Contents

Getting started with Windows drivers

7/24/2019 • 2 minutes to read • Edit Online

Start here to learn fundamental concepts about drivers.

You should already be familiar with the C programming language, and you should understand the ideas of function pointers, callback functions, and event handlers. If you are going to write a driver based on User-Mode Driver Framework 1.x, you should be familiar with C++ and COM.

In this section

- What is a driver?
- Do you need to write a driver?
- Choosing a driver model
- Write your first driver
- Windows compatible hardware development boards
- Sharks Cove hardware development board
- Provision a computer for driver deployment and testing (WDK 10)
- Concepts for all driver developers

Related topics

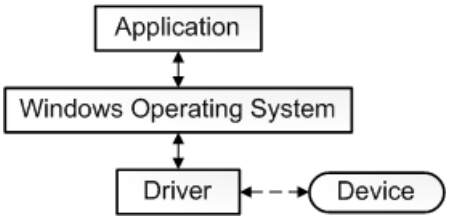
Windows Driver Kit (WDK)

Driver Security Guidance

What is a driver?

12/5/2018 • 4 minutes to read • Edit Online

It is challenging to give a single precise definition for the term *driver*. In the most fundamental sense, a driver is a software component that lets the operating system and a device communicate with each other. For example, suppose an application needs to read some data from a device. The application calls a function implemented by the operating system, and the operating system calls a function implemented by the driver. The driver, which was written by the same company that designed and manufactured the device, knows how to communicate with the device hardware to get the data. After the driver gets the data from the device, it returns the data to the operating system, which returns it to the application.

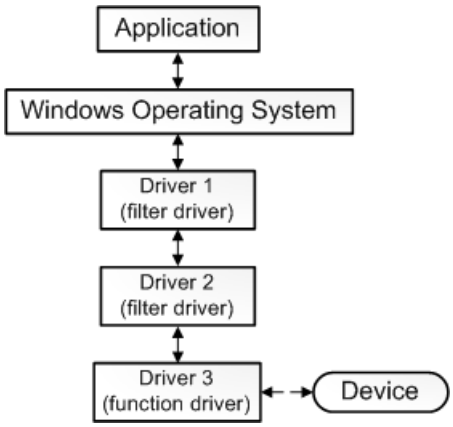


Expanding the definition

Our explanation so far is oversimplified in several ways:

- Not all drivers have to be written by the company that designed the device. In many cases, a device is designed according to a published hardware standard. This means that the driver can be written by Microsoft, and the device designer does not have to provide a driver.

- Not all drivers communicate directly with a device. For a given I/O request (like reading data from a device), there are often several drivers, layered in a stack, that participate in the request. The conventional way to visualize the stack is with the first participant at the top and the last participant at the bottom, as shown in this diagram. Some of the drivers in the stack might participate by transforming the request from one format to another. These drivers do not communicate directly with the device; they just manipulate the request and pass the request along to drivers that are lower in the stack.

  The one driver in the stack that communicates directly with the device is called the *function driver*; the drivers that perform auxiliary processing are called *filter drivers*.

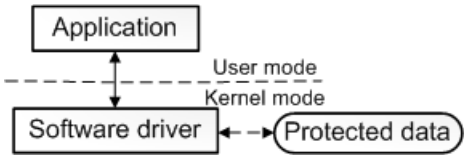
- Some filter drivers observe and record information about I/O requests but do not actively participate in them. For example, certain filter drivers act as verifiers to make sure the other drivers in the stack are handling the I/O request correctly.

We could expand our definition of *driver* by saying that a driver is any software component that observes or participates in the communication between the operating system and a device.

Software drivers

Our expanded definition is reasonably accurate but is still incomplete because some drivers are not associated with any hardware device at all. For example, suppose you need to write a tool that has access to core operating system data structures, which can be accessed only by code running in kernel mode. You can do that by splitting the tool into two components. The first component runs in user mode and presents the user interface. The second component runs in kernel mode and has access to the core operating system data. The component that runs in user mode is called an application, and the component that runs in kernel mode is called a *software driver*. A software driver is not associated with a hardware device. For more information about processor modes, see User Mode and Kernel Mode.

This diagram illustrates a user-mode application communicating with a kernel-mode software driver.

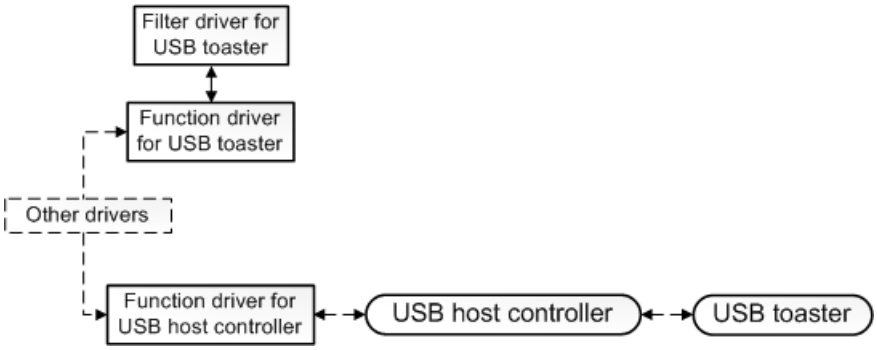


Additional notes

Software drivers always run in kernel mode. The main reason for writing a software driver is to gain access to protected data that is available only in kernel mode. But device drivers do not always need access to kernel-mode data and resources. So some device drivers run in user mode.

There is a category of driver we have not mentioned yet, the *bus driver*. To understand bus drivers, you need to understand device nodes and the device tree. For information about device trees, device nodes, and bus drivers, see Device Nodes and Device Stacks.

Our explanation so far over simplifies the definition of *function driver*. We said that the function driver for a device is the one driver in the stack that communicates directly with the device. This is true for a device that connects directly to the Peripheral Component Interconnect (PCI) bus. The function driver for a PCI device obtains addresses that are mapped to port and memory resources on the device. The function driver communicates directly with the device by writing to those addresses. However in many cases, a device does not connect directly to the PCI bus. Instead the device connects to a host bus adapter that is connected to the PCI bus. For example, a USB toaster connects to a host bus adapter (called a USB host controller), which is connected to the PCI bus. The USB toaster has a function driver, and the USB host controller also has a function driver. The function driver for the toaster communicates indirectly with the toaster by sending a request to the function driver for the USB host controller. The function driver for the USB host controller then communicates directly with the USB host controller hardware, which communicates with the toaster.



Do you need to write a driver?

10/23/2019 • 6 minutes to read • Edit Online

Microsoft Windows contains built-in drivers for many device types. If there is a built-in driver for your device type, you won't need to write your own driver. Your device can use the built-in driver.

Built-in drivers for USB devices

If your device belongs to a device class that is defined by the USB Device Working Group (DWG), there may already be an existing Windows USB class driver for it. For more information, see Drivers for the Supported USB Device Classes.

Built-in drivers for other devices

Currently, Microsoft provides built-in drivers for the following other types of devices:

DEVICE TECHNOLOGY AND DRIVER	BUILT-IN DRIVER	WINDOWS SUPPORT	DESCRIPTION
ACPI: ACPI driver	Acpi.sys	Windows XP and later	Microsoft provides support for basic ACPI device functionality by means of the Acpi.sys driver and ACPI BIOS. To enhance the functionality of an ACPI device, the vendor can supply a WDM function driver. For more information about Windows ACPI support, see Supporting ACPI Devices in the ACPI Design Guide.

DEVICE TECHNOLOGY AND DRIVER	BUILT-IN DRIVER	WINDOWS SUPPORT	DESCRIPTION
Audio: Microsoft Audio Class driver	PortCls.sys	Windows XP and later	Microsoft provides support for basic audio rendering and audio capture via its Port Class driver (PortCls). It is the responsibility of the hardware vendor of an audio device, to provide an adapter driver to work with PortCls. The adapter driver includes initialization code, driver-management code (including the DriverEntry function) and a collection of audio miniport drivers. For more information, see Introduction to Port Class.
Buses: Native SD bus driver, native SD storage class driver, and storage miniport driver	sdbus.sys, sffdisk.sys, sffp_sd.sys	Windows Vista and later	Microsoft provides support for SD card readers as follows: The operating system provides support for SD host controllers that connect directly to the PCI bus. When the system enumerates an SD host controller, it loads a native SD bus driver (sdbus.sys). If a user inserts an SD memory card, Windows loads a native SD storage class driver (sffdisk.sys) and storage miniport driver (sffp_sd.sys) on top of the bus driver. If a user inserts an SD card with a different kind of function, such as GPS or wireless LAN, Windows loads a vendor-supplied driver for the device.

DEVICE TECHNOLOGY AND DRIVER	BUILT-IN DRIVER	WINDOWS SUPPORT	DESCRIPTION
HID: HID I2C driver	HIDI2C.sys	Windows 8 and later	Microsoft provides support for HID over I2C devices on SoC systems that support Simple Peripheral Bus (SPB) and general-purpose I/O (GPIO). It does so by means of the HIDI2C.sys driver. For more information, see HID over I2C.
HID: Legacy game port driver	HidGame.sys, Gameenum.sys	Windows Vista Windows Server 2003 Windows XP	In Windows Vista and earlier, Microsoft provided support for legacy (non-USB, non-Bluetooth, non-I2C) game ports by means of the HidGame.sys and Gameenum.sys drivers. For more information, see HID Transports Supported in Windows.
HID: Legacy keyboard class driver	Kbdclass.sys	Windows XP and later	Microsoft provides support for legacy (non-USB, non-Bluetooth, non-I2C) keyboards by means of the Kbdclass.sys driver. For more information, see Keyboard and mouse HID client drivers. To enhance the functionality of a legacy keyboard, the vendor can supply a keyboard filter driver. For more information, see the Kbfiltr sample.
HID: Legacy mouse class driver	Mouclass.sys	Windows XP and later	Microsoft provides support for legacy (non-USB, non-Bluetooth, non-I2C) mice by means of the Mouclass.sys driver. Keyboard and mouse HID client drivers. To enhance the functionality of a legacy mouse, the vendor can supply a mouse filter driver. For more information, see the Moufiltr sample.

DEVICE TECHNOLOGY AND DRIVER	BUILT-IN DRIVER	WINDOWS SUPPORT	DESCRIPTION
HID: PS/2 (i8042prt) driver	I8042prt.sys	Windows XP and later	Microsoft provides support for legacy PS/2 keyboards and mice by means of the I8042.sys driver. To enhance the functionality of a PS/2 mouse or keyboard, the vendor can supply a keyboard or mouse filter driver. For more information, see the Kbfiltr sample and Moufiltr sample.
Imaging: Web Services for Devices (WSD) scan class driver	WSDScan.sys	Windows Vista and later	Microsoft provides support for web services scanners (that is, scanners that are meant to be used over the web) by means of the WSD scan driver (wsdscan.sys). However, a web services scanner device that supports WSD Distributed Scan Management must implement two web services protocols. For more information, see WIA with Web Services for Devices.
Print: Microsoft Plotter Driver	Msplot	Windows XP and later	Microsoft provides support for plotters that support the Hewlett-Packard Graphics Language by means of the Microsoft Plotter Driver (Msplot). To enhance the functionality of a plotter, you can create a minidriver, which consists of one or more plotter characterization data (PCD) files. For more information, see Plotter Driver Minidrivers.

DEVICE TECHNOLOGY AND DRIVER	BUILT-IN DRIVER	WINDOWS SUPPORT	DESCRIPTION
Print: Microsoft PostScript Printer Driver	Pscript	Windows XP and later	Microsoft provides support for PostScript printers by means of the PostScript Printer Driver (Pscript). To enhance the functionality of a PostScript printer, you can create a minidriver, which consists of one or more PostScript Printer Description (PPD) files and font (NTF) files. For more information, see Pscript Minidrivers.
Print: Microsoft Universal Printer Driver	Unidrv	Windows XP and later	Microsoft provides support for non-PostScript printers by means of the Universal Printer Driver (Unidrv). To enhance the functionality of a non-PostScript printer, you can create a minidriver, which consists of one or more generic printer description (GPD) files. For more information, see Microsoft Universal Printer Driver.
Print: Microsoft v4 Printer Driver		Windows 8 and later	Beginning with Windows 8, Microsoft provides a single in-box class driver that supports PostScript and non-PostScript printers as well as plotters. This driver supersedes the Microsoft Plotter Driver, Microsoft Universal Printer Driver, and Microsoft PostScript Printer Driver. Used on its own, without modification, this printer driver provides basic printing support. For more information, see V4 Printer Driver.

DEVICE TECHNOLOGY AND DRIVER	BUILT-IN DRIVER	WINDOWS SUPPORT	DESCRIPTION
Print: Microsoft XPS Printer Driver	XPSDrv	Windows Vista and later	Microsoft provides support for printing the XPS document format with the XPS Printer Driver (XPSDrv). This driver extends Microsoft's GDI-based, version 3 printer driver architecture to support consuming XML Paper Specification (XPS) documents. With an XPSDrv printer driver, the XPS Document format is used as a spool file format and as a document file format. Used on its own, without modification, the XPSDrv printer driver provides support for basic XPS printing. For more information, see XPSDrv Printer Drivers.
Sensors: Sensor HID class driver	SensorsHIDClassDriver.dll	Windows 8 and later	Microsoft provides support for motion, activity and other types of sensors by means of a HID class driver. Because Windows 8 includes this HID class driver, along with corresponding HID I2C and HID USB miniport drivers, you do not need to implement your own driver. You only need to report the usages described in this white paper, in the firmware for your sensor. Windows will use your firmware and its own HID driver to enable and initialize your sensor, and then furnish the relevant Windows APIs with access to your sensor.

DEVICE TECHNOLOGY AND DRIVER	BUILT-IN DRIVER	WINDOWS SUPPORT	DESCRIPTION
Touch: Windows pointer device driver		Windows 8 and later	Microsoft provides support for pen and touch devices by means of an HID class driver. Because Windows 8 includes this HID class driver and corresponding HID I2C and HID USB miniport drivers, you do not need to implement your own driver. You only need to report the usages described in this white paper in the firmware for your pointer device. Windows will use your firmware and its own HID driver to enable touch and pointer capabilities for your device and furnish the Windows touch and pointer APIs with access to your device.
WPD: Media Transfer Protocol class driver	WpdMtpDr.dll, WpdMtp.dll,WpdMtpUs.dll, WpdConns.dll, and WpdUsb.sys	Windows Vista and later	Microsoft provides support for portable devices that require connectivity with Windows, such as music players, digital cameras, cellular phones, and health-monitoring devices, by means of the Media Transfer Protocol class driver. A vendor that uses this class driver must implement the MTP class protocol on the device. (For digital still cameras, your MTP implementation should be backward compatible with PTP.) For more information, see Guidance for the Hardware Vendor.

Choosing a driver model

4/27/2020 • 4 minutes to read • Edit Online

Microsoft Windows provides a variety of driver models that you can use to write drivers. The strategy for choosing the best driver model depends on the type of driver you are planning to write. Here are the options:

- Device function driver
- Device filter driver
- Software driver
- File system filter driver
- File system driver

For a discussion about the differences between the various types of drivers, see What is a driver? and Device nodes and device stacks. The following sections explain how to choose a model for each type of driver.

Choosing a driver model for a device function driver

As you design a hardware device, one of the first things to consider is whether you need to write a function driver. Ask the following questions:

Can you avoid writing a driver entirely? If you must write a function driver, what is the best driver model to use? To answer these questions, determine where your device fits in the list of technologies described in Device and driver technologies. See the documentation for that particular technology to determine whether you need to write a function driver and to learn about which driver models are available for your device.

Some of the individual technologies have minidriver models. In a minidriver model, the device driver consists of two parts: one that handles general tasks, and one that handles device-specific tasks. Typically, Microsoft writes the general portion and the device manufacturer writes the device-specific portion. The device specific portions have a variety of names, most of which share the prefix *mini*. Here are some of the names used in minidriver models:

- Display miniport driver
- Audio miniport driver
- Battery miniclass driver
- Bluetooth protocol driver
- HID minidriver
- WIA minidriver
- NDIS miniport driver
- Storage miniport driver
- Streaming minidriver

For an overview of minidriver models, see Minidrivers and driver pairs.

Not every technology listed in Device and driver technologies has a dedicated minidriver model. The documentation for a particular technology might advise you to use the Kernel-Mode Driver Framework (KMDF); the documentation for another technology might advise you to use the User-Mode Driver Framework (UMDF). The key point is that you should start by studying the documentation for your specific device technology. If your device technology has a minidriver model, you must use the minidriver model. Otherwise follow the advice in the your technology-specific documentation about whether to use the UMDF, KMDF, or the Windows Driver Model (WDM).

Choosing a driver model for a device filter driver

Frequently several drivers participate in a single I/O request (like reading data from a device). The drivers are layered in a stack, and the conventional way to visualize the stack is with the first driver at the top and the last driver at the bottom. The stack has one function driver and can also have filter drivers. For a discussion about function drivers and filter drivers, see What is a driver? and Device nodes and device stacks.

If you are preparing to write a filter driver for a device, determine where your device fits in the list of technologies described in Device and driver technologies. Check to see whether the documentation for your particular device technology has any guidance on choosing a filter driver model. If the documentation for your device technology does not offer this guidance, then first consider using UMDF as your driver model. If your filter driver needs access to data structures that are not available through UMDF, consider using KMDF as your driver model. In the extremely rare case that your driver needs access to data structures not available through KMDF, use WDM as your driver model.

Choosing a driver model for a software driver

A driver that is not associated with a device is called a *software driver*. For a discussion about software drivers, see the What is a driver? topic. Software drivers are useful because they can run in kernel mode, which gives them access to protected operating system data. For information about processor modes, see User mode and kernel mode.

For a software driver, your two options are KMDF and the legacy Windows NT driver model. With both KMDF and the legacy Windows NT model, you can write your driver without being concerned about Plug and Play (PnP) and power management. You can concentrate instead on your driver's primary tasks. With KMDF, you do not have to be concerned with PnP and power because the framework handles PnP and power for you. With the legacy Windows NT model, you do not have to be concerned about PnP and power because kernel-mode services operate in an environment that is completely independent from PnP and power management.

Our recommendation is that you use KMDF, especially if you are already familiar with it. If you want your driver to be completely independent from PnP and power management, use the legacy Windows NT model. If you need to write a software driver that is aware of power transitions or PnP events, you cannot use the legacy Windows NT model; you must use KMDF.

Note In the very rare case that you need to write a software driver that is aware of PnP or power events, and your driver needs access to data that is not available through KMDF, you must use WDM.

Choosing a driver model for a file system driver

For help with choosing a model for a file system filter driver, see File system driver samples. Note that file system drivers can be quite complex and may require knowledge of advanced concepts for driver development.

Choosing a driver model for a file system filter driver

For help with choosing a model for a file system filter driver, see File system minifilter drivers and File system filter drivers.

Choosing a driver model for a file system minifilter driver

For help choosing a model for a file system minifilter driver, see File System Minifilter Drivers.

Related topics

Kernel-Mode Driver Framework

User-Mode Driver Framework

Write your first driver

6/25/2019 • 2 minutes to read • Edit Online

If you're writing your first driver, use these exercises to get started. Each exercise is independent of the others, so you can do them in any order.

In this section

TOPIC	DESCRIPTION
Write a Universal Windows driver (UMDF 2) based on a template	This topic describes how to write a Universal Windows driver using User-Mode Driver Framework (UMDF) 2. You'll start with a Microsoft Visual Studio template and then deploy and install your driver on a separate computer.
Write a universal Hello World driver (KMDF)	This topic describes how to write a Universal Windows driver using Kernel-Mode Driver Framework (KMDF). You'll start with a Visual Studio template and then deploy and install your driver on a separate computer.
Write a Universal Windows driver (KMDF) based on a template	This topic describes how to write a Universal Windows driver using KMDF. You'll start with a Visual Studio template and then deploy and install your driver on a separate computer.

UMDF and KMDF are part of the Windows Driver Frameworks (WDF).

Write a Universal Windows driver (UMDF 2) based on a template

7/8/2020 • 4 minutes to read • Edit Online

This topic describes how to write a Universal Windows driver using User-Mode Driver Framework (UMDF) 2. You'll start with a Microsoft Visual Studio template and then deploy and install your driver on a separate computer.

