for such purposes as requesting an interface pointer. Some COM objects also have GUID identifiers, which are referred to as CLSIDs.
* An interface pointer allows you to use any of the methods on the interface. However, it does not provide access to the methods on any other interfaces that the object might expose. You must use the **IUnknown::QueryInterface** method to obtain the appropriate interface pointer.
* COM objects cannot expose public data members. They must expose data through methods that are called *accessors*. Accessors are just another method, but they are usually distinguished from task-oriented methods by a naming convention. UMDF uses a Get/Retrieve or Set/Assign prefix for its read and write accessors, respectively.

Figure 1 shows the logical relationship between an object and its contents.



Figure 1. COM objects, interfaces, and methods

All access to COM objects is through a virtual function table—commonly called a VTable—that defines the physical memory structure of the interface. The VTable is an array of pointers to the implementation of each of the methods that the interface exposes. When a client gets a pointer to an interface, it is actually a pointer to the VTable pointer, which in turn points to the method pointer. For example, Figure 2 shows the memory structure of the VTable for **IWDFIoRequest**.



Figure 2. VTables and interface pointers

The VTable is exactly the memory structure that many C++ compilers create for a pure abstract base class. This is one of the main reasons that COM objects are normally implemented in C++, with interfaces declared as pure abstract base classes. You can then use C++ inheritance to implement the interface in your objects, and the VTable is created for you by the compiler.

**Note:** The relationship between pure abstract base classes and the VTable layout that is shown in Figure 2 is not intrinsic to C++; it is a compiler implementation detail. Microsoft C++ compilers always produce the correct VTable layout.

# How to Use UMDF COM Objects

A process that uses a COM object is known as a COM client. Both UMDF drivers and the UMDF run time function as COM clients:

* UMDF drivers interact with UMDF run time by using UMDF-provided COM objects. For example, the UMDF device object represents the device and drivers can use the object for tasks such as setting or retrieving the device's Plug and Play state.
* The UMDF run time interacts with drivers through the driver-provided COM-based callback objects. For example, a driver can create one or more queue callback objects to handle I/O requests. The UMDF run time uses those objects to pass requests to the driver.

## How to Call an Interface's Methods

After you get a pointer to an interface, you can call the interface methods by using the same syntax that is used for a pointer to a C++ method. For example, if *pWdfRequest* is a pointer to an **IWDFIoRequest** interface, the following code is an example of how to invoke the interface's **Send** method:

HRESULT hr;

...

hr = pWdfRequest->Send( m\_pIUsbTargetDevice,

 WDF\_REQUEST\_SEND\_OPTION\_SYNCHRONOUS,

 0);

The method's return value is an HRESULT, a typical return type for COM methods. HRESULT is similar to the NTSTATUS type that kernel-mode drivers use as a return value and is used in much the same way. It is important not to think of HRESULTs as error values. Methods sometimes have multiple return values for success as well as for failure. You can determine the result of calling a method by comparing the returned HRESULT to the list of possible values in the reference documentation. However, be aware that these lists are not always complete. Use the error-checking macros that are discussed below to ensure that you do not miss a possible return value.

You can also test an HRESULT for simple success or failure. COM provides two macros for that purpose that work much like the NT\_SUCCESS macro. For an HRESULT return value of *hr*:

* FAILED(*hr*) returns TRUE for failure and FALSE for success.
* SUCCEEDED(*hr*) returns FALSE for failure and TRUE for success.

**Important:** Although NTSTATUS and HRESULT are similar, they are not interchangeable. Occasionally information in the form of an NTSTATUS value must be returned as an HRESULT. In that case, you can use the HRESULT\_FROM\_NT macro to convert the NTSTATUS value into an equivalent HRESULT. Do not use this macro for an NTSTATUS value of STATUS\_SUCCESS. In that case, return the S\_OK HRESULT value.

## How to Obtain an Interface on a UMDF Object

You can obtain an interface on a UMDF object in one of three ways:

* The UMDF run time passes an interface pointer in to one of the driver's callback methods.
* The driver creates a new WDF object by calling a UMDF object creation method.
* The driver calls **IUnknown::QueryInterface** to request a new interface from an existing WDF object.

### Receive an Interface through a Driver Method

The first case is the simplest. For example, when the UMDF run time calls a driver's **IDriverEntry::OnDeviceAdd** method, it passes a pointer to the device object's **IWDFDriver** interface. The following example is from the Skeleton sample.