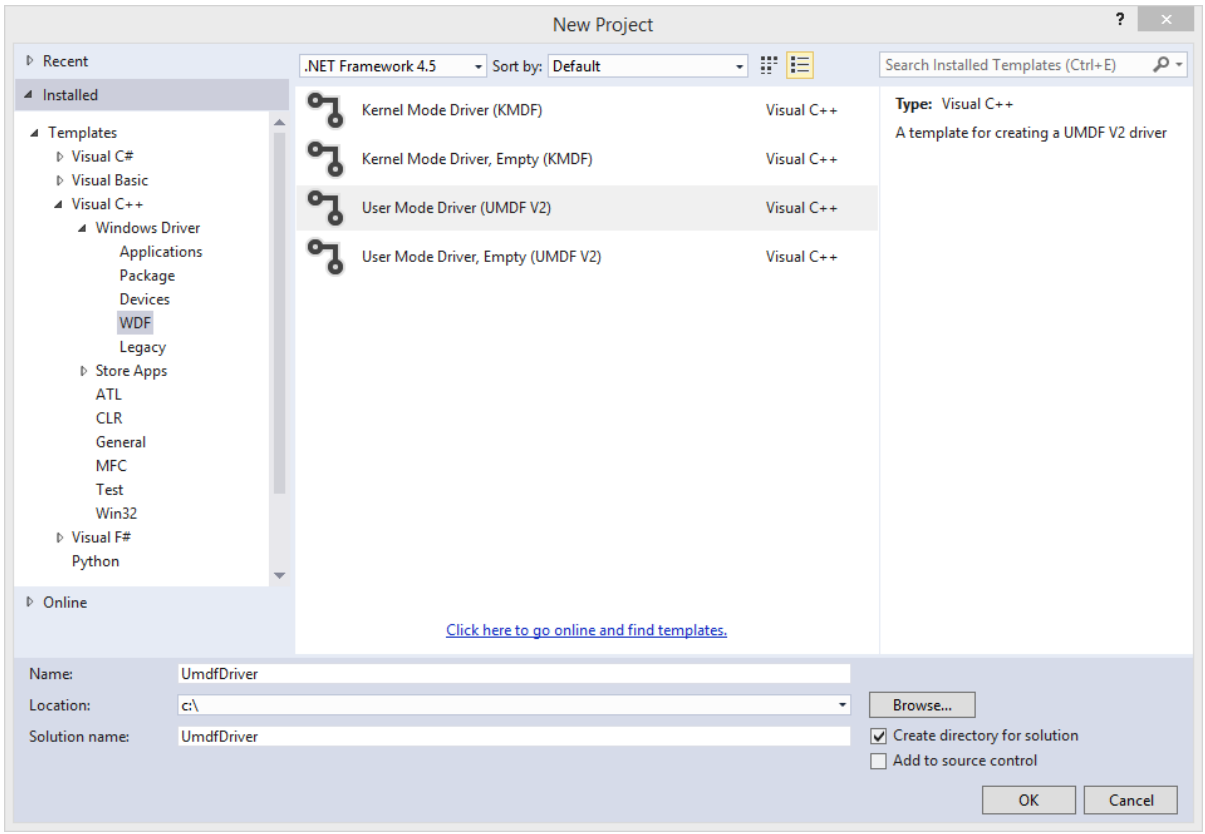
To get started, be sure you have the most recent version of Microsoft Visual Studio and the Windows Driver Kit (WDK). For download links, see Download the Windows Driver Kit (WDK).

Debugging Tools for Windows is included when you install the WDK.

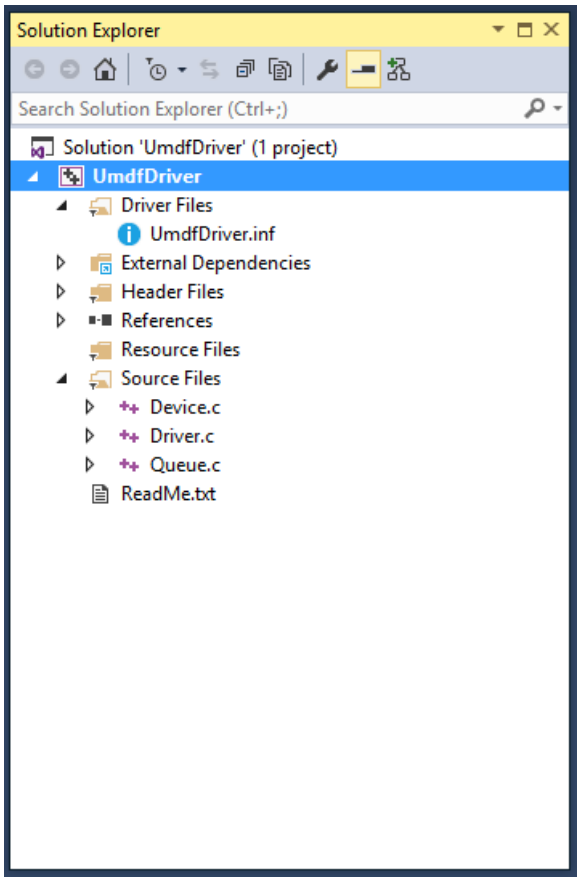
Create and build a driver

> **NOTE**
> When you create a new KMDF or UMDF driver, you must select a driver name that has 32 characters or less. This length limit is defined in wdfglobals.h.

1. Open Visual Studio. On the **File** menu, choose **New > Project**.
2. In the New Project dialog box, in the left pane, go to **Visual C++ > Windows Drivers > WDF**. Select **User Mode Driver (UMDF V2)**.
3. In the **Name** field, enter "UmdfDriver" as the project name.
4. In the **Location** field, enter the directory where you want to create the new project.
5. Check **Create directory for solution**. Click **OK**.



Visual Studio creates one project and a solution. You can see them in the **Solution Explorer** window. (If the **Solution Explorer** window is not visible, choose **Solution Explorer** from the **View** menu.) The solution has a driver project named UmdfDriver. To see the driver source code, open any of the files under **Source Files**. Driver.c and Device.c are good places to start.



6. In the **Solution Explorer** window, right-click **Solution 'UmdfDriver' (1 project)**, and choose **Configuration Manager**. Choose a configuration and platform for the driver project. For example, choose **Debug** and **x64**.

7. In the **Solution Explorer** window, right-click **UmdfDriver**, and choose **Properties**. Navigate to **Configuration Properties > Driver Settings > General**, and note that **Target Platform** defaults to **Universal.**

8. To build your driver, choose **Build Solution** from the **Build** menu. Microsoft Visual Studio displays build progress in the **Output** window. (If the **Output** window is not visible, choose **Output** from the **View** menu.)

   Verify that the build output includes:

   ```
   >  Driver is a Universal Driver.
   ```

   When you've verified that the solution built successfully, you can close Visual Studio.

9. To see the built driver, in File Explorer, go to your **UmdfDriver** folder, and then to **x64\Debug\UmdfDriver**. The directory includes the following files:

   - UmdfDriver.dll -- the user-mode driver file
   - UmdfDriver.inf -- an information file that Windows uses when you install the driver

Deploy and install the Universal Windows driver

Typically when you test and debug a driver, the debugger and driver run on separate computers. The computer that runs the debugger is called the *host computer*, and the computer that runs the driver is called the *target computer*. The target computer is also called the *test computer*.

So far, you've used Visual Studio to build a driver on the host computer. Now you need to configure a target computer. Follow the instructions in Provision a computer for driver deployment and testing (WDK 10). Then you'll be ready to deploy, install, load, and debug your driver:

1. On the host computer, open your solution in Visual Studio. You can double-click the solution file, UmdfDriver.sln, in your UmdfDriver folder.

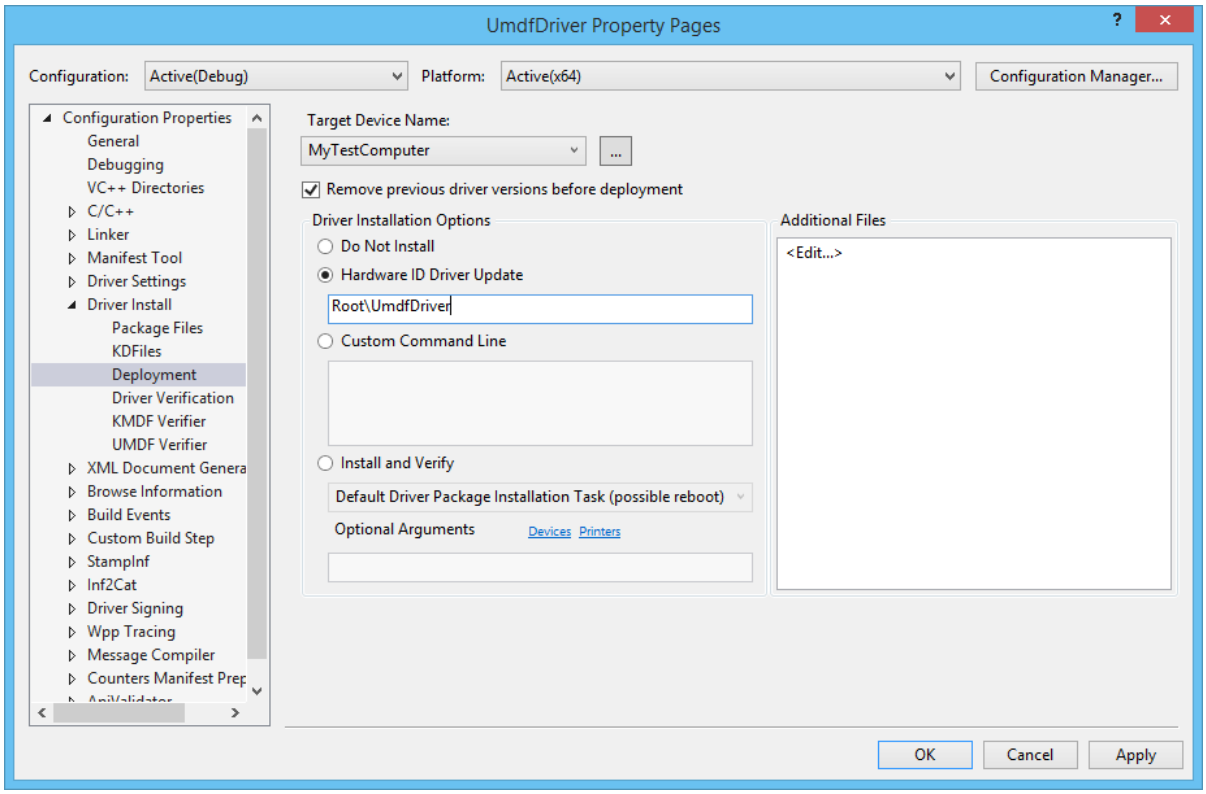
2. In the **Solution Explorer** window, right-click **UmdfDriver**, and choose **Properties**.

3. In the **UmdfDriver Property Pages** window, go to **Configuration Properties > Driver Install > Deployment**, as shown here.

4. Check **Remove previous driver versions before deployment**.

5. For **Target Device Name**, select the name of the computer that you configured for testing and debugging.

6. Select **Hardware ID Driver Update**, and enter the hardware ID for your driver. In this exercise, the hardware ID is Root\UmdfDriver. Click **OK**.



Note In this exercise, the hardware ID does not identify a real piece of hardware. It identifies an imaginary device that will be given a place in the device tree as a child of the root node. For real hardware, do not select **Hardware ID Driver Update**; instead, select **Install and Verify**. You can see the hardware ID in your driver's information (INF) file. In the **Solution Explorer** window, go to **UmdfDriver > Driver Files**, and double-click UmdfDriver.inf. The hardware ID is under [Standard.NT$ARCH$].

```
[Standard.NT$ARCH$]
%DeviceName%=MyDevice_Install,Root\UmdfDriver
```

7. On the **Debug** menu, choose **Start Debugging**, or press **F5** on the keyboard.
8. Wait until your driver has been deployed, installed, and loaded on the target computer. This might take several minutes.

Using the Driver Module Framework (DMF)

The Driver Module Framework (DMF) is an extension to WDF that enables extra functionality for a WDF driver developer. It helps developers write any type of WDF driver better and faster.

DMF as a framework allows creation of WDF objects called DMF Modules. The code for these DMF Modules can be shared between different drivers. In addition, DMF bundles a library of DMF Modules that we have developed for our drivers and feel would provide value to other driver developers.

DMF does not replace WDF. DMF is a second framework that is used with WDF. The developer leveraging DMF still uses WDF and all its primitives to write device drivers.

For more info, see Driver Module Framework (DMF).

Related topics

Developing, Testing, and Deploying Drivers

Debugging Tools for Windows

Write a Hello World Windows Driver (KMDF)

7/8/2020 • 11 minutes to read • Edit Online

This topic describes how to write a very small Universal Windows driver using Kernel-Mode Driver Framework (KMDF) and then deploy and install your driver on a separate computer.

To get started, be sure you have Microsoft Visual Studio, the Windows SDK, and the Windows Driver Kit (WDK) installed.

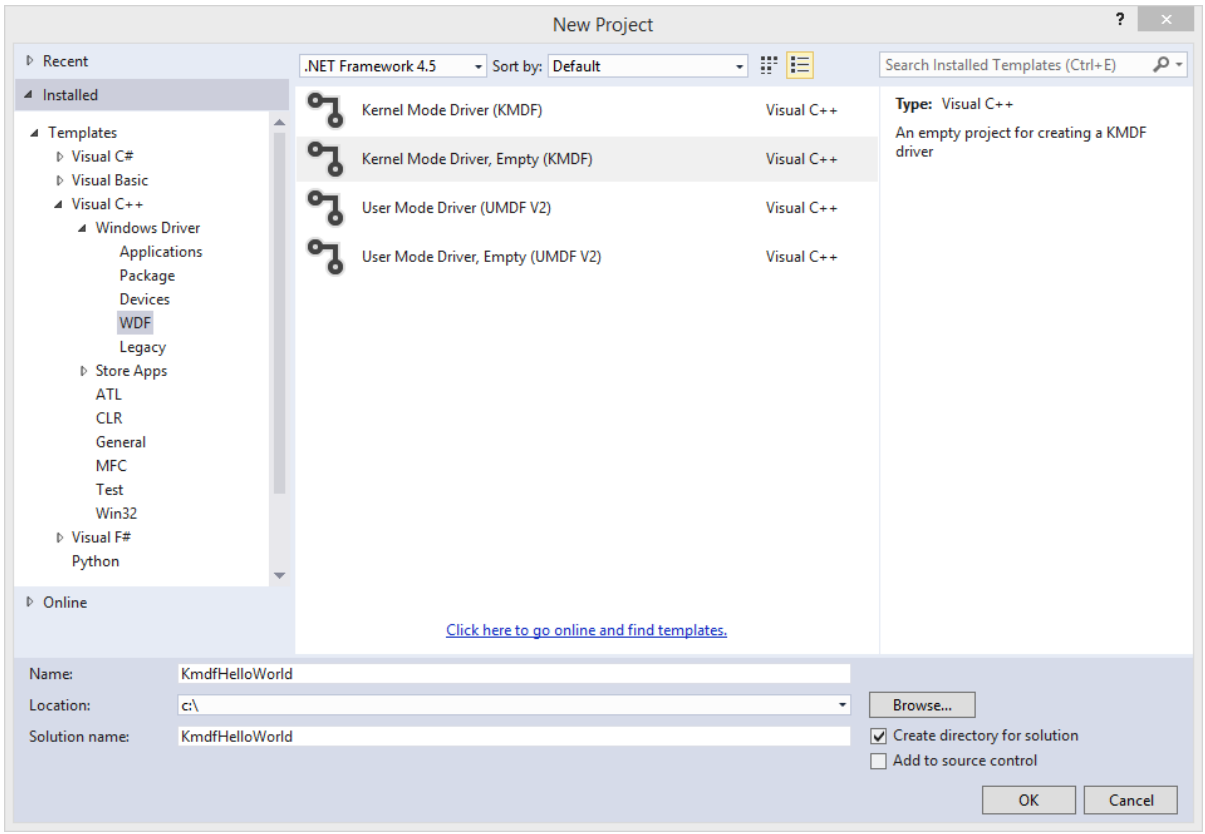
Debugging Tools for Windows is included when you install the WDK.

Create and build a driver

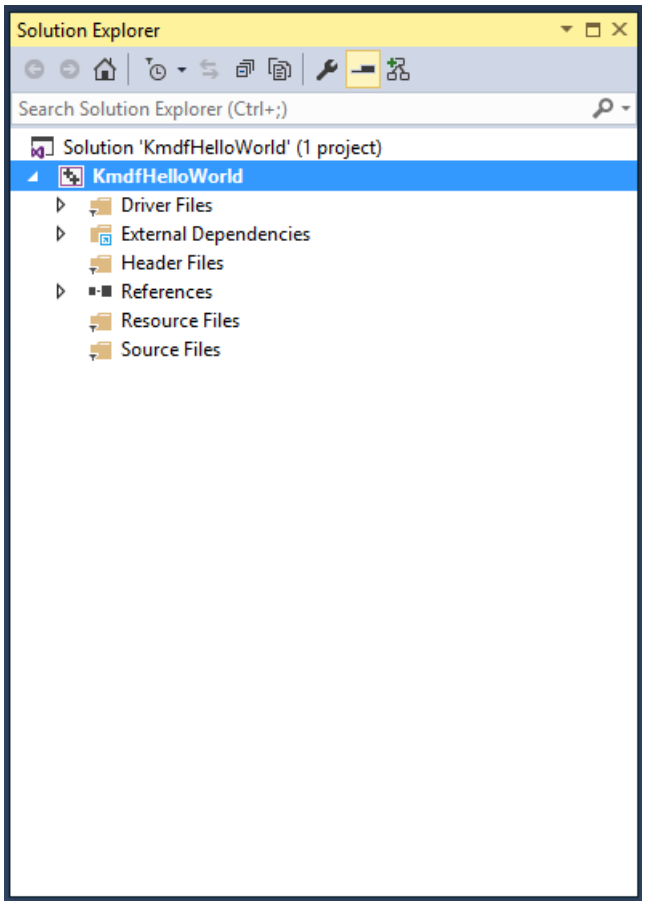
1. Open Microsoft Visual Studio. On the **File** menu, choose **New > Project**.
2. In the New Project dialog box, in the left pane, go to **Visual C++ > Windows Drivers > WDF**.
3. In the middle pane, select **Kernel Mode Driver, Empty (KMDF)**.
4. In the **Name** field, enter "KmdfHelloWorld" for the project name.

> **NOTE**
> When you create a new KMDF or UMDF driver, you must select a driver name that has 32 characters or less. This length limit is defined in wdfglobals.h.

5. In the **Location** field, enter the directory where you want to create the new project.
6. Check **Create directory for solution**. Click **OK**.



Visual Studio creates one project and a solution. You can see them in the **Solution Explorer** window, shown here. (If the Solution Explorer window is not visible, choose **Solution Explorer** from the **View** menu.) The solution has a driver project named KmdfHelloWorld.



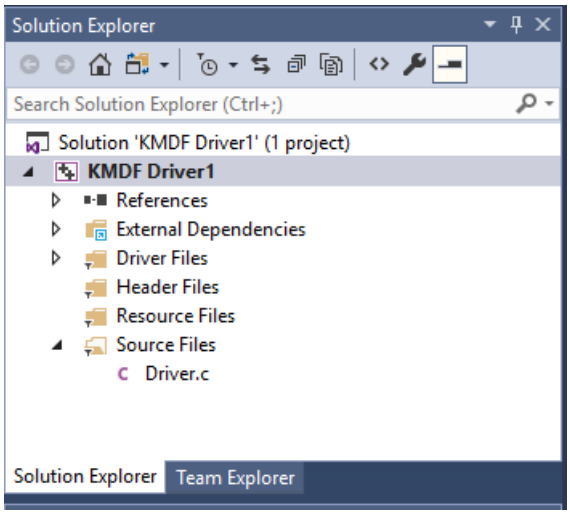
7. In the **Solution Explorer** window, right-click the **KmdfHelloWorld** project and choose **Properties**. Navigate to **Configuration Properties > Driver Settings > General**, and note that **Target Platform** defaults to **Universal.** Click **Apply**, and then click **OK**..

8. In the **Solution Explorer** window, again right-click the **KmdfHelloWorld** project, choose **Add**, and then click **New Item**.

9. In the **Add New Item** dialog box, select **C++ File**. For **Name**, enter "Driver.c".

   > **NOTE**
   > The file name extension is **.c**, not **.cpp**.

   Click **Add**. The Driver.c file is added under Source Files, as shown here.



Write your first driver code

Now that you've created your empty Hello World project and added the Driver.c source file, you'll write the most basic code necessary for the driver to run by implementing two basic event callback functions.

1. In Driver.c, start by including these headers:

   ```
   #include <ntddk.h>
   #include <wdf.h>
   ```

   Ntddk.h contains core Windows kernel definitions for all drivers, while Wdf.h contains definitions for drivers based on the Windows Driver Framework (WDF).

2. Next, provide declarations for the two callbacks you'll use:

   ```
   DRIVER_INITIALIZE DriverEntry;
   EVT_WDF_DRIVER_DEVICE_ADD KmdfHelloWorldEvtDeviceAdd;
   ```

3. Use the following code to write your *DriverEntry*:

```
NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT     DriverObject,
    _In_ PUNICODE_STRING    RegistryPath
)
{
    // NTSTATUS variable to record success or failure
    NTSTATUS status = STATUS_SUCCESS;

    // Allocate the driver configuration object
    WDF_DRIVER_CONFIG config;

    // Print "Hello World" for DriverEntry
    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "KmdfHelloWorld: DriverEntry\n" ));

    // Initialize the driver configuration object to register the
    // entry point for the EvtDeviceAdd callback, KmdfHelloWorldEvtDeviceAdd
    WDF_DRIVER_CONFIG_INIT(&config,
                           KmdfHelloWorldEvtDeviceAdd
                           );

    // Finally, create the driver object
    status = WdfDriverCreate(DriverObject,
                             RegistryPath,
                             WDF_NO_OBJECT_ATTRIBUTES,
                             &config,
                             WDF_NO_HANDLE
                             );
    return status;
}
```

DriverEntry is the entry point for all drivers, like `Main()` is for many user mode applications. The job of *DriverEntry* is to initialize driver-wide structures and resources. In this example, you printed "Hello World" for *DriverEntry*, configured the driver object to register your *EvtDeviceAdd* callback's entry point, then created the driver object and returned.