HRESULT CMyDriver::OnDeviceAdd(

 \_\_in IWDFDriver \*FxWdfDriver,

 \_\_in IWDFDeviceInitialize \*FxDeviceInit

 )

{

 //Install the driver in the device stack

}

You can then use *FxWdfDriver* to access the methods on the driver object's **IWDFDriver** interface. Do not release *FxWdfDriver* when you are finished with it. The caller ensures that the object remains valid during the scope of the method call.

### Call a UMDF Object Creation Method

Sometimes a client must explicitly create a WDF object by calling the appropriate UMDF object creation method. For example, to create a request object, call the UMDF device object's **IWDFDevice::CreateRequest** method.

If you look at the UMDF reference in the Windows Driver Kit (WDK), you will find syntax like that for **IWDFDevice::CreateRequest**:

HRESULT CreateRequest(

 IN IUnknown\* pCallbackInterface,

 IN IWDFObject\* pParentObject,

 OUT IWDFIoRequest\*\* ppRequest

 );

*ppRequest* is an OUT parameter that provides an address at which the **CreateRequest** method can store a pointer to the newly created request object's **IWDFObject** interface. The following procedure and sample show how to handle such parameters, by using a call to **CreateRequest** by the UMDF's fx2\_driver sample as an example.

1. Declare a variable, *pWdfRequest*, to hold a pointer to **IWDFIoRequest**.

2. Pass a reference to *pWdfRequest* to **CreateRequest**.

IWDFIoRequest \*pWdfRequest = NULL;

...

hr = m\_FxDevice->CreateRequest( NULL,

 NULL,

 &pWdfRequest);

When **CreateRequest** returns, *pWdfRequest* holds a pointer to an **IWDFIoRequest** interface. When the caller has finished with *pWdfRequest*, it should release the interface pointer by calling **IUnknown::Release**.

### Call QueryInterface to Request a New Interface

Objects can expose more than one interface. Sometimes you have a pointer to one interface and need a pointer to another interface on the same object. In that case, call **IUnknown::QueryInterface** to request the desired pointer. Pass **QueryInterface** the IID of the desired interface and the address of an interface pointer, and **QueryInterface** returns the requested pointer. When the caller is finished with the interface pointer, the caller should release it.

The following example is from the fx2\_driver sample. It requests an **IWDFIoTargetStateManagement** interface pointer from the UMDF's I/O target object. This example uses the IID\_PPV\_ARGS macro—declared in objbase.h—which takes an interface pointer and produces the correct arguments for **QueryInterface**.

VOID CMyDevice::StartTarget(IWDFIoTarget \* pTarget)

{

 IWDFIoTargetStateManagement \* pStateMgmt = NULL;

 HRESULT hrQI = pTarget->QueryInterface(IID\_PPV\_ARGS(&pStateMgmt));

 ...

}

**Note:** **QueryInterface** belongs to the **IUnknown** interface. However, as shown above, there is no need to have an explicit pointer to an object's **IUnknown** interface to call **QueryInterface**. All interfaces inherit from **IUnknown**, so you can use any interface to call **QueryInterface**.

## Reference Counting

Unlike C++ objects, a client does not directly manage the lifetime of a COM object. Instead, a COM object maintains a reference count on itself. When a client creates a new object with an object-creation method, the object has a reference count of 1. Each time the client requests an additional interface on the object, the object increments the reference count. When a client is finished with an interface, it releases the interface pointer, which decrements the reference count. When all the interface pointers on the object have been released, the reference count is zero and the object destroys itself.

**Important:** You must be extremely careful about handling reference counts when you use or implement COM objects. Although clients do not explicitly destroy COM objects, there is no garbage collection to take care of the problem automatically as there is with managed code. A common mistake is to fail to release an interface. In that case, the reference count never goes to zero and the object remains in memory indefinitely. Conversely, releasing the interface pointer too many times causes the object to be destroyed prematurely, which can cause a crash. Failure to correctly manage reference counts is a common cause of memory leaks in COM-based applications, along with a variety of other problems. Even worse, bugs that are caused by mismanaged reference counts can be very difficult to locate.