The driver object acts as the parent object for all other framework objects you might create in your driver, which include device objects, I/O queues, timers, spinlocks, and more. For more information about framework objects, see Introduction to Framework Objects.

> **TIP**
> For *DriverEntry*, we strongly recommend keeping the name as "DriverEntry" to help with code analysis and debugging.

4. Next, use the following code to write your *KmdfHelloWorldEvtDeviceAdd*:

```
NTSTATUS
KmdfHelloWorldEvtDeviceAdd(
    _In_    WDFDRIVER       Driver,
    _Inout_ PWDFDEVICE_INIT DeviceInit
)
{
    // We're not using the driver object,
    // so we need to mark it as unreferenced
    UNREFERENCED_PARAMETER(Driver);

    NTSTATUS status;

    // Allocate the device object
    WDFDEVICE hDevice;

    // Print "Hello World"
    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "KmdfHelloWorld: KmdfHelloWorldEvtDeviceAdd\n"
));

    // Create the device object
    status = WdfDeviceCreate(&DeviceInit,
                             WDF_NO_OBJECT_ATTRIBUTES,
                             &hDevice
                             );
    return status;
}
```

EvtDeviceAdd is invoked by the system when it detects that your device has arrived. Its job is to initialize structures and resources for that device. In this example, you simply printed out a "Hello World" message for *EvtDeviceAdd*, created the device object, and returned. In other drivers you write, you might create I/O queues for your hardware, set up a *device context* storage space for device-specific information, or perform other tasks needed to prepare your device.

> **TIP**
>
> For the device add callback, notice how you named it with your driver's name as a prefix (*KmdfHelloWorld*EvtDeviceAdd). Generally, we recommend naming your driver's functions in this way to differentiate them from other drivers' functions. *DriverEntry* is the only one you should name exactly that.

5. Your complete Driver.c now looks like this:

```
#include <ntddk.h>
#include <wdf.h>
DRIVER_INITIALIZE DriverEntry;
EVT_WDF_DRIVER_DEVICE_ADD KmdfHelloWorldEvtDeviceAdd;

NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT     DriverObject,
    _In_ PUNICODE_STRING    RegistryPath
)
{
    // NTSTATUS variable to record success or failure
    NTSTATUS status = STATUS_SUCCESS;

    // Allocate the driver configuration object
    WDF_DRIVER_CONFIG config;

    // Print "Hello World" for DriverEntry
    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "KmdfHelloWorld: DriverEntry\n" ));

    // Initialize the driver configuration object to register the
    // entry point for the EvtDeviceAdd callback, KmdfHelloWorldEvtDeviceAdd
    WDF_DRIVER_CONFIG_INIT(&config,
                           KmdfHelloWorldEvtDeviceAdd
                           );

    // Finally, create the driver object
    status = WdfDriverCreate(DriverObject,
                             RegistryPath,
                             WDF_NO_OBJECT_ATTRIBUTES,
                             &config,
                             WDF_NO_HANDLE
                             );
    return status;
}

NTSTATUS
KmdfHelloWorldEvtDeviceAdd(
    _In_    WDFDRIVER       Driver,
    _Inout_ PWDFDEVICE_INIT DeviceInit
)
{
    // We're not using the driver object,
    // so we need to mark it as unreferenced
    UNREFERENCED_PARAMETER(Driver);

    NTSTATUS status;

    // Allocate the device object
    WDFDEVICE hDevice;

    // Print "Hello World"
    KdPrintEx(( DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL, "KmdfHelloWorld: KmdfHelloWorldEvtDeviceAdd\n"
));

    // Create the device object
    status = WdfDeviceCreate(&DeviceInit,
                             WDF_NO_OBJECT_ATTRIBUTES,
                             &hDevice
                             );
    return status;
}
```

6. Save Driver.c.

This example illustrates a fundamental concept of drivers: they are a "collection of callbacks" that, once initialized,

sit and wait for the system to call them when it needs something. This could be a new device arrival event, an I/O request from a user mode application, a system power shutdown event, a request from another driver, or a surprise removal event when a user unplugs the device unexpectedly. Fortunately, to say "Hello World," you only needed to worry about driver and device creation.

Next, you'll build your driver.

Build the driver

1. In the **Solution Explorer** window, right-click **Solution 'KmdfHelloWorld' (1 project)** and choose **Configuration Manager**. Choose a configuration and platform for the driver project. For this exercise, we choose **Debug** and **x64**.
2. In the **Solution Explorer** window, right-click **KmdfHelloWorld** and choose **Properties**. In **Wpp Tracing > All Options**, set **Run Wpp tracing** to **No**. Click **Apply** and then **OK**.
3. To build your driver, choose **Build Solution** from the **Build** menu. Visual Studio shows the build progress in the **Output** window. (If the **Output** window is not visible, choose **Output** from the **View** menu.) When you have verified that the solution built successfully, you can close Visual Studio.
4. To see the built driver, in File Explorer, go to your **KmdfHelloWorld** folder, and then to **C:\KmdfHelloWorld\x64\Debug\KmdfHelloWorld**. The folder includes:
   - KmdfHelloWorld.sys -- the kernel-mode driver file
   - KmdfHelloWorld.inf -- an information file that Windows uses when you install the driver
   - KmdfHelloWorld.cat -- a catalog file that the installer uses to verify the driver's test signature

> **TIP**
> If you see `DriverVer set to a date in the future` when building your driver, change your driver project settings so that Inf2Cat sets `/uselocaltime`. To do so, use **Configuration Properties->Inf2Cat->General->Use Local Time**. Now both Stampinf and Inf2Cat use local time.

Deploy the driver

Typically when you test and debug a driver, the debugger and the driver run on separate computers. The computer that runs the debugger is called the *host computer*, and the computer that runs the driver is called the *target computer*. The target computer is also called the *test computer*.

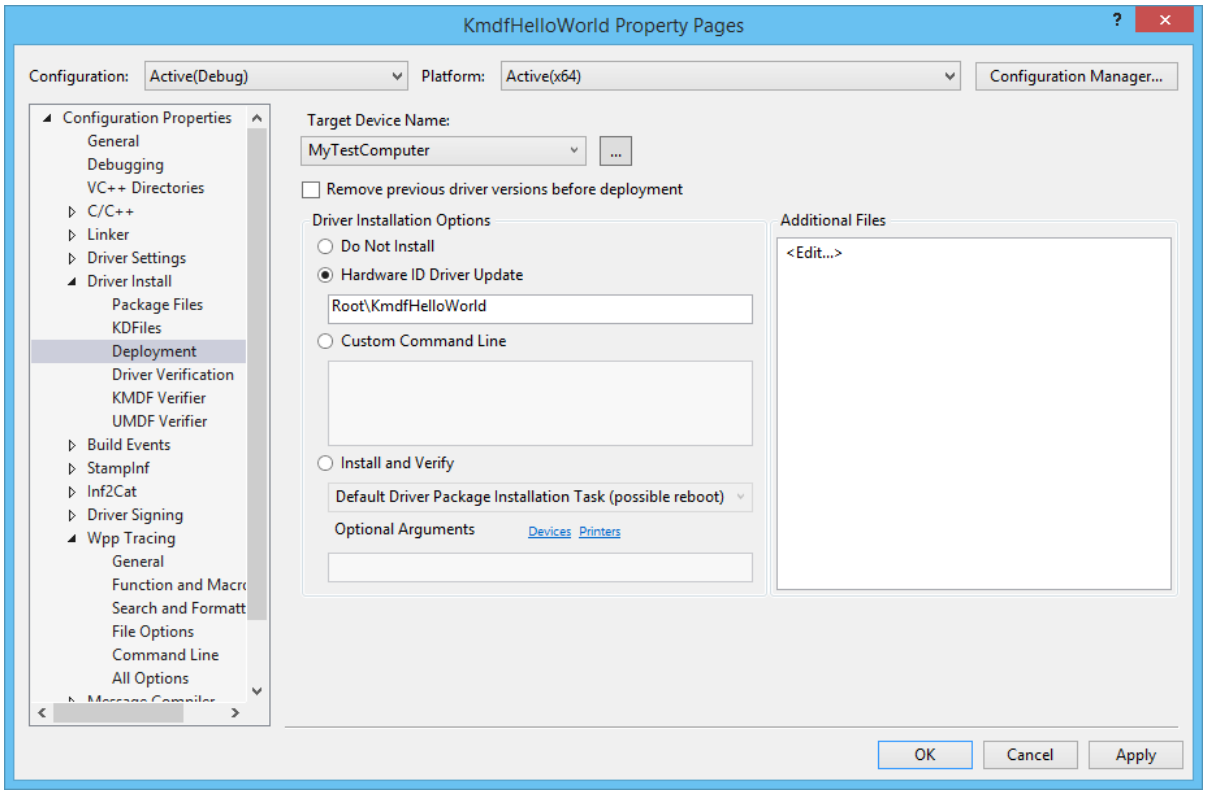
So far you've used Visual Studio to build a driver on the host computer. Now you need to configure a target computer.

1. Follow the instructions in Provision a computer for driver deployment and testing (WDK 10).

   > **TIP**
   > When you follow the steps to provision the target computer automatically using a network cable, take note of the port and key. You'll use them later in the debugging step. In this example, we'll use **50000** as the port and **1.2.3.4** as the key.
   >
   > In real driver debugging scenarios, we recommend using a KDNET-generated key. For more information about how to use KDNET to generate a random key, see the Debug Drivers - Step by Step Lab (Sysvad Kernel Mode) topic.

2. On the host computer, open your solution in Visual Studio. You can double-click the solution file, KmdfHelloWorld.sln, in your KmdfHelloWorld folder.

3. In the **Solution Explorer** window, right-click the **KmdfHelloWorld** project, and choose **Properties**.
4. In the **KmdfHelloWorld Property Pages** window, go to **Configuration Properties > Driver Install > Deployment**, as shown here.
5. Check **Remove previous driver versions before deployment**.
6. For **Target Device Name**, select the name of the computer that you configured for testing and debugging. In this exercise, we use a computer named MyTestComputer.
7. Select **Hardware ID Driver Update**, and enter the hardware ID for your driver. For this exercise, the hardware ID is Root\KmdfHelloWorld. Click **OK**.



> **NOTE**
>
> In this exercise, the hardware ID does not identify a real piece of hardware. It identifies an imaginary device that will be given a place in the device tree as a child of the root node. For real hardware, do not select **Hardware ID Driver Update**; instead, select **Install and Verify**. You'll see the hardware ID in your driver's information (INF) file. In the **Solution Explorer** window, go to **KmdfHelloWorld > Driver Files**, and double-click KmdfHelloWorld.inf. The hardware ID is located under [Standard.NT$ARCH$].

```
[Standard.NT$ARCH$]
%KmdfHelloWorld.DeviceDesc%=KmdfHelloWorld_Device, Root\KmdfHelloWorld
```

8. On the **Build** menu, choose **Deploy Solution**. Visual Studio automatically copies the files required to install and run the driver to the target computer. This may take a minute or two.

   When you deploy a driver, the driver files are copied to the %Systemdrive%\drivertest\drivers folder on the test computer. If something goes wrong during deployment, you can check to see if the files are copied to the test computer. Verify that the .inf, .cat, test cert, and .sys files, and any other necessary files, are present in the %systemdrive%\drivertest\drivers folder.

   For more information about deploying drivers, see Deploying a Driver to a Test Computer.

Install the driver

With your Hello World driver deployed to the target computer, now you'll install the driver. When you previously provisioned the target computer with Visual Studio using the *automatic* option, Visual Studio set up the target computer to run test signed drivers as part of the provisioning process. Now you just need to install the driver using the DevCon tool.

1. On the host computer, navigate to the Tools folder in your WDK installation and locate the DevCon tool. For example, look in the following folder:

   C:\Program Files (x86)\Windows Kits\10\Tools\x64\devcon.exe

   Copy the DevCon tool to your remote computer.

2. On the target computer, install the driver by navigating to the folder containing the driver files, then running the DevCon tool.

   a. Here's the general syntax for the devcon tool that you will use to install the driver:

      devcon install <INF file> <hardware ID>

      The INF file required for installing this driver is KmdfHelloWorld.inf. The INF file contains the hardware ID for installing the driver binary, *KmdfHelloWorld.sys*. Recall that the hardware ID, located in the INF file, is **Root\KmdfHelloWorld**.

   b. Open a Command Prompt window as Administrator. Navigate to your folder containing the built driver .sys file and enter this command:

      devcon install kmdfhelloworld.inf root\kmdfhelloworld

      If you get an error message about *devcon* not being recognized, try adding the path to the *devcon* tool. For example, if you copied it to a folder on the target computer called *C:\Tools*, then try using the following command:

      c:\tools\devcon install kmdfhelloworld.inf root\kmdfhelloworld

      A dialog box will appear indicating that the test driver is an unsigned driver. Click **Install this driver anyway** to proceed.

Debug the driver

Now that you have installed your KmdfHelloWorld driver on the target computer, you'll attach a debugger remotely from the host computer.

1. On the host computer, open a Command Prompt window as Administrator. Change to the WinDbg.exe directory. We will use the x64version of WinDbg.exe from the Windows Driver Kit (WDK) that was installed as part of the Windows kit installation. Here is the default path to WinDbg.exe:

C:\Program Files (x86)\Windows Kits\10\Debuggers\x64

2. Launch WinDbg to connect to a kernel debug session on the target computer by using the following command. The value for the port and key should be the same as what you used to provision the target computer. We'll use **50000** for the port and **1.2.3.4** for the key, the values we used during the deploy step. The *k* flag indicates that this is a kernel debug session.

   WinDbg -k net:port=50000,key=1.2.3.4

3. On the **Debug** menu, choose **Break**. The debugger on the host computer will break into the target computer. In the **Debugger Command** window, you can see the kernel debugging command prompt: **kd>**.

4. At this point, you can experiment with the debugger by entering commands at the **kd>** prompt. For example, you could try these commands:

   - lm
   - .sympath
   - .reload
   - x KmdfHelloWorld!*

5. To let the target computer run again, choose **Go** from the **Debug** menu or press "g," then press "enter."

6. To stop the debugging session, choose **Detach Debuggee** from the **Debug** menu.

   > **IMPORTANT**
   > Make sure you use the "go" command to let the target computer run again before exiting the debugger, or the target computer will remain unresponsive to your mouse and keyboard input because it is still talking to the debugger.

For a detailed step-by-step walkthrough of the driver debugging process, see Debug Universal Drivers - Step by Step Lab (Echo Kernel-Mode).

For more information about remote debugging, see Remote Debugging Using WinDbg.

Related topics

Developing, Testing, and Deploying Drivers

Debugging Tools for Windows

Debug Universal Drivers - Step by Step Lab (Echo Kernel-Mode)

Write your first driver

Write a Universal Windows driver (KMDF) based on a template

7/8/2020 • 7 minutes to read • Edit Online

This topic describes how to write a Universal Windows driver using Kernel-Mode Driver Framework (KMDF). You'll start with a Microsoft Visual Studio template and then deploy and install your driver on a separate computer.

To get started, be sure you have the latest versions of [Microsoft Visual Studio]https://visualstudio.microsoft.com/vs/) and the Windows Driver Kit (WDK) installed.

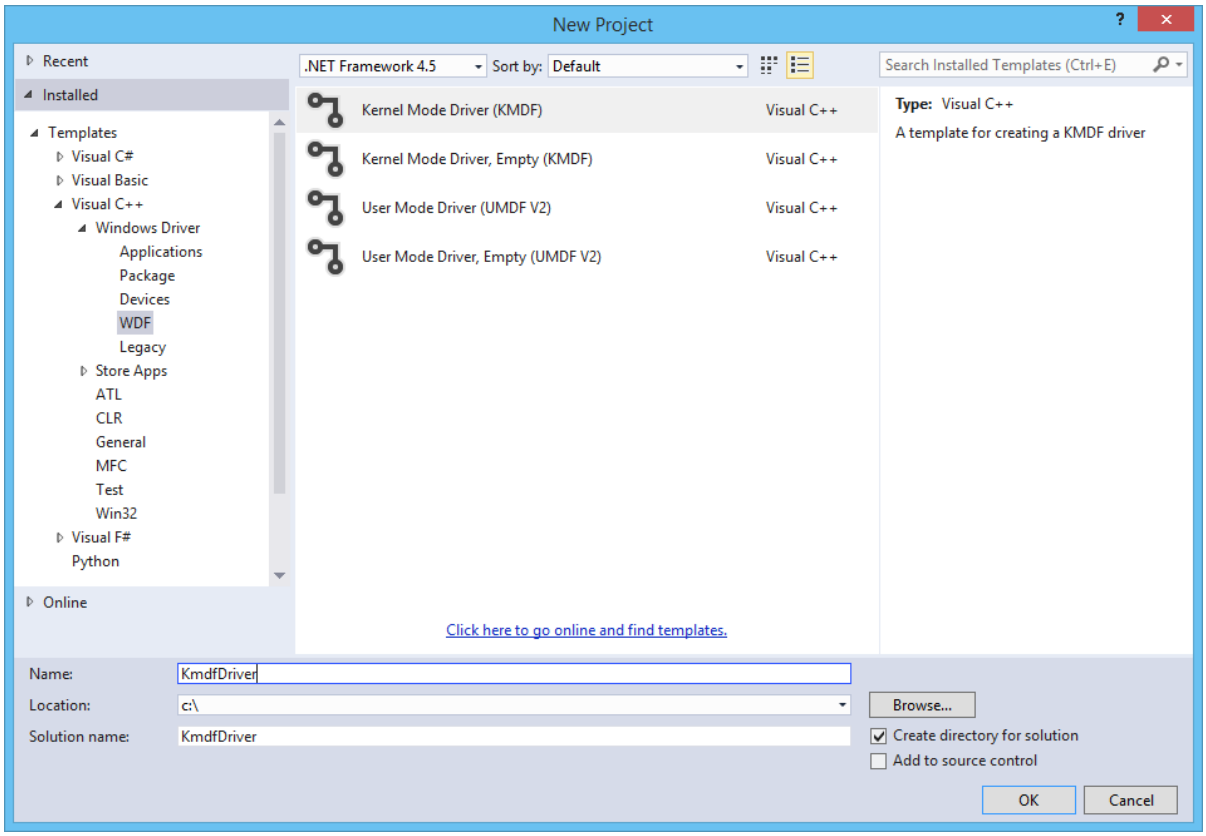
Debugging Tools for Windows is included when you install the WDK.

Create and build a driver package

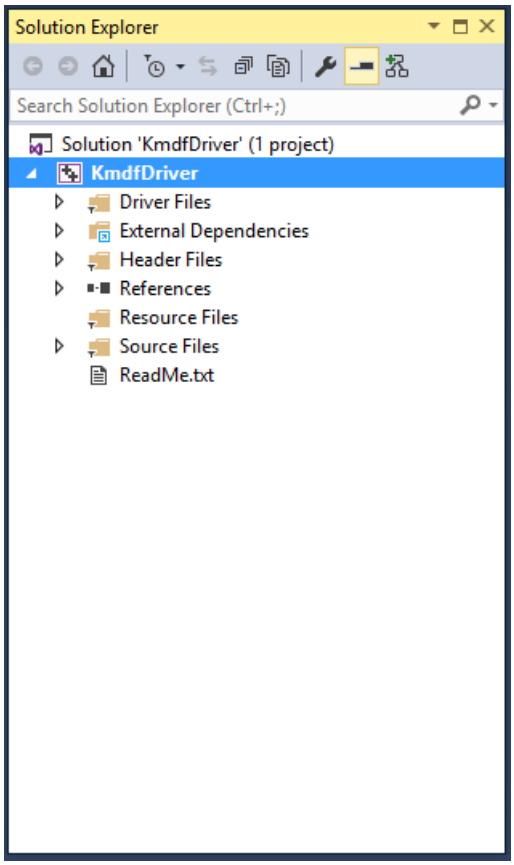
1. Open Microsoft Visual Studio. On the **File** menu, choose **New > Project**. The **New Project** dialog box opens, as shown here.
2. In the **New Project** dialog box, select **WDF**.
3. In the middle pane, select **Kernel Mode Driver (KMDF)**.
4. In the **Name** field, enter "KmdfDriver" as the project name.

   > **NOTE**
   > When you create a new KMDF or UMDF driver, you must select a driver name that has 32 characters or less. This length limit is defined in wdfglobals.h.

5. In the **Location** field, enter the directory where you want to create the new project.
6. Check **Create directory for solution**. Click **OK**.



Visual Studio creates one project and a solution. You can them in the **Solution Explorer** window, as shown here. (If the **Solution Explorer** window is not visible, choose **Solution Explorer** from the **View** menu.) The solution has a driver project named KmdfDriver. To see the driver source code, open any of the files under **Source Files**. Driver.c and Device.c are good places to start.



7. In the **Solution Explorer** window, right-click **Solution 'KmdfDriver' (1 project)**, and choose **Configuration Manager**. Choose a configuration and platform for both the driver project and the package project. In this exercise, we choose Debug and x64.

8. To build your driver and create a driver package, choose **Build Solution** from the **Build** menu. Visual Studio shows the build progress in the **Output** window. (If the **Output** window is not visible, choose **Output** from the **View** menu.)

   When you've verified that the solution built successfully, you can close Visual Studio.

9. To see the built driver, in File Explorer, go to your **KmdfDriver** folder, and then to **x64\Debug\KmdfDriver**. The folder includes:

   - KmdfDriver.sys -- the kernel-mode driver file
   - KmdfDriver.inf -- an information file that Windows uses when you install the driver

Deploy the driver

Typically when you test and debug a driver, the debugger and driver run on separate computers. The computer that runs the debugger is called the *host computer*, and the computer that runs the driver is called the *target computer*. The target computer is also called the *test computer*. For more information about debugging drivers, see Debugging Tools for Windows.

So far you've used Visual Studio to build a driver on the host computer. Now you need to configure a target computer.

1. Follow the instructions in Provision a computer for driver deployment and testing (WDK 10).

   > **TIP**
   > When you follow the steps to provision the target computer automatically using a network cable, take note of the port and key. You'll use them later in the debugging step. In this example, we'll use **50000** as the port and **1.2.3.4** as the key.
   >
   > In real driver debugging scenarios, we recommend using a KDNET-generated key. For more information about how to use KDNET to generate a random key, see the Debug Drivers - Step by Step Lab (Sysvad Kernel Mode) topic.

2. On the host computer, open your solution in Visual Studio. You can double-click the solution file, KmdfDriver.sln, in your KmdfDriver folder.

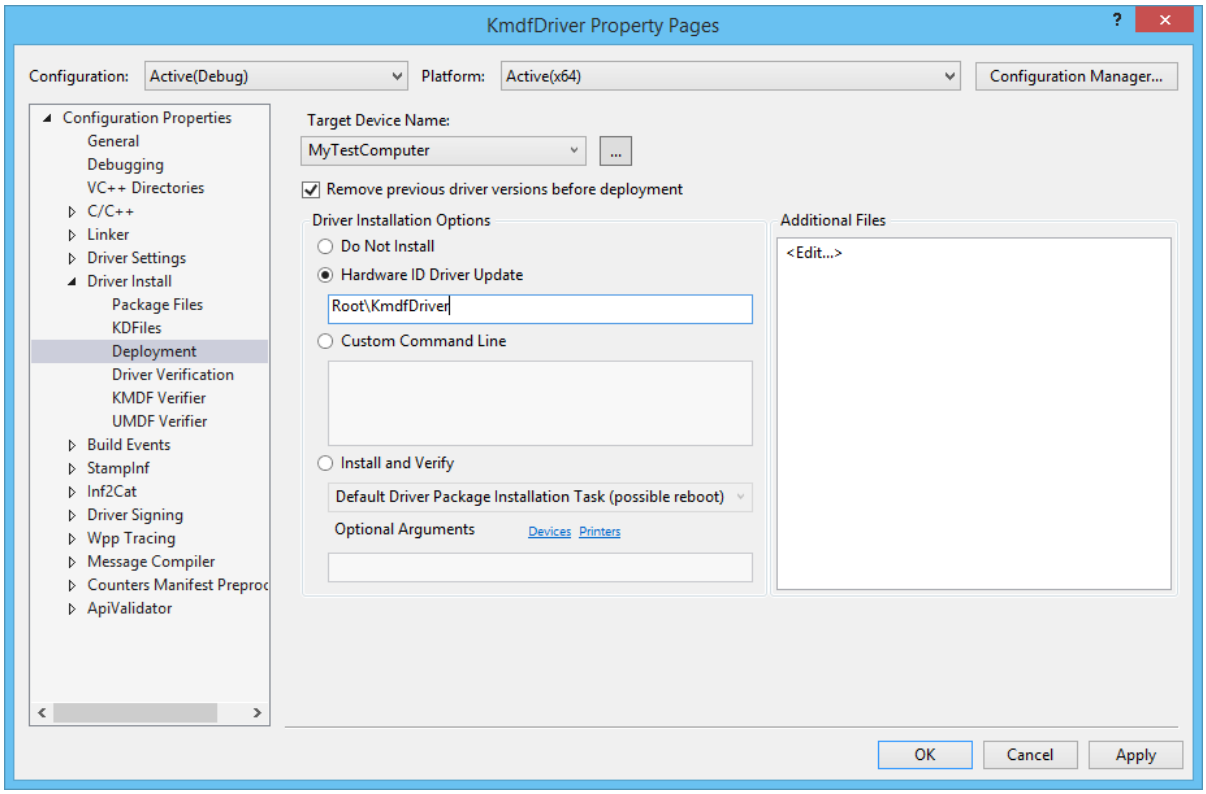
3. In the **Solution Explorer** window, right-click the **KmdfDriver** project, and choose **Properties**.

4. In the **KmdfDriver Package Property Pages** window, in the left pane, go to **Configuration Properties > Driver Install > Deployment**.

5. Check **Remove previous driver versions before deployment**.

6. For **Remote Computer Name**, select the name of the computer that you configured for testing and debugging. In this exercise, we use a computer named MyTestComputer.

7. Select **Hardware ID Driver Update**, and enter the hardware ID for your driver. In this exercise, the hardware ID is Root\KmdfDriver. Click **OK**.



NOTE

In this exercise, the hardware ID does not identify a real piece of hardware. It identifies an imaginary device that will be given a place in the device tree as a child of the root node. For real hardware, do not select **Hardware ID Driver Update**; instead, select **Install and Verify**. You'll see the hardware ID in your driver's information (INF) file. In the **Solution Explorer** window, go to **KmdfDriver > Driver Files** and double-click KmdfDriver.inf. The hardware ID is located under [Standard.NT$ARCH$].

```
[Standard.NT$ARCH$]
%KmdfDriver.DeviceDesc%=KmdfDriver_Device, Root\KmdfDriver
```

8. On the **Build** menu, choose **Deploy Solution**. Visual Studio automatically copies the files required to install and run the driver to the target computer. This may take a minute or two.

   When you deploy a driver, the driver files are copied to the %Systemdrive%\drivertest\drivers folder on the test computer. If something goes wrong during deployment, you can check to see if the files are copied to the test computer. Verify that the .inf, .cat, test cert, and .sys files, and any other necessary files, are present in the %systemdrive%\drivertest\drivers folder.

   For more information about deploying drivers, see Deploying a Driver to a Test Computer.

Install the driver

With your KMDF driver deployed to the target computer, now you'll install the driver. When you previously provisioned the target computer with Visual Studio using the *automatic* option, Visual Studio set up the target computer to run test signed drivers as part of the provisioning process. Now you just need to install the driver using the DevCon tool.

1. On the host computer, navigate to the Tools folder in your WDK installation and locate the DevCon tool. For example, look in the following folder:

   C:\Program Files (x86)\Windows Kits\10\Tools\x64\devcon.exe

Copy the DevCon tool to your remote computer.

2. On the target computer, install the driver by navigating to the folder containing the driver files, then running the DevCon tool.

   a. Here's the general syntax for the devcon tool that you will use to install the driver:

      devcon install <INF file> <hardware ID>

      The INF file required for installing this driver is KmdfDriver.inf. The INF file contains the hardware ID for installing the driver binary, *KmdfDriver.sys*. Recall that the hardware ID, located in the INF file, is **Root\KmdfDriver**.

   b. Open a Command Prompt window as Administrator. Navigate to your driver package folder, then enter this command:

      devcon install kmdfdriver.inf root\kmdfdriver

      If you get an error message about *devcon* not being recognized, try adding the path to the *devcon* tool. For example, if you copied it to a folder on the target computer called *C:\Tools*, then try using the following command:

      c:\tools\devcon install kmdfdriver.inf root\kmdfdriver

      A dialog box will appear indicating that the test driver is an unsigned driver. Click **Install this driver anyway** to proceed.

Debug the driver

Now that you have installed your KMDF driver on the target computer, you'll attach a debugger remotely from the host computer.

1. On the host computer, open a Command Prompt window as Administrator. Change to the WinDbg.exe directory. We will use the x64version of WinDbg.exe from the Windows Driver Kit (WDK) that was installed as part of the Windows kit installation. Here is the default path to WinDbg.exe:

   C:\Program Files (x86)\Windows Kits\10\Debuggers\x64

2. Launch WinDbg to connect to a kernel debug session on the target computer by using the following command. The value for the port and key should be the same as what you used to provision the target computer. We'll use **50000** for the port and **1.2.3.4** for the key, the values we used during the deploy step. The *k* flag indicates that this is a kernel debug session.

   WinDbg -k net:port=50000,key=1.2.3.4

3. On the **Debug** menu, choose **Break**. The debugger on the host computer will break into the target computer. In the **Debugger Command** window, you can see the kernel debugging command prompt: **kd>**.

4. At this point, you can experiment with the debugger by entering commands at the **kd>** prompt. For example, you could try these commands:

   - lm
   - .sympath
   - .reload
   - x KmdfHelloWorld!*

5. To let the target computer run again, choose **Go** from the **Debug** menu or press "g," then press "enter."

6. To stop the debugging session, choose **Detach Debuggee** from the **Debug** menu.

> **IMPORTANT**
> Make sure you use the "go" command to let the target computer run again before exiting the debugger, or the target computer will remain unresponsive to your mouse and keyboard input because it is still talking to the debugger.

For a detailed step-by-step walkthrough of the driver debugging process, see Debug Universal Drivers - Step by Step Lab (Echo Kernel-Mode).

For more information about remote debugging, see Remote Debugging Using WinDbg.

Using the Driver Module Framework (DMF)

The Driver Module Framework (DMF) is an extension to WDF that enables extra functionality for a WDF driver developer. It helps developers write any type of WDF driver better and faster.

DMF as a framework allows creation of WDF objects called DMF Modules. The code for these DMF Modules can be shared between different drivers. In addition, DMF bundles a library of DMF Modules that we have developed for our drivers and feel would provide value to other driver developers.

DMF does not replace WDF. DMF is a second framework that is used with WDF. The developer leveraging DMF still uses WDF and all its primitives to write device drivers.

For more info, see Driver Module Framework (DMF).

Related topics

Developing, Testing, and Deploying Drivers

Debugging Tools for Windows

Debug Universal Drivers - Step by Step Lab (Echo Kernel-Mode)

Write your first driver

Windows compatible hardware development boards

7/11/2019 • 2 minutes to read • Edit Online

Last updated

- December 2016

Applies to

- Windows 10
- Windows 8.1 Update

Leverage the power of the Windows platform and Visual Studio to create innovative experiences and solutions that are brought to life by interfacing hardware components to Windows devices.

Windows compatible hardware development boards offer an affordable, yet a powerful development system targeted towards hardware developer, IHV, OEM or any other developer that loves to incorporate hardware in their projects and needs the power of a full PC. It allows you to easily interface your hardware components to General Purpose I/O (GPIO) pins and low-power buses like I²C and I²S.

Supported Hardware Development Boards

For information on the latest supported development boards, see IoT Device Options.

Learn more

- **Simple Development for Windows Platforms**

  SoC-based hardware development boards enable hardware manufacturers to develop and certify their Windows drivers without time-consuming purchase and licensing processes, and without the need for extensive engagement with Microsoft. Purchasing the boards is quick and easy, and development tips can be found within the Microsoft developer community.

  Watch the video to learn how Microsoft partnered with vendors like Intel to create easy onboarding opportunities for your off-SOC hardware.

- **Blog post**

  Read about the initiative to make Windows and driver development tools available for affordable development boards.

  Windows-compatible hardware development boards available soon

- **Join the community**

  Find out what the community is doing. Share your stories and learn from others.

  Windows Hardware and Driver Developer Community

Sharks Cove hardware development board

10/21/2019 • 12 minutes to read • Edit Online

WARNING

The Sharks Cove hardware development board is no longer supported for Windows IoT Core. For a list of currently supported boards, see SoCs and custom boards.

Sharks Cove is a hardware development board that you can use to develop hardware and drivers for Windows.

The Intel Sharks Cove board supports driver development for devices that use a variety of interfaces, including GPIO, I2C, I2S, UART, SDIO, and USB. You can also use the Sharks Cove board to develop drivers for cameras and touch screens.

For downloads related to the Sharks Cove board, see Sharks Cove UEFI Firmware.

For detailed specifications, see Sharks Cove Technical Specifications.

Before you start

The instructions given here require that you are running Windows 10, Windows 8.1, or Windows 7. These instructions do not work if you are running Windows 8.

If you are running Windows 7, you need to install PowerShell 4.0 and theWindows Assessment and Deployment Kit (ADK) for Windows 8.1 Update. Then on the **Start** menu, go to **All Programs > Windows Kits > Windows ADK > Deployment and Imaging Tools Environment**. Open this Command Prompt window as Administrator. Use this Command Prompt window when you enter the commands given in these instructions.

Step 1: Get the board and related hardware

You will need this hardware:

- Sharks Cove board with included power cord and adapter.
- USB hub
- USB keyboard
- USB mouse
- USB network adapter
- Monitor and HDMI cable (and possibly adapters)

You can get a Sharks Cove board at Mouser Electronics.

Step 2: Download kits and tools

A driver development environment has two computers: the *host computer* and the *target computer*. The target computer is also called the *test computer*. You develop and build your driver in Microsoft Visual Studio on the host computer. The debugger runs on the host computer and is available in the Visual Studio user interface. When you test and debug a driver, the driver runs on the target computer. In this case, the Sharks Cove board is the target computer.

To develop hardware and drivers for the Sharks Cove board, you need these kits and tools on the host computer:

- Visual Studio
- Windows Driver Kit (WDK), WDK Test Pack, and Debugging Tools for Windows

On the host computer, first download Visual Studio, then download the WDK, and then download the WDK Test Pack. You do not need to download Debugging Tools for Windows separately, because it is included in the WDK.

Documentation

The online documentation for the WDK starts here.

The online documentation for Debugging Tools for Windows starts here.

The documentation for Debugging Tools for Windows is also available as a CHM file in the installation directory. Example: C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x64\debugger.chm.

Step 3: Install Windows on the Sharks Cove board

You can install one of these versions of Windows on your Sharks Cove board:

TERM	DESCRIPTION
Windows Embedded 8.1 Industry Pro Evaluation	This is a 180-day free trial. We will refer to this as the evaluation version.
Windows Embedded 8.1 Industry Pro with Update (x86) - DVD	This requires a subscription. We will refer to this as the full version.

If you intend to install the evaluation version, read this amendment to the license agreement:

Evaluation Software License Terms Amendment for the Hardware Developer Program

If the use of this software is in support of the Hardware Developer Program the following terms shall apply:

- You agree to the terms of the Microsoft Evaluation Software License Terms for Windows Embedded 8.1 Industry Pro (“Evaluation Software License Terms”) in its entirety except for:
  - Section 1.b. (Demonstration Rights) of the Evaluation Software License Terms is amended in part, as:
    - You may demonstrate or deliver for demonstration use to potential customers, a number reasonably necessary for demonstration purposes, Windows Embedded 8.1 Industry Pro devices developed by you through your use of the software ("Demonstration Device"). You may demonstrate and deliver Demonstration Devices to customers that are not under non-disclosure obligations.
  - All provisions in Section 1.b. that do not directly conflict with the amended section above, shall apply.
- **By using the software, you accept these terms.** If you do not accept and comply with these terms, you may not use the software or its features.

Download Windows Embedded 8.1 Industry (x86) Pro Evaluation or Windows Embedded 8.1 Industry Pro with Update (x86) - DVD. Locate the downloaded file. For example,

9600.17050.WINBLUE_REFRESH...X86FRE_EN-US_DV9.ISO.

Create a folder that will be the root for your Sharks Cove setup files (for example, C:\SharksCoveWindows). We will call this folder *Root*. In *Root*, create these subfolders:

- Setup
- SharksCoveBsp

Double click your ISO file, and copy these files to *Root*\Setup.

- Boot
- Efi
- Sources
- Support
- Autorun.inf
- Bootmgr
- Bootmgr.efi
- Setup.exe

Note If you are running Windows 7, right-click the ISO file, and choose **Burn disk image** . Burn the image to a recordable DVD. Then copy the files from the DVD to *Root*\Setup.

Get the Sharks Cove board support package (BSP) here. Copy all the files in the package to *Root*\SharksCoveBsp.

Get the WDK Development Boards Add-on Kit here. Open the **SourceCode** tab. Click **Download** (not the Downloads tab) to get the kit scripts. Open the Scripts folder, and copy these two items to *Root*.

- Create-DevboardImage.ps1
- DevBoard folder

Note The DevBoard folder contains several scripts and modules (DevboardImage.ps1, Devboard.psm1, enable-telnet.ps1, and others).

Open a Command Prompt window as Administrator, and enter **Powershell**. Navigate to *Root*. To add the BSP to your Windows image, enter one of these commands:

If you are using the evaluation version of Windows, enter this command:

```
.\Create-DevboardImage -SourcePath Setup\sources\install.wim -Index 2 -BspManifest
SharksCoveBsp\SharksCoveBsp.xml
```

If you are using the full version of Windows, enter this command:

```
.\Create-DevboardImage -SourcePath Setup\sources\install.wim -Index 1 -BspManifest
SharksCoveBsp\SharksCoveBsp.xml
```

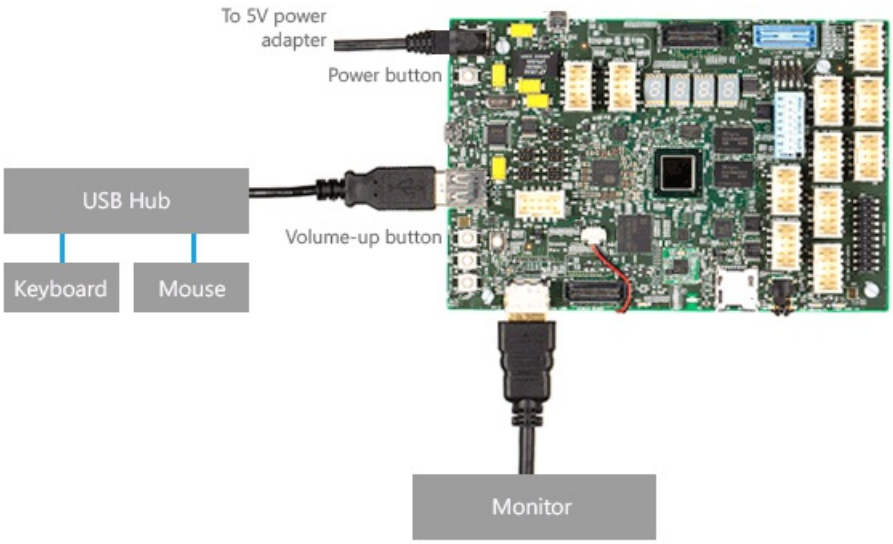
Note You might need to set your execution policy before you run the **Create-DevboardImage** script. For example:

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted
```

Now that you have added the BSP to your Windows image, copy these folders and files from *Root*\Setup to a USB flash drive (FAT32).

- Boot
- Efi
- Sources
- Support
- Autorun.inf
- Bootmgr
- Bootmgr.efi
- Setup.exe

Set up your Sharks Cove hardware as shown here:



Plug the flash drive into the hub that is connected to the Sharks Cove board. Hold the volume-up button as you start or restart the Sharks Cove board. The volume-up button is the top button in the set of three buttons on the left side of the board as shown in the preceding diagram. (If the board is already started, you can turn it off by holding the Power button for several seconds.) When the board starts, you will see the EFI shell on the screen.

Note You might need to navigate to the EFI Shell. Go to **Boot Manager > EFI Internal Shell**.

Note the name of the USB flash drive (for example, **fs1**:).

(Here we will use **fs1**: for the name of the USB flash drive.) At the **Shell>** prompt, enter these commands:

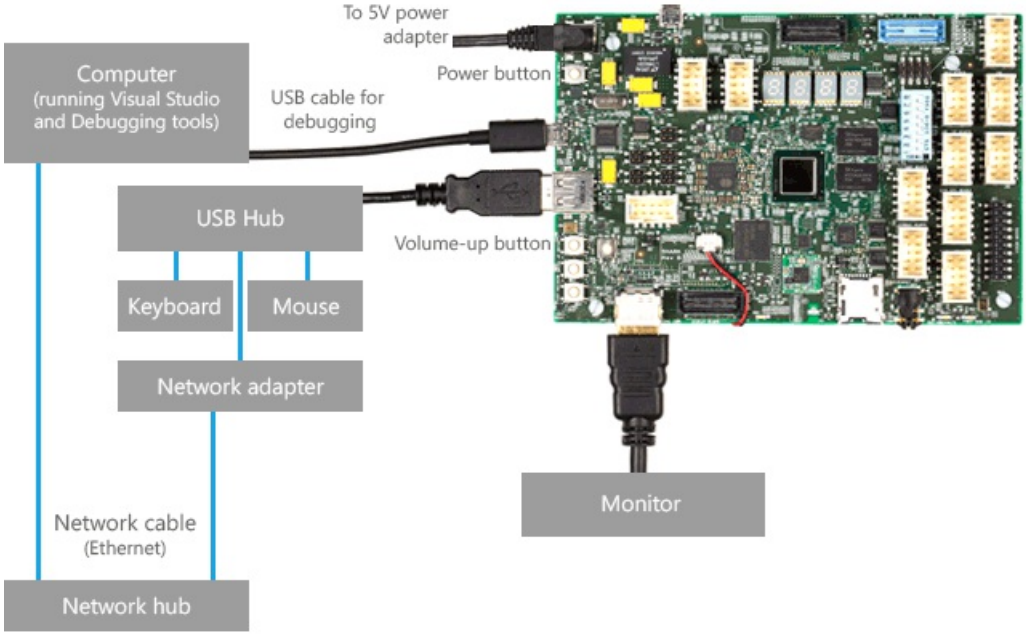
fs1: cd efi\boot dir Verify that bootia32.efi is in the directory. Enter this command:

bootia32.efi Follow the Windows setup instructions on the screen.

Step 4: Provision the Sharks Cove board for driver deployment and testing

Provisioning is the process of configuring a computer for automatic driver deployment, testing, and debugging.

Set up your hardware as shown here.



Provisioning the Sharks Cove board is similar to provisioning any other computer. To provision the Sharks Cove board, follow the instructions in this topic:

- Provision a computer for driver development and testing (WDK 8.1)

and this topic, which is available on line and in debugger.chm.

- Setting up Kernel-Mode Debugging using Serial over USB in Visual Studio

Note Before you provision the Sharks Cove board, you need to disable Secure Boot. Restart the Sharks Cove board. As the board restarts, hold the Volume-up button. Go to **Device Manager > System Setup > Boot**. Set **UEFI Security Boot** to **Disabled**.

Step 5: Write a software driver for the Sharks Cove board

Before you write a device driver for the Sharks Cove board, it is helpful to familiarize yourself with the driver development tools by writing a software driver. The procedure is similar to writing a software driver for any other target computer. To get started, follow the hands-on exercises here:

- Write your first driver

Step 6: Alter the Secondary System Description Table (SSDT)

If you are writing a driver for a driver for a device that connects to a simple peripheral bus (SPB) on the Sharks Cove board, you need to update the Secondary System Description Table (SSDT) in the Sharks Cove firmware. An example of this is writing a driver for an accelerometer that transfers data over an I2C bus and generates interrupts through a general-purpose I/O (GPIO) pin. For more information, see Simple Peripheral Buses.

Here's an example of altering the SSDT. We will add a table entry for the ADXL345 accelerometer.

Note See the SpbAccelerometer driver cookbook for a step-by-step guide to the SpbAccelerometer sample driver and the ADXL345 accelerometer.

1. Copy the x86 version of ASL.exe to the Sharks Cove board. ASL.exe is included in the WDK.

   Example: C:\Program Files (x86)\Windows Kits\8.1\Tools\x86\ACPIVerify\ASL.exe

2. Open a Command Prompt windows as Administrator. Decompile the SSDT by entering this command:

asl /tab=ssdt

This creates the file Ssdt.asl.

3. Open Ssdt.asl (for example, in Notepad).

```
DefinitionBlock("SSDT.AML", "SSDT", 0x01, "Intel_", "ADebTabl", 0x00001000)
{
    Scope()
    {
        Name(DPTR, 0x3bf2d000)
        Name(EPTR, 0x3bf3d000)
        Name(CPTR, 0x3bf2d010)
        Mutex(MMUT, 0x0)
        Method(MDBG, 0x1, Serialized)
        {
            Store(Acquire(MMUT, 0x3e8), Local0)
            If(LEqual(Local0, Zero))
            {
                OperationRegion(ABLK, SystemMemory, CPTR, 0x10)
                Field(ABLK, ByteAcc, NoLock, Preserve)
                {
                    AAAA, 128
                }
                Store(Arg0, AAAA)
                Add(CPTR, 0x10, CPTR)
                If(LNot(LLess(CPTR, EPTR)))
                {
                    Add(DPTR, 0x10, CPTR)
                }
                Release(MMUT)
            }
            Return(Local0)
        }
    }

    // Insert a Scope(_SB_) and a Device entry here.

}
```

4. Insert a Scope(_SB_) entry. Inside your scope entry, insert your own Device entry. For example, here's a scope(_SB_) entry and a Device entry for the ADXL345 accelerometer.

```
Scope(_SB_)
{
    Device(SPBA)
    {
        Name(_HID, "SpbAccelerometer")
        Name(_UID, 1)



    Method(_CRS, 0x0, NotSerialized)
    {
        Name(RBUF, ResourceTemplate()
        {
            I2CSerialBus(0x53, ControllerInitiated, 400000, AddressingMode7Bit, "\\_SB.I2C3", 0,
ResourceConsumer)
            GpioInt(Edge, ActiveHigh, Exclusive, PullDown, 0, "\\_SB.GPO2") {0x17}
        })

        Return(RBUF)
    }


    Method(_DSM, 0x4, NotSerialized)
    {
        If(LEqual(Arg0, Buffer(0x10)
        {
            0x1e, 0x54, 0x81, 0x76, 0x27, 0x88, 0x39, 0x42, 0x8d, 0x9d, 0x36, 0xbe, 0x7f, 0xe1, 0x25,
0x42
        }))
        {
            If(LEqual(Arg2, Zero))
            {
                Return(Buffer(One)
                {
                    0x03
                })
            }

            If(LEqual(Arg2, One))
            {
                Return(Buffer(0x4)
                {
                    0x00, 0x01, 0x02, 0x03
                })
            }
        }
        Else
        {
            Return(Buffer(One)
            {
                0x00
            })
        }
    } // Method(_DSM ...)

} // Device(SPBA)
```

} // Scope(*SB*)

```
In this example, the entries under `ResourceTemplate()` specify that the accelerometer needs two hardware resources: a connection ID to a particular I2C bus controller (I2C3) and a GPIO interrupt. The interrupt uses pin 0x17 on the GPIO controller named GPO2.


5.  After you have added your own Device entry to Ssdt.asl, compile Ssdt.asl by entering this command:

    **asl ssdt.asl**

    This puts the compiled output in a file named Ssdt.aml.

6.  Verify that test signing is turned on for the Sharks Cove board.

    **Note**  Test signing is turned on automatically during provisioning.



On the Sharks Cove board, open a Command Prompt window as Administrator. Enter this command.

**bcdedit /enum {current}**

Verify that you see `testsigning Yes` in the output.

``` syntax
Windows Boot Loader
-------------------
identifier              {current}
...
testsigning             Yes
...
```
```

If you need to turn on test signing manually, here are the steps:

1. Open a Command Prompt window as Administrator, and enter this command.

   bcdedit /set TESTSIGNING ON

2. Restart the Sharks Cove board. As the board restarts, hold the Volume-up button. Go to **Device Manager > System Setup > Boot**. Set **UEFI Security Boot** to **Disabled**.

3. Save your changes and continue booting to Windows.

4. To load your updated SSDT, open a Command Prompt window as Administrator, and enter this command:

   asl /loadtable ssdt.aml

   Restart the Sharks Cove board.

Step 7: Connect your device to the Sharks Cove board

Get the specification for the Sharks Cove headers and pins here.

Use the specification to determine which pins to use for your device. For example, suppose you want to connect the ADXL345 accelerometer to an I2C bus. In the specification, you can see that the J1C1 header has the pins you need. Here are some, but not all, of the pins you would use on the J1C1 header.

PIN	PIN NAME	COMMENTS	ACPI OBJECT
7	GPIO_S5[23]	Accelerometer Interrupt signal	_SB.GPO2 {0x17}

PIN	PIN NAME	COMMENTS	ACPI OBJECT
13	SIO_I2C2_DATA	I2C Data line for I2C controller 2	_SB.I2C3
15	SIO_I2C2_CLK	I2C clock line for I2C controller 2	_SB.I2C3

Notice the relationship to the Device entry in the SSDT.