The following are some basic rules for reference counting:

* Release any interface pointer that is passed to you as an OUT parameter when you are finished with it by calling **IUnknown::Release**. Do not release pointers that are passed as IN parameters. A common practice to ensure that all interface pointers are properly released is to initialize all pointers to NULL. Then set them to NULL again when they are released. That convention allows you to test all the interface pointers in your cleanup code; any non-NULL pointers are still valid and should be released.
* The reference count is usually incremented for you. The main exception is when you make a copy of an interface pointer. In that case, call **IUnknown::AddRef** to explicitly increment the object's reference count. You must then release the pointer when you are finished.
* When you discover that the driver has reference counting problems, do not attempt to fix them by simply adding calls to **AddRef** or **Release**. Make sure that the driver is acquiring and releasing references according to the rules. Otherwise you may find, for example, that the **Release** call that you added to solve a memory leak occasionally deletes the object prematurely and instead causes a crash.

**Note:** As with **QueryInterface**, you do not need a pointer to the object's **IUnknown** interface to call **AddRef** or **Release**. You can call these methods from any of the object's interfaces.

# How to Implement the Basic Infrastructure

This section discusses the required basic infrastructure to support UMDF drivers. A good starting point for your implementation is to take the Skeleton sample and modify that code to suit your driver's needs. That code should require at most only modest changes to adapt it to your driver's requirements.

## DllMain

A dynamic-link library (DLL) can contain any number of in-process COM objects, but it must have a single entry point that is named **DllMain**. Windows calls **DllMain** after the driver binary has been loaded into a host process and before it is unloaded. The function is also called when threads are created or destroyed. The *dwReason* parameter indicates why the function was called.

* When a UMDF driver's **DllMain** function is called for DLL loading or unloading, it should perform only simple module-wide initialization and termination tasks, such as initializing or freeing global variables and registering or unregistering Windows Software Trace Preprocessor (WPP) tracing. There are a number of things that **DllMain** should definitely not do, such as calling **LoadLibrary**.
* When a UMDF driver's **DllMain** function is called for thread creation or destruction, it can ignore the call.

For more information, see the function's reference page in the Platform Software Developers Kit (SDK). For a typical UMDF **DllMain** implementation, see dllsup.cpp from the UMDF's Skeleton sample.

## DllGetClassObject

Because class factories aren't exported by name, there is no direct way for a client to get access to them. Instead, the DLL exports the **DllGetClassObject** function by name, which allows it to be called by any client with access to the DLL. For many COM DLLs, including the UMDF samples, **DllGetClassObject** is the only function that is listed in the project's .def file to be exported by name from the DLL.

When a client wants to create an instance of one of the COM objects in the DLL, it passes the CLSID of the desired class factory object to **DllGetClassObject** and the IID of the desired interface, usually **IClassFactory**. **DllGetClassObject** creates a new class factory object and returns a pointer to the appropriate interface on the object. The client can then use the **IClassFactory::CreateInstance** method to create an instance of the object. For a typical implementation of **DllGetClassObject**, see dllsup.cpp from the UMDF's Skeleton sample.

A standard COM server is also required to implement **DllCanUnloadNow** and, optionally, **DllRegisterServer** and **DllUnregisterServer.** These exports are not required for UMDF drivers.

## The Driver Object's Class Factory

Some COM objects must be created by external clients. For UMDF drivers, there is usually only one such object: the driver callback object. A COM object that can be created by an external client must have a class factory. This is a small specialized COM object whose sole purpose is to create a new instance of its associated COM object and return a pointer to a specified interface. For a typical implementation of a class factory, see comsup.cpp from the UMDF's Skeleton sample.

Class factories usually expose only one interface in addition to **IUnknown**, **IClassFactory**. The **IClassFactory** interface has two members:

* **CreateInstance** creates an instance of the object and returns the requested interface pointer to the client.
* **LockServer** can be used to keep the DLL in memory. UMDF class factories typically have only a token implementation because UMDF does not use **LockServer**.

The following are some recommendations for implementing **CreateInstance**:

* Ignore the first parameter. Its purpose is to support COM aggregation, which is not used by UMDF.
* Create a new driver callback object by whatever means is convenient. The Skeleton sample puts the object creation code in a static method on the class that implements the callback object.
* Return the appropriate interface as an OUT parameter. At this point, the object should have a reference count of 1.

# How to Implement a UMDF Callback Object

A UMDF driver consists in large part of a collection of COM callback objects. These objects respond to notifications by the UMDF run time and allow the driver to process various events, such as read or write requests. All callback objects are in-process COM objects. This means that they are packaged in a DLL and run in the process context of a UMDF host.