```
I2CSerialBus(... "\\_SB.I2C3", , )
GpioInt(... "\\_SB.GPO2") {0x17}
```

Step 8: Write, build, and deploy a driver for your device

Writing a device driver for the Sharks Cove board is similar to writing a device driver for any other computer. In Visual Studio, you can start with a driver template or you can start with a driver sample.

When you are ready to test your driver on the Sharks Cove board, follow these steps:

1. On the host computer, in Visual Studio, right click your package project, and choose **Properties**. Go to **Driver Install > Deployment**. Check **Enable Deployment** and **Remove previous driver versions before deployment**. For **Target Computer Name**, select the name of your Sharks Cove board. Select **Install and Verify**.
2. Still in the property pages, go to **Driver Signing > General**. For **Sign Mode**, select **Test Sign**. Click **OK**.
3. In your driver project, open your INF file. Edit the hardware ID so that it matches the hardware ID (_HID) that you created in the SSDT. For example, suppose you put this Device entry in the SSDT.

   ```
   Device(SPBA)
   {
      Name(_HID, "SpbAccelerometer")
      ...
   ```

   Then the hardware ID in you INF file would be ACPI\SpbAccelerometer.

   ```
   [Standard.NT$ARCH$]
   %KMDFDriver1.DeviceDesc%=KMDFDriver1_Device, ACPI\SpbAccelerometer
   ```

4. In Visual Studio, on the **Debug** menu, choose **Start Debugging**.
5. Microsoft Visual Studio first shows progress in the **Output** window. Then it opens the **Debugger Immediate Window** and continues to show progress.

   Wait until your driver has been deployed, installed, and loaded on the Sharks Cove board. This might take a minute or two.
6. If the debugger does not automatically break in, choose **Break All** from the **Debug** menu. The debugger on the host computer will break into the target computer (kernel-mode) or to the correct instance of Wudfhost.exe (UMDF). In the **Debugger Immediate Window**, you'll see a debugger command prompt.
7. To view the loaded modules, enter **lm**. Verify that your driver appears in the list of loaded modules.

Using WinDbg to debug the Sharks Cove board

As an alternative to using Visual Studio to set up kernel-mode debugging, you can do the setup manually. This topic is available on line and in debugger.chm.

- Setting up Kernel-Mode Debugging using Serial over USB Manually

As an alternative to using Visual Studio for debugging, you can use WinDbg.

Regardless of whether you use Visual Studio or WinDbg, these hands-on guides are helpful for learning debugger commands:

- Getting Started with WinDbg (User-Mode)
- Getting Started with WinDbg (Kernel-Mode)

Sample driver code

- SpbAccelerometer Sample Driver (UMDF Version 1)

Understanding simple peripheral buses

To learn how Windows drivers work with simple peripheral buses, see Simple Peripheral Buses.

Related topics

Concepts for all driver developers

Developing, Testing, and Deploying Drivers

Windows Driver Frameworks

Windows Hardware Developer Center

WDK Samples for Windows

Windows Hardware and Driver Developer Community

Technical Support

Provision a computer for driver deployment and testing (WDK 10)

7/8/2020 • 3 minutes to read • Edit Online

Provisioning a target or test computer is the process of configuring a computer for automatic driver deployment, testing, and debugging. To provision a computer, use Microsoft Visual Studio.

A testing and debugging environment has two computers: the *host computer* and the *target computer*. The target computer is also called the *test computer*. You develop and build your driver in Visual Studio on the host computer. The debugger runs on the host computer and is available in the Visual Studio user interface. When you test and debug a driver, the driver runs on the target computer.

The host and target computers must be able to ping each other by name. This might be easier if both computers are joined to the same workgroup or the same network domain. If your computers are in a workgroup, we recommend that you connect the computers with a router rather than a hub or switch.

> **TIP**
> For the latest information on the WDK and known issues, see the WDK support forum.

Prepare the target computer for provisioning

1. On the target computer, install the operating system that you'll use to run and test your driver.
2. Install the WDK. You do not need to install Visual Studio, however, unless you plan on doing driver development on the target computer.
3. If Secure Boot is enabled on an x86 or x64 target computer, disable it. For information about Unified Extensible Firmware Interface (UEFI) and Secure Boot, see UEFI Firmware.

   If the target computer uses an ARM processor, install the Windows Debug Policy. This can be done only by Microsoft or the manufacturer of the target computer. You do not need to disable Secure Boot.
4. On the target computer, run the WDK Test Target Setup MSI that matches the platform of the target computer. You can find the MSI in the Windows Driver Kit (WDK) installation directory under Remote.

   Example: C:\Program Files (x86)\Windows Kits\10\Remote\x64\WDK Test Target Setup x64-x64_en-us.msi
5. If the target computer is running an N or KN version of Windows, install the Media Feature Pack for N and KN versions of Windows:
   - Media Feature Pack for N and KN versions of Windows 8.1
   - Media Feature Pack for N and KN versions of Windows 8
   - Media Feature Pack for N and KN versions of Windows 7
6. If your target computer is running Windows Server, find the DriverTest folder that was just created by WDK Test Target Setup MSI. (Example: c:\DriverTest). Right click the **DriverTest** folder, and choose **Properties**. On the **Security** tab, give **Modify** permission to the **Authenticated Users** group.

Verify that the host and target computers can ping each other. Open a Command Prompt window, and enter **ping** *ComputerName*.

If the host and target computers are joined to a workgroup and are on different subnets, you might have to adjust

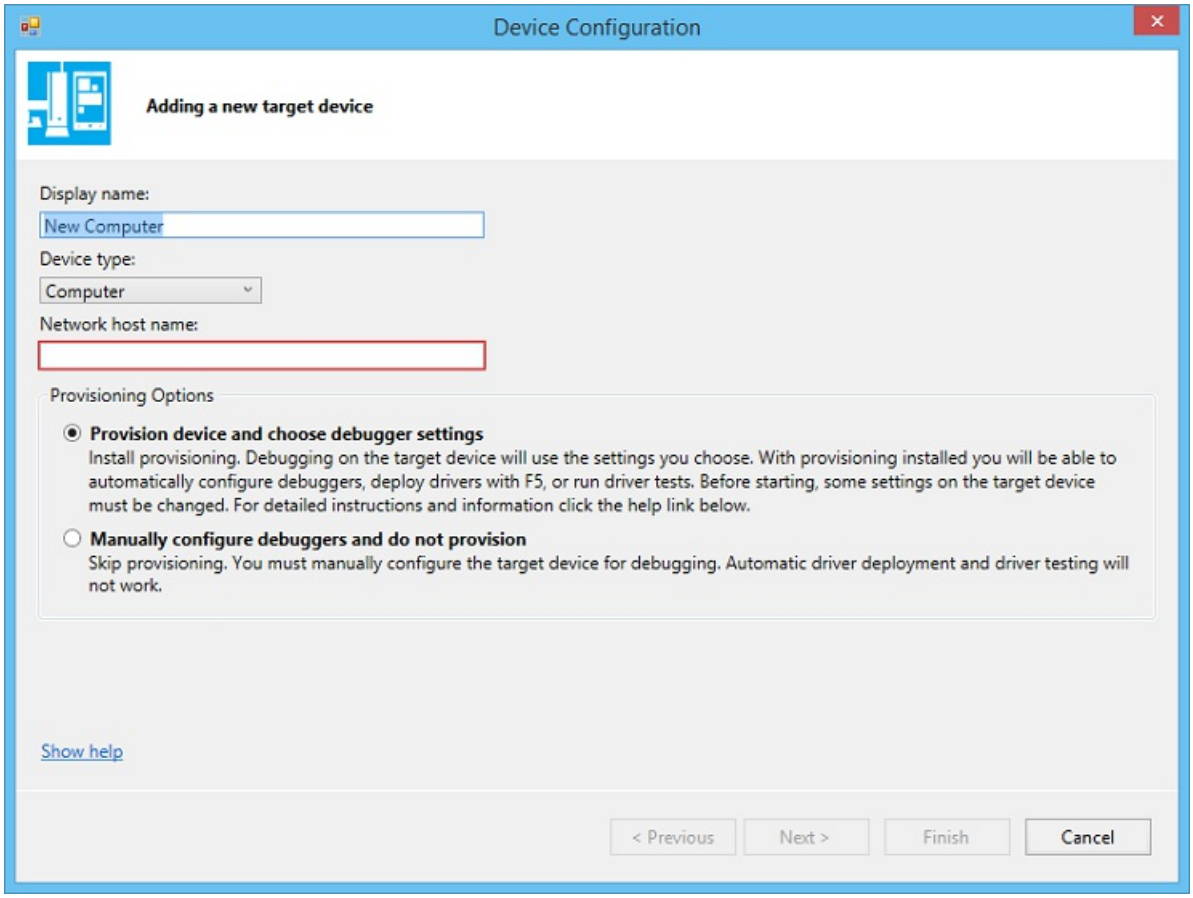
some firewall settings so that the host and target computers can communicate. Follow these steps:

1. On the target computer, in Control Panel, go to **Network and Internet** > **Network Sharing Center**. Note your active network. This will be **Public network**, **Private network**, or **Domain**.
2. On the target computer, in Control Panel, go to **System and Security** > **Windows Firewall** > **Advanced settings** > **Inbound Rules**.
3. In the list of inbound rules, find all Network Discovery rules for your active network. (For example, find all Network Discovery rules that have a **Profile** of **Private**.) Double click each rule and open the **Scope** tab. Under **Remote IP address**, select **Any IP address**.
4. In the list of inbound rules, locate all File and Printer Sharing rules for your active network. For each of those rules, double click the rule, and open the **Scope** tab. Under **Remote IP address**, select **Any IP address**.

Provision the target computer

Now you're ready to provision the target computer from the host computer in Visual Studio.

1. On the host computer, in Visual Studio, click the **Extensions** menu, point to **Driver**, point to **Test**, and click **Configure Devices**.
2. In the **Configure Devices** dialog, click **Add new device**.
3. For **Network host name**, enter the name or local IP address of your target computer. Select **Provision device and choose debugger settings**.



4. Click **Next**.
5. Select a type of debugging connection, and enter the required parameters.

   For more information about setting up debugging over various types of connections, see Setting Up Kernel-Mode Debugging Manually in the CHM or online documentation for Debugging Tools for Windows.
6. The provisioning process takes several minutes and might automatically reboot the target computer once

or twice. When provisioning is complete, click **Finish**.

> **TIP**
>
> Provisioning virtual machines through the WDK's automatic provisioning process is not supported. However, you can test drivers on a VM by setting up the target VM manually as described in the step by step echo lab.

See Also

Deploying a Driver to a Test Computer

Concepts for all driver developers

12/5/2018 • 2 minutes to read • Edit Online

In this section:

User mode and kernel mode

12/5/2018 • 2 minutes to read • Edit Online

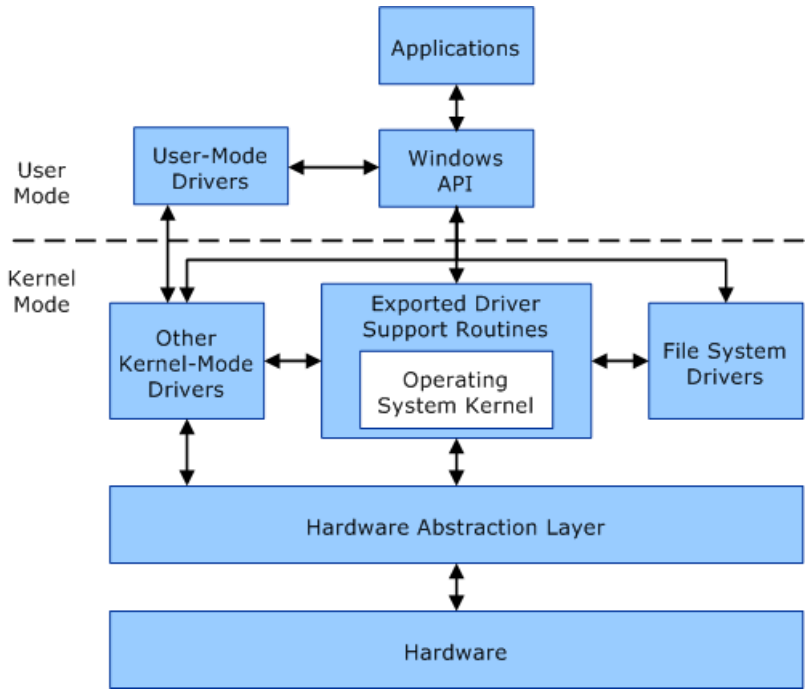
A processor in a computer running Windows has two different modes: *user mode* and *kernel mode*. The processor switches between the two modes depending on what type of code is running on the processor. Applications run in user mode, and core operating system components run in kernel mode. While many drivers run in kernel mode, some drivers may run in user mode.

When you start a user-mode application, Windows creates a *process* for the application. The process provides the application with a private *virtual address space* and a private *handle table*. Because an application's virtual address space is private, one application cannot alter data that belongs to another application. Each application runs in isolation, and if an application crashes, the crash is limited to that one application. Other applications and the operating system are not affected by the crash.

In addition to being private, the virtual address space of a user-mode application is limited. A processor running in user mode cannot access virtual addresses that are reserved for the operating system. Limiting the virtual address space of a user-mode application prevents the application from altering, and possibly damaging, critical operating system data.

All code that runs in kernel mode shares a single virtual address space. This means that a kernel-mode driver is not isolated from other drivers and the operating system itself. If a kernel-mode driver accidentally writes to the wrong virtual address, data that belongs to the operating system or another driver could be compromised. If a kernel-mode driver crashes, the entire operating system crashes.

This diagram illustrates communication between user-mode and kernel-mode components.



Related topics

Virtual Address Spaces

Virtual address spaces

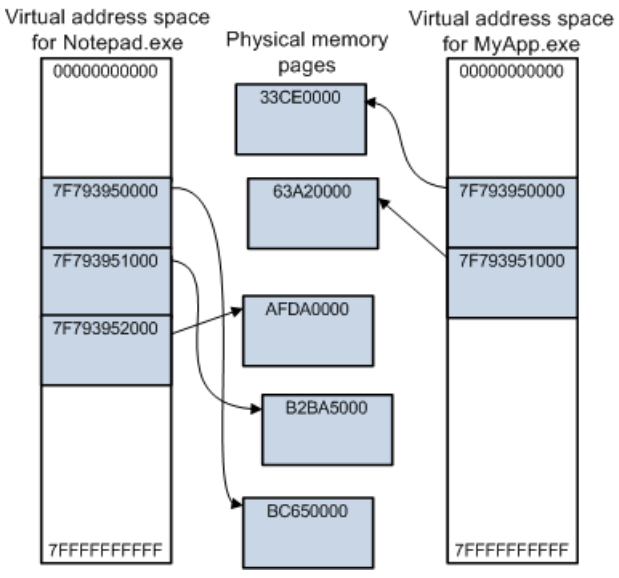
4/24/2020 • 4 minutes to read • Edit Online

When a processor reads or writes to a memory location, it uses a virtual address. As part of the read or write operation, the processor translates the virtual address to a physical address. Accessing memory through a virtual address has these advantages:

- A program can use a contiguous range of virtual addresses to access a large memory buffer that is not contiguous in physical memory.
- A program can use a range of virtual addresses to access a memory buffer that is larger than the available physical memory. As the supply of physical memory becomes small, the memory manager saves pages of physical memory (typically 4 kilobytes in size) to a disk file. Pages of data or code are moved between physical memory and the disk as needed.
- The virtual addresses used by different processes are isolated from each other. The code in one process cannot alter the physical memory that is being used by another process or the operating system.

The range of virtual addresses that is available to a process is called the *virtual address space* for the process. Each user-mode process has its own private virtual address space. For a 32-bit process, the virtual address space is usually the 2-gigabyte range 0x00000000 through 0x7FFFFFFF. For a 64-bit process on 64-bit Windows, virtual address space is the 128-terabyte range 0x000'00000000 through 0x7FFF'FFFFFFFF. A range of virtual addresses is sometimes called a range of *virtual memory*. For more info, see Memory and Address Space Limits.

This diagram illustrates some of the key features of virtual address spaces.

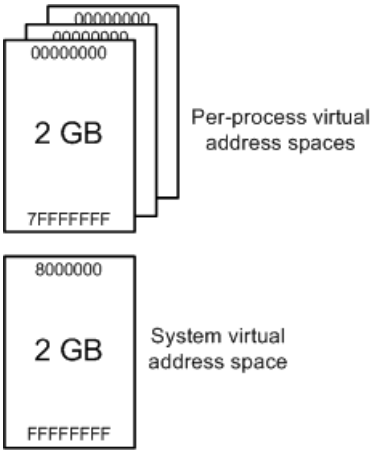


The diagram shows the virtual address spaces for two 64-bit processes: Notepad.exe and MyApp.exe. Each process has its own virtual address space that goes from 0x000'0000000 through 0x7FF'FFFFFFFF. Each shaded block represents one page (4 kilobytes in size) of virtual or physical memory. Notice that the Notepad process uses three contiguous pages of virtual addresses, starting at 0x7F7'93950000. But those three contiguous pages of virtual addresses are mapped to noncontiguous pages in physical memory. Also notice that both processes use a page of virtual memory beginning at 0x7F7'93950000, but those virtual pages are mapped to different pages of physical memory.

User space and system space

Processes like Notepad.exe and MyApp.exe run in user mode. Core operating system components and many drivers run in the more privileged kernel mode. For more information about processor modes, see User mode and kernel mode. Each user-mode process has its own private virtual address space, but all code that runs in kernel mode shares a single virtual address space called *system space*. The virtual address space for a user-mode process is called *user space*.

In 32-bit Windows, the total available virtual address space is 2^32 bytes (4 gigabytes). Usually the lower 2 gigabytes are used for user space, and the upper 2 gigabytes are used for system space.



In 32-bit Windows, you have the option of specifying (at boot time) that more than 2 gigabytes are available for user space. The consequence is that fewer virtual addresses are available for system space. You can increase the size of user space to as much as 3 gigabytes, in which case only 1 gigabyte is available for system space. To increase the size of user space, use **BCDEdit /set increaseuserva**.

In 64-bit Windows, the theoretical amount of virtual address space is 2^64 bytes (16 exabytes), but only a small portion of the 16-exabyte range is actually used.

Code running in user mode has access to user space but does not have access to system space. This restriction prevents user-mode code from reading or altering protected operating system data structures. Code running in kernel mode has access to both user space and system space. That is, code running in kernel mode has access to system space and the virtual address space of the current user-mode process.

Drivers that run in kernel mode must be very careful about directly reading from or writing to addresses in user space. This scenario illustrates why.

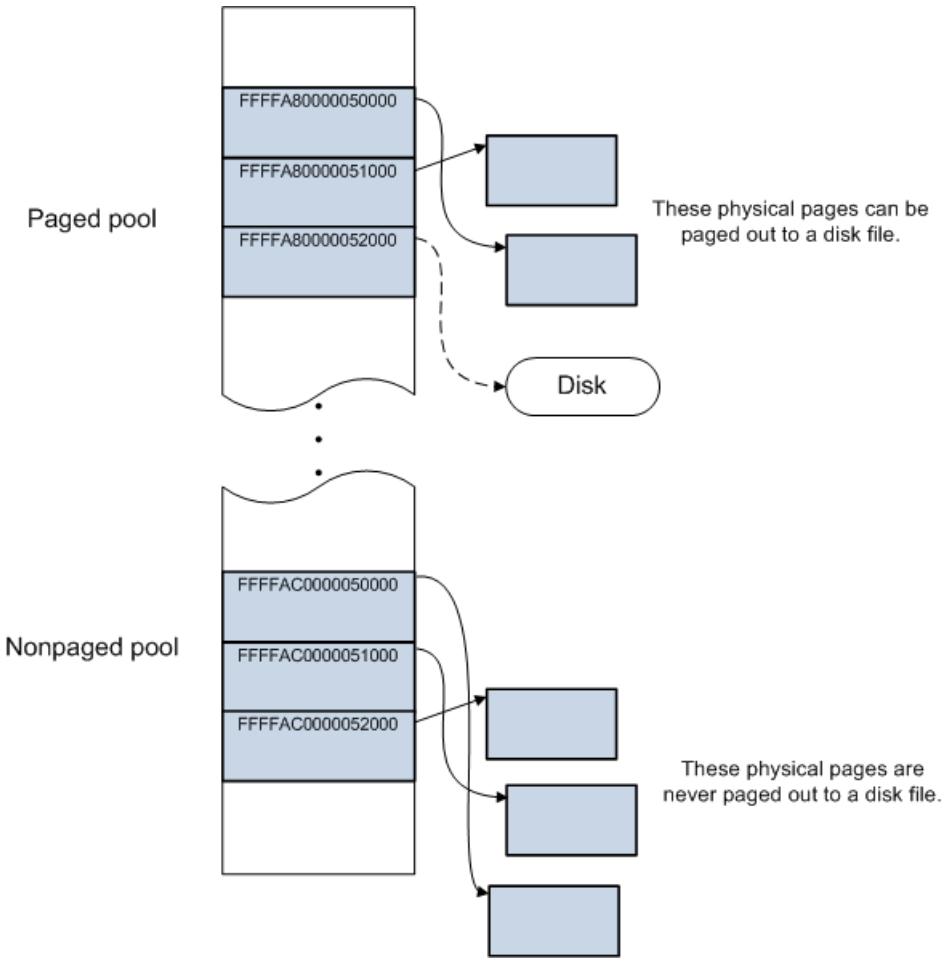
1. A user-mode program initiates a request to read some data from a device. The program supplies the starting address of a buffer to receive the data.
2. A device driver routine, running in kernel mode, starts the read operation and returns control to its caller.
3. Later the device interrupts whatever thread is currently running to say that the read operation is complete. The interrupt is handled by kernel-mode driver routines running on this arbitrary thread, which belongs to an arbitrary process.
4. At this point, the driver must not write the data to the starting address that the user-mode program supplied in Step 1. This address is in the virtual address space of the process that initiated the request, which is most likely not the same as the current process.

Paged pool and Nonpaged pool

In user space, all physical memory pages can be paged out to a disk file as needed. In system space, some physical pages can be paged out and others cannot. System space has two regions for dynamically allocating memory: paged pool and nonpaged pool.

Memory that is allocated in paged pool can be paged out to a disk file as needed. Memory that is allocated in

nonpaged pool can never be paged out to a disk file.



Related topics

User mode and kernel mode

Device nodes and device stacks

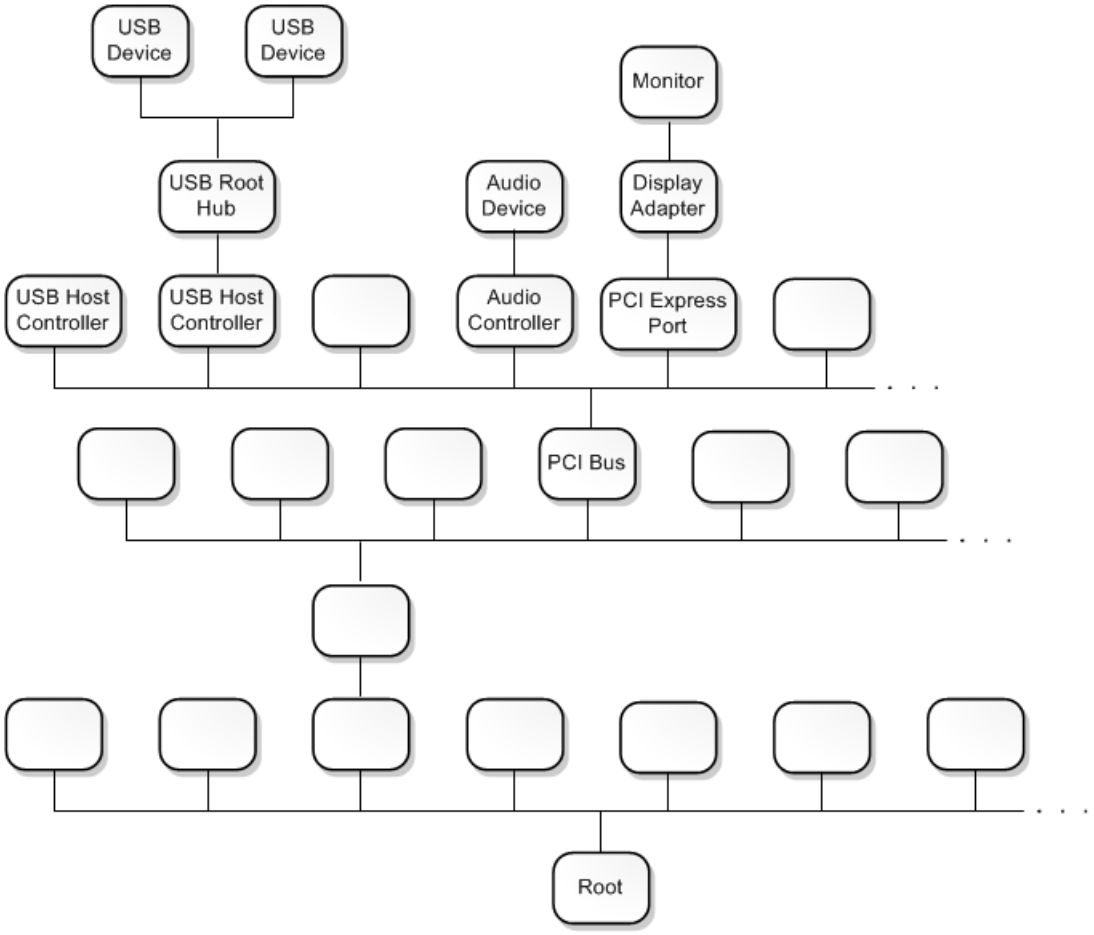
10/23/2019 • 7 minutes to read • Edit Online

In Windows, devices are represented by device nodes in the Plug and Play (PnP) device tree. Typically, when an I/O request is sent to a device, several drivers help handle the request. Each of these drivers is associated with a device object, and the device objects are arranged in a stack. The sequence of device objects along with their associated drivers is called a device stack. Each device node has its own device stack.

Device nodes and the Plug and Play device tree

Windows organizes devices in a tree structure called the *Plug and Play device tree*, or simply the *device tree*. Typically, a node in the device tree represents either a device or an individual function on a composite device. However, some nodes represent software components that have no association with physical devices.

A node in the device tree is called a *device node*. The root node of the device tree is called the *root device node*. By convention, the root device node is drawn at the bottom of the device tree, as shown in the following diagram.



The device tree illustrates the parent/child relationships that are inherent in the PnP environment. Several of the nodes in the device tree represent buses that have child devices connected to them. For example, the PCI Bus node represents the physical PCI bus on the motherboard. During startup, the PnP manager asks the PCI bus driver to enumerate the devices that are connected to the PCI bus. Those devices are represented by child nodes of the PCI Bus node. In the preceding diagram, the PCI Bus node has child nodes for several devices that are connected to the PCI bus, including USB host controllers, an audio controller, and a PCI Express port.

Some of the devices connected to the PCI bus are buses themselves. The PnP manager asks each of these buses to enumerate the devices that are connected to it. In the preceding diagram, we can see that the audio controller

is a bus that has an audio device connected to it. We can see that the PCI Express port is a bus that has a display adapter connected to it, and the display adapter is a bus that has one monitor connected to it.

Whether you think of a node as representing a device or a bus depends on your point of view. For example, you can think of the display adapter as a device that plays a key role in preparing frames that appear on the screen. However, you can also think of the display adapter as a bus that is capable of detecting and enumerating connected monitors.

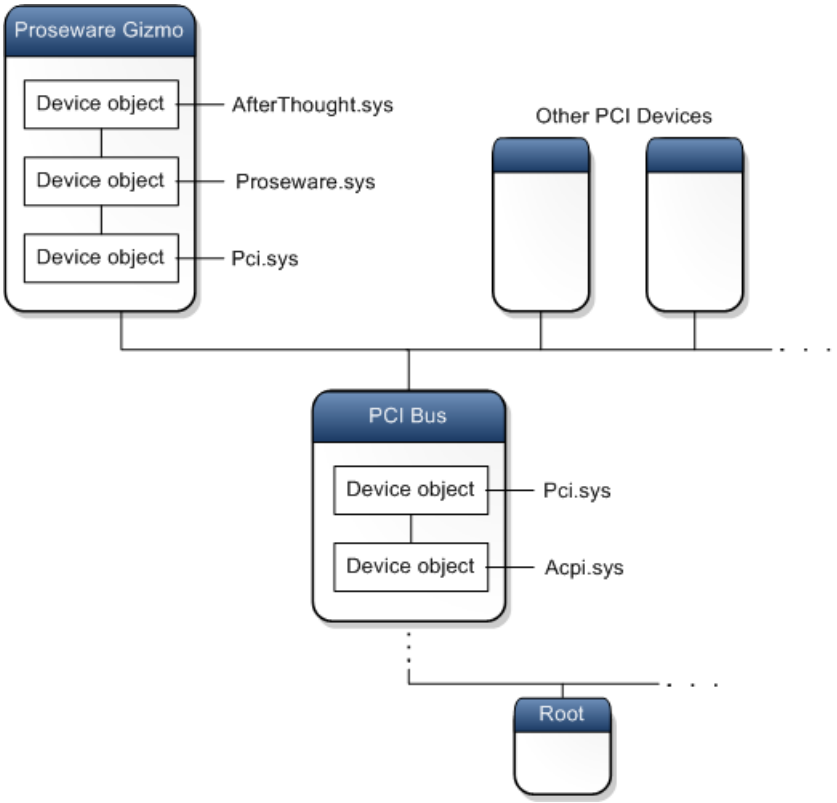
Device objects and device stacks

A *device object* is an instance of a DEVICE_OBJECT structure. Each device node in the PnP device tree has an ordered list of device objects, and each of these device objects is associated with a driver. The ordered list of device objects, along with their associated drivers, is called the *device stack* for the device node.

You can think of a device stack in several ways. In the most formal sense, a device stack is an ordered list of (device object, driver) pairs. However, in certain contexts it might be useful to think of the device stack as an ordered list of device objects. In other contexts, it might be useful to think of the device stack as an ordered list of drivers.

By convention, a device stack has a top and a bottom. The first device object to be created in the device stack is at the bottom, and the last device object to be created and attached to the device stack is at the top.

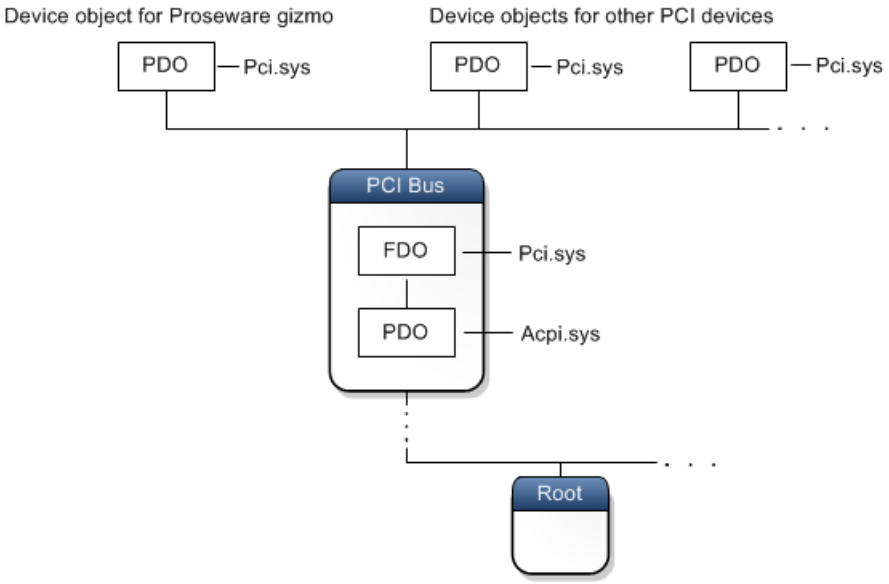
In the following diagram, the Proseware Gizmo device node has a device stack that contains three (device object, driver) pairs. The top device object is associated with the driver AfterThought.sys, the middle device object is associated with the driver Proseware.sys, and the bottom device object is associated with the driver Pci.sys. The PCI Bus node in the center of the diagram has a device stack that contains two (device object, driver) pairs--a device object associated with Pci.sys and a device object associated with Acpi.sys.



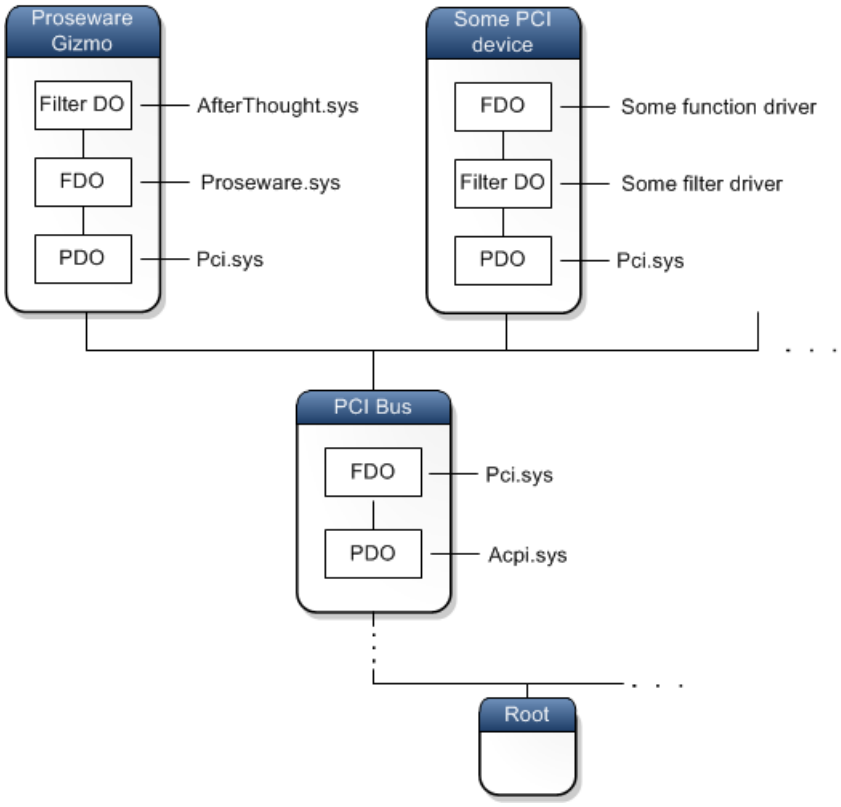
How does a device stack get constructed?

During startup, the PnP manager asks the driver for each bus to enumerate child devices that are connected to the bus. For example, the PnP manager asks the PCI bus driver (Pci.sys) to enumerate the devices that are connected to the PCI bus. In response to this request, Pci.sys creates a device object for each device that is

connected to the PCI bus. Each of these device objects is called a *physical device object* (PDO). Shortly after Pci.sys creates the set of PDOs, the device tree looks like the one shown in the following diagram.



The PnP manager associates a device node with each newly created PDO and looks in the registry to determine which drivers need to be part of the device stack for the node. The device stack must have one (and only one) *function driver* and can optionally have one or more *filter drivers*. The function driver is the main driver for the device stack and is responsible for handling read, write, and device control requests. Filter drivers play auxiliary roles in processing read, write, and device control requests. As each function and filter driver is loaded, it creates a device object and attaches itself to the device stack. A device object created by the function driver is called a *functional device object* (FDO), and a device object created by a filter driver is called a *filter device object* (Filter DO). Now the device tree looks something like this diagram.



In the diagram, notice that in one node, the filter driver is above the function driver, and in the other node, the filter driver is below the function driver. A filter driver that is above the function driver in a device stack is called an *upper filter driver*. A filter driver that is below the function driver is called a *lower filter driver*.

The PDO is always the bottom device object in a device stack. This results from the way a device stack is

constructed. The PDO is created first, and as additional device objects are attached to the stack, they are attached to the top of the existing stack.

Note When the drivers for a device are installed, the installer uses information in an information (INF) file to determine which driver is the function driver and which drivers are filters. Typically the INF file is provided either by Microsoft or by the hardware vendor. After the drivers for a device are installed, the PnP manager can determine the function and filter drivers for the device by looking in the registry.

Bus drivers

In the preceding diagram, you can see that the driver Pci.sys plays two roles. First, Pci.sys is associated with the FDO in the PCI Bus device node. In fact, it created the FDO in the PCI Bus device node. So Pci.sys is the function driver for the PCI bus. Second, Pci.sys is associated with the PDO in each child of the PCI Bus node. Recall that it created the PDOs for the child devices. The driver that creates the PDO for a device node is called the *bus driver* for the node.

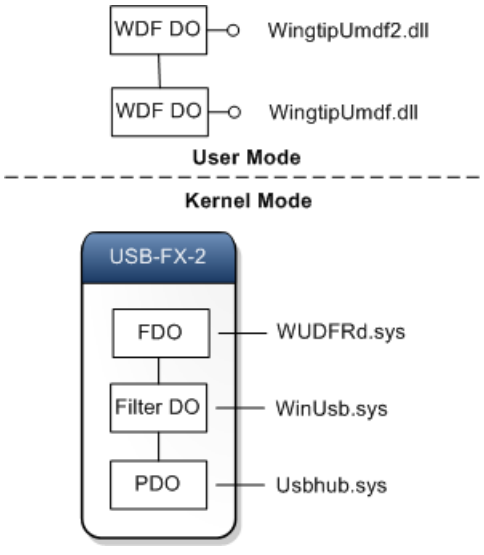
If your point of reference is the PCI bus, then Pci.sys is the function driver. But if your point of reference is the Proseware Gizmo device, then Pci.sys is the bus driver. This dual role is typical in the PnP device tree. A driver that serves as function driver for a bus also serves as bus driver for a child device of the bus.

User-mode device stacks

So far we've been discussing kernel-mode device stacks. That is, the drivers in the stacks run in kernel mode, and the device objects are mapped into system space, which is the address space that is available only to code running in kernel mode. For information about the difference between kernel mode and user mode, see User mode and kernel mode.

In some cases, a device has a user-mode device stack in addition to its kernel-mode device stack. User-mode drivers are often based on the User-Mode Driver Framework (UMDF), which is one of the driver models provided by the Windows Driver Frameworks (WDF). In UMDF, the drivers are user-mode DLLs, and the device objects are COM objects that implement the IWDFDevice interface. A device object in a UMDF device stack is called a *WDF device object* (WDF DO).

The following diagram shows the device node, kernel-mode device stack, and the user-mode device stack for a USB-FX-2 device. The drivers in both the user-mode and kernel-mode stacks participate in I/O requests that are directed at the USB-FX-2 device.



Related topics

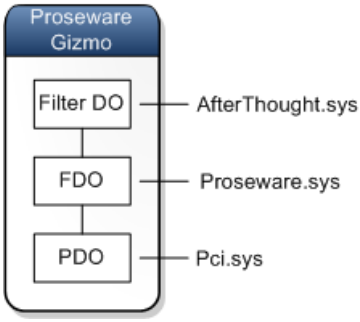
Concepts for all driver developers

I/O request packets

10/23/2019 • 2 minutes to read • Edit Online

Most of the requests that are sent to device drivers are packaged in I/O request packets (**IRP**s). An operating system component or a driver sends an IRP to a driver by calling **IoCallDriver**, which has two parameters: a pointer to a **DEVICE_OBJECT** and a pointer to an **IRP**. The **DEVICE_OBJECT** has a pointer to an associated **DRIVER_OBJECT**. When a component calls **IoCallDriver**, we say the component *sends the IRP to the device object* or *sends the IRP to the driver associated with the device object*. Sometimes we use the phrase *passes the IRP* or *forwards the IRP* instead of *sends the IRP*.

Typically an IRP is processed by several drivers that are arranged in a stack. Each driver in the stack is associated with a device object. For more information, see Device nodes and device stacks. When an **IRP** is processed by a device stack, the **IRP** is usually sent first to the top device object in the device stack. For example, if an **IRP** is processed by the device stack shown in this diagram, the IRP would be sent first to the filter device object (Filter DO) at the top of the device stack.



Passing an IRP down the device stack

Suppose the I/O manager sends an IRP to the Filter DO in the diagram. The driver associated with the Filter DO, AfterThought.sys, processes the IRP and then passes it to the functional device object (FDO), which is the next lower device object in the device stack. When a driver passes an IRP to the next lower device object in the device stack, we say the driver *passes the IRP down the device stack*.

Some IRPs are passed all the way down the device stack to the physical device object (PDO). Other IRPs never reach the PDO because they are completed by one of the drivers above the PDO.

IRPs are self-contained

The IRP structure is self-contained in the sense that it holds all of the information that a driver needs to handle an I/O request. Some parts of the IRP structure hold information that is common to all of the participating drivers in the stack. Other parts of the IRP hold information that is specific to a particular driver in the stack.

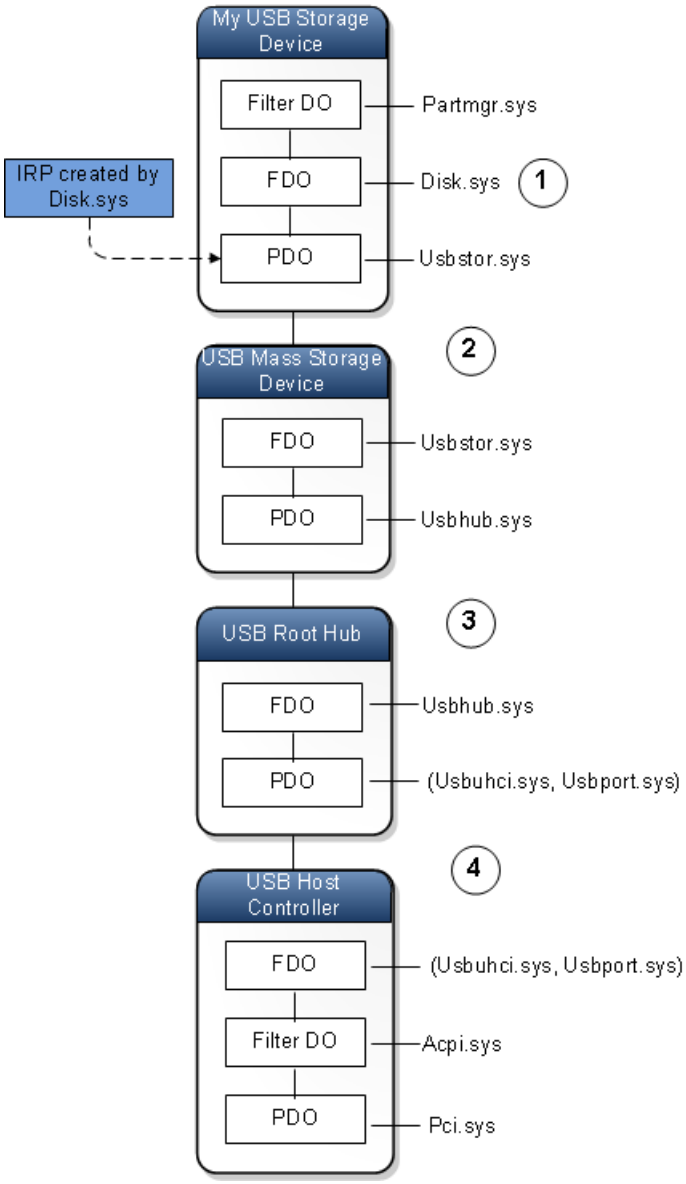
Driver stacks

10/23/2019 • 4 minutes to read • Edit Online

Most of the requests that are sent to device drivers are packaged in I/O request packets (IRPs). Each device is represented by a device node, and each device node has a device stack. For more information, see Device nodes and device stacks. To send a read, write, or control request to a device, the I/O manager locates the device node for the device and then sends an IRP to the device stack of that node. Sometimes more than one device stack is involved in processing an I/O request. Regardless of how many device stacks are involved, the overall sequence of drivers that participate in an I/O request is called the *driver stack* for the request. We also use the term *driver stack* to refer to the layered set of drivers for a particular technology.

I/O requests that are processed by several device stacks

In some cases, more than one device stack is involved in processing an IRP. The following diagram illustrates a case where four device stacks are involved in processing a single IRP.



Here is how the IRP is processed at each numbered stage in the diagram:

1. The IRP is created by Disk.sys, which is the function driver in the device stack for the My USB Storage

Device node. Disk.sys passes the IRP down the device stack to Usbstor.sys.

2. Notice that Usbstor.sys is the PDO driver for the My USB Storage Device node and the FDO driver for the USB Mass Storage Device node. At this point, it is not important to decide whether the IRP is owned by the (PDO, Usbstor.sys) pair or the (FDO, Usbstor.sys) pair. The IRP is owned by the driver, Usbstor.sys, and the driver has access to both the PDO and the FDO.

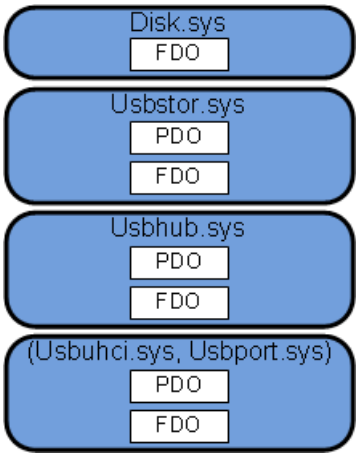
3. When Usbstor.sys has finished processing the IRP, it passes the IRP to Usbhub.sys. Usbhub.sys is the PDO driver for the USB Mass Storage Device node and the FDO driver for the USB Root Hub node. It is not important to decide whether the IRP is owned by the (PDO, Usbhub.sys) pair or the (FDO, Usbhub.sys) pair. The IRP is owned by the driver, Usbhub.sys, and the driver has access to both the PDO and the FDO.

4. When Usbhub.sys has finished processing the IRP, it passes the IRP to the (Usbuhci.sys, Usbport.sys) pair.

   Usbuhci.sys is a miniport driver, and Usbport.sys is a port driver. The (miniport, port) pair plays the role of a single driver. In this case, both the miniport driver and the port driver are written by Microsoft. The (Usbuhci.sys, Usbport.sys) pair is the PDO driver for the USB Root Hub node, and the (Usbuhci.sys, Usbport.sys) pair is also the FDO driver for the USB Host Controller node. The (Usbuhci.sys, Usbport.sys) pair does the actual communication with the host controller hardware, which in turn communicates with the physical USB storage device.

The driver stack for an I/O request

Consider the sequence of four drivers that participated in the I/O request illustrated in the preceding diagram. We can get another view of the sequence by focusing on the drivers rather than on the device nodes and their individual device stacks. The following diagram shows the drivers in sequence from top to bottom. Notice that Disk.sys is associated with one device object, but each of the other three drivers is associated with two device objects.

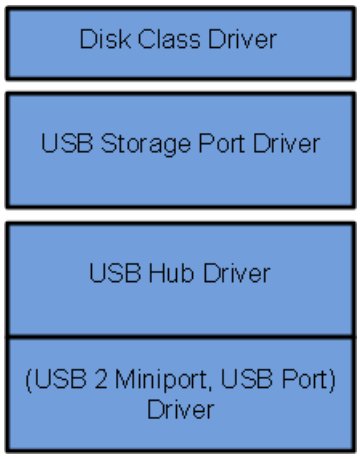


The sequence of drivers that participate in an I/O request is called the *driver stack for the I/O request*. To illustrate a driver stack for an I/O request, we draw the drivers from top to bottom in the order that they participate in the request.

Notice that the driver stack for an I/O request is quite different from the device stack for a device node. Also notice that the driver stack for an I/O request does not necessarily remain in one branch of the device tree.

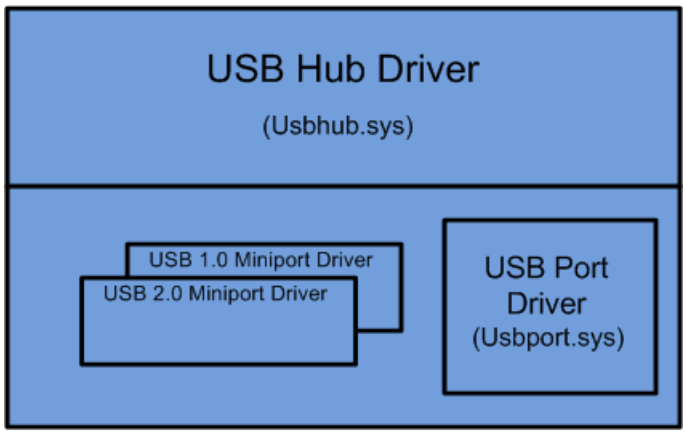
Technology driver stacks

Consider the driver stack for the I/O request shown in the preceding diagram. If we give each of the drivers a friendly name and make some slight changes to the diagram, we have a block diagram that is similar to many of those that appear in the Windows Driver Kit (WDK) documentation.



In the diagram, the driver stack is divided into three sections. We can think of each section as belonging to a particular technology or to a particular component or portion of the operating system. For example, we might say that the first section at the top of the driver stack belongs to the Volume Manager, the second section belongs to the storage component of the operating system, and the third section belongs to the core USB portion of the operating system.

Consider the drivers in the third section. These drivers are a subset of a larger set of core USB drivers that Microsoft provides for handling various kinds of USB requests and USB hardware. The following diagram shows what the entire USB core block diagram might look like.



A block diagram that shows all of the drivers for a particular technology or a particular component or portion of the operating system is called a *technology driver stack*. Typically, technology driver stacks are given names like the USB Core Driver Stack, the Storage Stack, the 1394 Driver Stack, and the Audio Driver Stack.

Note The USB core block diagram in this topic shows one of several possible ways to illustrate the technology driver stacks for USB 1.0 and 2.0. For the official diagrams of the USB 1.0, 2.0, and 3.0 driver stacks, see USB Driver Stack Architecture.

Related topics

Device nodes and device stacks

Minidrivers and driver pairs

Concepts for all driver developers

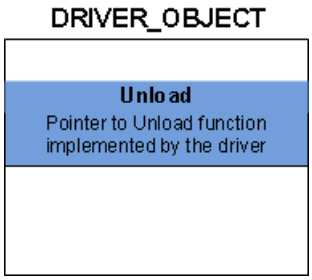
Minidrivers, Miniport drivers, and driver pairs

10/23/2019 • 9 minutes to read • Edit Online

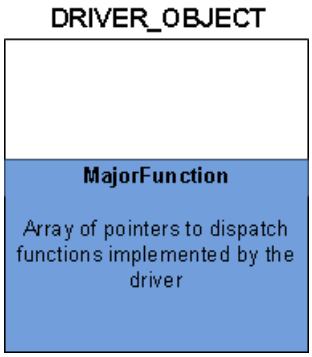
A minidriver or a miniport driver acts as half of a driver pair. Driver pairs like (miniport, port) can make driver development easier. In a driver pair, one driver handles general tasks that are common to a whole collection of devices, while the other driver handles tasks that are specific to an individual device. The drivers that handle device-specific tasks go by a variety of names, including miniport driver, miniclass driver, and minidriver.

Microsoft provides the general driver, and typically an independent hardware vendor provides the specific driver. Before you read this topic, you should understand the ideas presented in Device nodes and device stacks and I/O request packets.

Every kernel-mode driver must implement a function named **DriverEntry**, which gets called shortly after the driver is loaded. The **DriverEntry** function fills in certain members of a **DRIVER_OBJECT** structure with pointers to several other functions that the driver implements. For example, the **DriverEntry** function fills in the **Unload** member of the **DRIVER_OBJECT** structure with a pointer to the driver's *Unload* function, as shown in the following diagram.



The **MajorFunction** member of the **DRIVER_OBJECT** structure is an array of pointers to functions that handle I/O request packets (**IRP**s), as shown in the following diagram. Typically the driver fills in several members of the **MajorFunction** array with pointers to functions (implemented by the driver) that handle various kinds of IRPs.



An IRP can be categorized according to its major function code, which is identified by a constant, such as **IRP_MJ_READ**, **IRP_MJ_WRITE**, or **IRP_MJ_PNP**. The constants that identify major function code serve as indices in the **MajorFunction** array. For example, suppose the driver implements a dispatch function to handle IRPs that have the major function code **IRP_MJ_WRITE**. In this case, the driver must fill in the **MajorFunction**[IRP_MJ_WRITE] element of the array with a pointer to the dispatch function.

Typically the driver fills in some of the elements of the **MajorFunction** array and leaves the remaining elements set to default values provided by the I/O manager. The following example shows how to use the **!drvobj** debugger extension to inspect the function pointers for the parport driver.

```
0: kd> !drvobj parport 2
Driver object (fffffa80048d9e70) is for:
 \Driver\Parport
DriverEntry:   fffff880065ea070 parport!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  fffff880065e131c parport!PptUnload
AddDevice:     fffff880065d2008 parport!P5AddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff880065d49d0    parport!PptDispatchCreateOpen
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff880065d4a78    parport!PptDispatchClose
[03] IRP_MJ_READ                        fffff880065d4bac    parport!PptDispatchRead
[04] IRP_MJ_WRITE                       fffff880065d4bac    parport!PptDispatchRead
[05] IRP_MJ_QUERY_INFORMATION           fffff880065d4c40    parport!PptDispatchQueryInformation
[06] IRP_MJ_SET_INFORMATION             fffff880065d4ce4    parport!PptDispatchSetInformation
[07] IRP_MJ_QUERY_EA                    fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              fffff880065d4be8    parport!PptDispatchDeviceControl
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff880065d4c24    parport!PptDispatchInternalDeviceControl
[10] IRP_MJ_SHUTDOWN                    fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     fffff880065d4af4    parport!PptDispatchCleanup
[13] IRP_MJ_CREATE_MAILSLOT             fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       fffff880065d491c    parport!PptDispatchPower
[17] IRP_MJ_SYSTEM_CONTROL              fffff880065d4d4c    parport!PptDispatchSystemControl
[18] IRP_MJ_DEVICE_CHANGE               fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         fffff880065d4840    parport!PptDispatchPnp
```

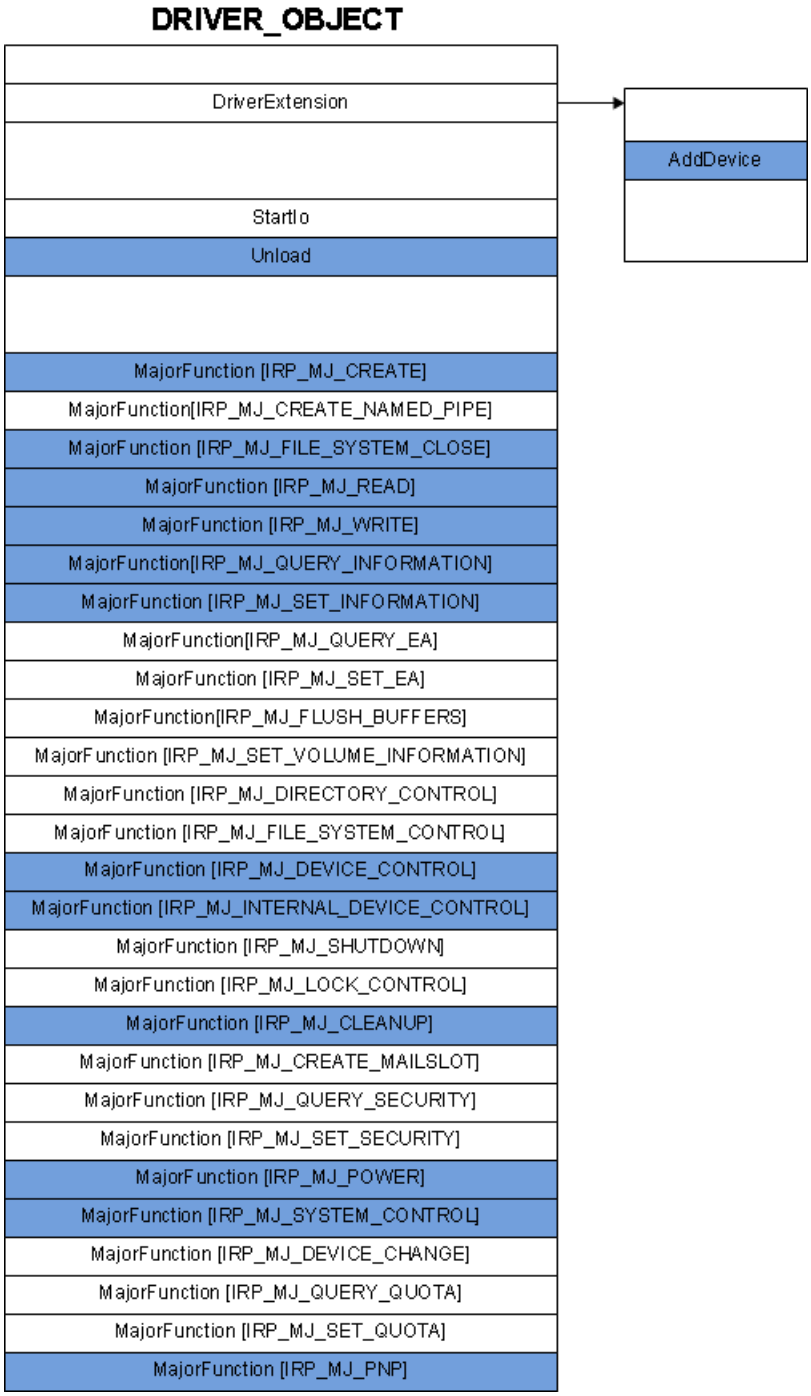
In the debugger output, you can see that parport.sys implements **GsDriverEntry**, the entry point for the driver. **GsDriverEntry**, which was generated automatically when the driver was built, performs some initialization and then calls **DriverEntry**, which was implemented by the driver developer.

You can also see that the parport driver (in its **DriverEntry** function) provides pointers to dispatch functions for these major function codes:

- IRP_MJ_CREATE
- IRP_MJ_CLOSE
- IRP_MJ_READ
- IRP_MJ_WRITE
- IRP_MJ_QUERY_INFORMATION
- IRP_MJ_SET_INFORMATION
- IRP_MJ_DEVICE_CONTROL
- IRP_MJ_INTERNAL_DEVICE_CONTROL
- IRP_MJ_CLEANUP
- IRP_MJ_POWER
- IRP_MJ_SYSTEM_CONTROL
- IRP_MJ_PNP

The remaining elements of the **MajorFunction** array hold pointers to the default dispatch function **nt!IopInvalidDeviceRequest**.

In the debugger output, you can see that the parport driver provided function pointers for *Unload* and *AddDevice*, but did not provide a function pointer for *StartIo*. The *AddDevice* function is unusual because its function pointer is not stored in the **DRIVER_OBJECT** structure. Instead, it is stored in the **AddDevice** member of an extension to the **DRIVER_OBJECT** structure. The following diagram illustrates the function pointers that the parport driver provided in its **DriverEntry** function. The function pointers provided by parport are shaded.



Making it easier by using driver pairs

Over a period of time, as driver developers inside and outside of Microsoft gained experience with the Windows Driver Model (WDM), they realized a couple of things about dispatch functions:

- Dispatch functions are largely boilerplate. For example, much of the code in the dispatch function for IRP_MJ_PNP is the same for all drivers. It is only a small portion of the Plug and Play (PnP) code that is specific to an individual driver that controls an individual piece of hardware.
- Dispatch functions are complicated and difficult to get right. Implementing features like thread synchronization, IRP queuing, and IRP cancellation is challenging and requires a deep understanding of how the operating system works.

To make things easier for driver developers, Microsoft created several technology-specific driver models. At first glance, the technology-specific models seem quite different from each other, but a closer look reveals that many of them are based on this paradigm:

- The driver is split into two pieces: one that handles the general processing and one that handles processing specific to a particular device.
- The general piece is written by Microsoft.
- The specific piece may be written by Microsoft or an independent hardware vendor.

Suppose that the Proseware and Contoso companies both make a toy robot that requires a WDM driver. Also suppose that Microsoft provides a General Robot Driver called GeneralRobot.sys. Proseware and Contoso can each write small drivers that handle the requirements of their specific robots. For example, Proseware could write ProsewareRobot.sys, and the pair of drivers (ProsewareRobot.sys, GeneralRobot.sys) could be combined to form a single WDM driver. Likewise, the pair of drivers (ContosoRobot.sys, GeneralRobot.sys) could combine to form a single WDM driver. In its most general form, the idea is that you can create drivers by using (specific.sys, general.sys) pairs.

Function pointers in driver pairs

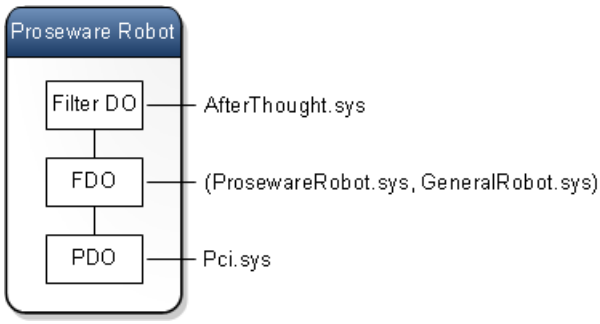
In a (specific.sys, general.sys) pair, Windows loads specific.sys and calls its **DriverEntry** function. The **DriverEntry** function of specific.sys receives a pointer to a **DRIVER_OBJECT** structure. Normally you would expect **DriverEntry** to fill in several elements of the **MajorFunction** array with pointers to dispatch functions. Also you would expect **DriverEntry** to fill in the **Unload** member (and possibly the **StartIo** member) of the **DRIVER_OBJECT** structure and the **AddDevice** member of the driver object extension. However, in a driver pair model, **DriverEntry** does not necessarily do this. Instead the **DriverEntry** function of specific.sys passes the **DRIVER_OBJECT** structure along to an initialization function implemented by general.sys. The following code example shows how the initialization function might be called in the (ProsewareRobot.sys, GeneralRobot.sys) pair.