The basic requirements for implementing UMDF callback objects are relatively simple and straightforward.

* Implement the **IUnknown** methods to handle reference counting and provide pointers to the object's interfaces.
* Implement the methods of the UMDF callback interfaces that are to be exported by the object.

## How to Implement the Class

UMDF callback objects are typically implemented as a C++ class that contains the code to support **IUnknown** plus any UMDF interfaces that the object exposes. The UMDF interfaces are declared in wudfddi.h.

* The class must inherit from every interface that it exposes. However, it can do so indirectly, for example, by inheriting from a class that in turn inherits from one or more interfaces.
* Interfaces are declared as abstract base classes, so the class must implement all the interface methods.
* The class often inherits from a parent class in addition to interfaces. Many of the UMDF samples, for instance, inherit from a parent class, named *CUnknown,* that contains a base implementation of **IUnknown**.
* The class can also contain private data members, public methods that are not part of an interface, and so on. These are for internal use and are not visible to clients.
* Constructors are optional. However, if a class has a constructor, it should contain no code in it that might fail. Put any code that can fail in a public initialization method that can be called after object creation. For an example of such a function, see the *CMyDevice::Initialize* method in the fx2\_driver sample's device.cpp file.

## How to Implement IUnknown

**IUnknown** is the core COM interface; it is exposed by every COM object and is essential to the object's operation. The approach that is used by the UMDF samples is to have an **IUnknown** base class, called *CUnknown*, plus an implementation for each exposed interface that inherits from the base class. For a simple example, see the Skeleton sample's driver.cpp and comsup.cpp files.

### How to Implement AddRef and Release

Reference counting is arguably the key task of **IUnknown**. Normally, a single reference count is maintained for the object as a whole. The following are some recommendations for handling **AddRef** and **Release**:

* Have the interface-specific implementations pass their calls to the base implementation and let it handle incrementing or decrementing the reference count for the object.
* Use **InterlockedIncrement** and **InterlockedDecrement** to modify the reference count. This eliminates the possibility of a race condition.
* After the **Release** method decrements the reference count, check to see whether the count has gone to zero. If so, there are no outstanding interface pointers and you can use **delete** to destroy the object.
* Both **AddRef** and **Release** return the current reference count, which is sometimes useful for debugging purposes.

### How to Implement QueryInterface

**QueryInterface** is the fundamental mechanism by which a COM object provides pointers to its interfaces. It responds to a client's request by returning the specified interface pointer. The following are some recommendations for **QueryInterface**:

* **QueryInterface** must check the incoming IID to see if the request is for a supported interface. **IsEqualIID** is a utility function—declared in guiddef.h,—that simplifies comparing IIDs.
* If the object supports the requested interface, **QueryInterface** calls **AddRef** to increment the object's reference count and returns the requested interface pointer. To return the pointer, **QueryInterface** casts a *this* pointer to the requested interface type. This cast is required because of the way in which C++ handles multiple inheritance.
* When a client queries for **IUnknown**, an object must always return the same **IUnknown** pointer regardless of which interface **QueryInterface** is called from.

## How to Implement UMDF Interfaces

The basic process of implementing UMDF callback interfaces is similar to **IUnknown**. Most of the implementation details are governed by the requirements of the individual methods and are beyond the scope of this paper. For more information on implementing particular callback interfaces, see the UMDF samples or the white paper titled *Sample Drivers for the User-Mode Driver Framework*.

# Resources

The following links provide further information about UMDF and COM.

*Inside COM*, Rogerson, D., Microsoft Press, 1997.

A detailed introduction to basic COM programming.

*Inside OLE*, Brockschmidt, C., Microsoft Press, 1995

A detailed discussion of how to use COM for object linking and embedding (OLE).

*Inside ATL*, Shepherd, G. and B. King, Microsoft Press, 1999

How to implement COM objects by using ATL.

Platform SDK COM Documentation

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/componentobjectmodelanchor.asp>

ATL Documentation

<http://msdn.microsoft.com/library/en-us/vccore/html/_atl_ATL_Article_Overview.asp>

Introduction to the WDF User-Mode Driver Framework

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

Architecture of the User-Mode Driver Framework

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

Sample Drivers for the User-Mode Driver Framework

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