```
PVOID g_ProsewareRobottCallbacks[3] = {DeviceControlCallback, PnpCallback, PowerCallback};

// DriverEntry function in ProsewareRobot.sys
NTSTATUS DriverEntry (DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
{
   // Call the initialization function implemented by GeneralRobot.sys.
   return GeneralRobotInit(DriverObject, RegistryPath, g_ProsewareRobottCallbacks);
}
```

The initialization function in GeneralRobot.sys writes function pointers to the appropriate members of the **DRIVER_OBJECT** structure (and its extension) and the appropriate elements of the **MajorFunction** array. The idea is that when the I/O manager sends an IRP to the driver pair, the IRP goes first to a dispatch function implemented by GeneralRobot.sys. If GeneralRobot.sys can handle the IRP on its own, then the specific driver, ProsewareRobot.sys, does not have to be involved. If GeneralRobot.sys can handle some, but not all, of the IRP processing, it gets help from one of the callback functions implemented by ProsewareRobot.sys. GeneralRobot.sys receives pointers to the ProsewareRobot callbacks in the GeneralRobotInit call.

At some point after **DriverEntry** returns, a device stack gets constructed for the Proseware Robot device node. The device stack might look like this.



As shown in the preceding diagram, the device stack for Proseware Robot has three device objects. The top device object is a filter device object (Filter DO) associated with the filter driver AfterThought.sys. The middle device object is a functional device object (FDO) associated with the driver pair (ProsewareRobot.sys, GeneralRobot.sys). The driver pair serves as the function driver for the device stack. The bottom device object is a physical device object (PDO) associated with Pci.sys.

Notice that the driver pair occupies only one level in the device stack and is associated with only one device object: the FDO. When GeneralRobot.sys processes an IRP, it might call ProsewareRobot.sys for assistance, but that is not the same as passing the request down the device stack. The driver pair forms a single WDM driver that is at one level in the device stack. The driver pair either completes the IRP or passes it down the device stack to the PDO, which is associated with Pci.sys.

Example of a driver pair

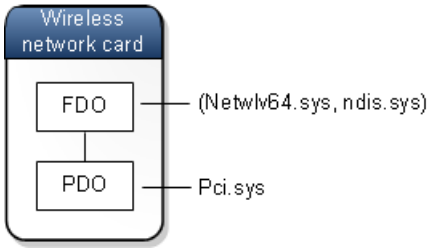
Suppose you have a wireless network card in your laptop computer, and by looking in Device Manager, you determine that netwlv64.sys is the driver for the network card. You can use the **!drvobj** debugger extension to inspect the function pointers for netwlv64.sys.

```
1: kd> !drvobj netwlv64 2
Driver object (fffffa8002e5f420) is for:
 \Driver\netwlv64
DriverEntry:   fffff8800482f064 netwlv64!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  fffff8800195c5f4 ndis!ndisMUnloadEx
AddDevice:     fffff88001940d30 ndis!ndisPnPAddDevice
Dispatch routines:
[00] IRP_MJ_CREATE                      fffff880018b5530 ndis!ndisCreateIrpHandler
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff88001936f00 ndis!ndisDummyIrpHandler
[02] IRP_MJ_CLOSE                       fffff880018b5870 ndis!ndisCloseIrpHandler
[03] IRP_MJ_READ                        fffff88001936f00 ndis!ndisDummyIrpHandler
[04] IRP_MJ_WRITE                       fffff88001936f00 ndis!ndisDummyIrpHandler
[05] IRP_MJ_QUERY_INFORMATION           fffff88001936f00 ndis!ndisDummyIrpHandler
[06] IRP_MJ_SET_INFORMATION             fffff88001936f00 ndis!ndisDummyIrpHandler
[07] IRP_MJ_QUERY_EA                    fffff88001936f00 ndis!ndisDummyIrpHandler
[08] IRP_MJ_SET_EA                      fffff88001936f00 ndis!ndisDummyIrpHandler
[09] IRP_MJ_FLUSH_BUFFERS               fffff88001936f00 ndis!ndisDummyIrpHandler
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    fffff88001936f00 ndis!ndisDummyIrpHandler
[0b] IRP_MJ_SET_VOLUME_INFORMATION      fffff88001936f00 ndis!ndisDummyIrpHandler
[0c] IRP_MJ_DIRECTORY_CONTROL           fffff88001936f00 ndis!ndisDummyIrpHandler
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         fffff88001936f00 ndis!ndisDummyIrpHandler
[0e] IRP_MJ_DEVICE_CONTROL              fffff8800193696c ndis!ndisDeviceControlIrpHandler
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff880018f9114 ndis!ndisDeviceInternalIrpDispatch
[10] IRP_MJ_SHUTDOWN                    fffff88001936f00 ndis!ndisDummyIrpHandler
[11] IRP_MJ_LOCK_CONTROL                fffff88001936f00 ndis!ndisDummyIrpHandler
[12] IRP_MJ_CLEANUP                     fffff88001936f00 ndis!ndisDummyIrpHandler
[13] IRP_MJ_CREATE_MAILSLOT             fffff88001936f00 ndis!ndisDummyIrpHandler
[14] IRP_MJ_QUERY_SECURITY              fffff88001936f00 ndis!ndisDummyIrpHandler
[15] IRP_MJ_SET_SECURITY                fffff88001936f00 ndis!ndisDummyIrpHandler
[16] IRP_MJ_POWER                       fffff880018c35e8 ndis!ndisPowerDispatch
[17] IRP_MJ_SYSTEM_CONTROL              fffff880019392c8 ndis!ndisWMIDispatch
[18] IRP_MJ_DEVICE_CHANGE               fffff88001936f00 ndis!ndisDummyIrpHandler
[19] IRP_MJ_QUERY_QUOTA                 fffff88001936f00 ndis!ndisDummyIrpHandler
[1a] IRP_MJ_SET_QUOTA                   fffff88001936f00 ndis!ndisDummyIrpHandler
[1b] IRP_MJ_PNP                         fffff8800193e518 ndis!ndisPnPDispatch
```

In the debugger output, you can see that netwlv64.sys implements **GsDriverEntry**, the entry point for the driver. **GsDriverEntry**, which was automatically generated when the driver was built, performs some initialization and then calls **DriverEntry**, which was written by the driver developer.

In this example, netwlv64.sys implements **DriverEntry**, but ndis.sys implements *AddDevice*, *Unload*, and several dispatch functions. Netwlv64.sys is called an NDIS miniport driver, and ndis.sys is called the NDIS Library. Together, the two modules form an (NDIS miniport, NDIS Library) pair.

This diagram shows the device stack for the wireless network card. Notice that the driver pair (netwlv64.sys, ndis.sys) occupies only one level in the device stack and is associated with only one device object: the FDO.



Available driver pairs

The different technology-specific driver models use a variety of names for the specific and general pieces of a driver pair. In many cases, the specific portion of the pair has the prefix "mini." Here are some of (specific, general) pairs that are available:

- (display miniport driver, display port driver)
- (audio miniport driver, audio port driver)
- (storage miniport driver, storage port driver)
- (battery miniclass driver, battery class driver)
- (HID minidriver, HID class driver)
- (changer miniclass driver, changer port driver)
- (NDIS miniport driver, NDIS library)

Note As you can see in the list, several of the models use the term *class driver* for the general portion of a driver pair. This kind of class driver is different from a standalone class driver and different from a class filter driver.

Related topics

Concepts for all driver developers

Device nodes and device stacks

Driver stacks

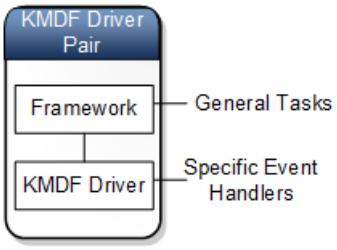
KMDF as a generic driver pair model

12/5/2018 • 3 minutes to read • Edit Online

In this topic, we discuss the idea that the Kernel Mode Driver Framework can be viewed as a generic driver pair model. Before you read this topic, you should understand the ideas presented in Minidrivers and driver pairs.

Over the years, Microsoft has created several technology-specific driver models that use this paradigm:

- The driver is split into two pieces: one that handles general processing and one that handles processing that is specific to a particular device.
- The general piece, called the Framework, is written by Microsoft.
- The specific piece, called the KMDF driver, may be written by Microsoft or an independent hardware vendor.



The Framework portion of the driver pair performs general tasks that are common to a wide variety of drivers. For example, the Framework can handle I/O request queues, thread synchronization, and a large portion of the power management duties.

The Framework owns the dispatch table for the KMDF driver, so when someone sends an I/O request packet (IRP) to the (KMDF driver, Framework) pair, the IRP goes to Framework. If the Framework can handle the IRP by itself, the KMDF driver is not involved. If the Framework cannot handle the IRP by itself, it gets help by calling event handlers implemented by the KMDF driver. Here are some examples of event handlers that might be implemented by a KMDF driver.

- EvtDevicePrepareHardware
- EvtIoRead
- EvtIoDeviceControl
- EvtInterruptIsr
- EvtInterruptDpc
- EvtDevicePnpStateChange

For example, a USB 2.0 host controller driver has a specific piece named usbehci.sys and a general piece named usbport.sys. Usbehci.sys, which is called the USB 2.0 Miniport driver, has code that is specific to USB 2.0 host controllers. Usbport.sys, which is called the USB Port driver, has general code that applies to both USB 2.0 and USB 1.0. The pair of drivers (usbehci.sys, usbport.sys) combine to form a single WDM driver for a USB 2.0 host controller.

The (specific, general) driver pairs have a variety of names across different device technologies. Most of the device-specific drivers have the prefix *mini*. The general drivers are often called port or class drivers. Here are some examples of (specific, general) pairs:

- (display miniport driver, display port driver)
- (USB miniport driver, USB port driver)
- (battery miniclass driver, battery class driver)
- (HID minidriver, HID class driver)

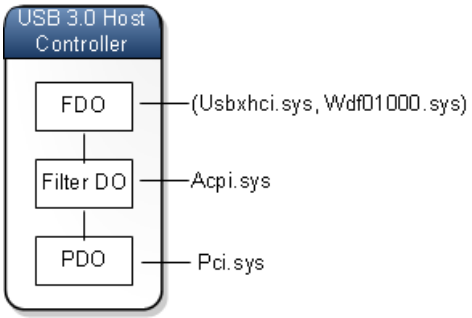
- (storage miniport driver, storage port driver)

As more and more driver pair models were developed, it became difficult to keep track of all the different ways to write a driver. Each model has it's own interface for communication between the device-specific driver and the general driver. The body of knowledge required to develop drivers for one device technology (for example, Audio) can be quite different from the body of knowledge required to develop drivers for another device technology (for example, Storage).

Over time, developers realized that it would be good to have a single unified model for kernel-mode driver pairs. The Kernel Mode Driver Framework (KMDF), which was first available in Windows Vista, fulfills that need. A driver based on KMDF uses a paradigm that is similar to many of the technology-specific driver pair models.

- The driver is split into two pieces: one that handles general processing and one that handles processing that is specific to a particular device.
- The general piece, which is written by Microsoft, is called the Framework.
- The specific piece, which is written by Microsoft or an independent hardware vendor, is called the KMDF driver.

The USB 3.0 host controller driver is an example of a driver based on KMDF. In this example, both drivers in the pair are written by Microsoft. The general driver is the Framework, and the device-specific driver is the USB 3.0 Host Controller Driver. This diagram illustrates the device node and device stack for a USB 3.0 host controller.



In the diagram, Usbxhci.sys is the USB 3.0 host controller driver. It is paired with Wdf01000.sys, which is the Framework. The (usbxhci.sys, wdf01000.sys) pair forms a single WDM driver that serves as the function driver for the USB 3.0 host controller. Notice that the driver pair occupies one level in the device stack and is represented by single device object. The single device object that represents the (usbxhci.sys, wdf01000.sys) pair is the functional device object (FDO) for the USB 3.0 host controller.

In a (KMDF driver, Framework) pair, the Framework handles tasks that are common to a wide variety of kernel-mode drivers. For example, the Framework can handle queuing of I/O requests, thread synchronization, most of the Plug and Play tasks, and most of the power management tasks. The KMDF driver handles tasks that require interaction with a specific device. The KMDF driver participates in processing requests by registering event handlers that the Framework calls as needed.

Related topics

Minidrivers and driver pairs

Kernel-Mode Driver Framework

KMDF extensions and driver triples

12/5/2018 • 2 minutes to read • Edit Online

In this topic, we discuss class-based extensions to the Kernel Mode Driver Framework (KMDF). Before you read this topic, you should understand the ideas presented in Minidrivers and driver pairs and KMDF as a Generic Driver Pair Model.

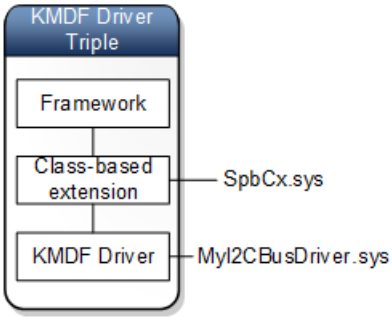
For some device classes, Microsoft provides KMDF extensions that further reduce the amount of processing that must be performed by KMDF drivers. A driver that uses a class-based KMDF extension has these three pieces, which we call a *driver triple*.

- The Framework, which handles tasks common to most all drivers
- The class-based framework extension, which handles tasks that are specific to a particular class of devices
- The KMDF driver, which handles tasks that are specific to a particular device.

The three drivers in a driver triple (KMDF driver, device-class KMDF extension, Framework) combine to form a single WDM driver.

An example of a device-class KMDF extension is SpbCx.sys, which is the KMDF extension for the Simple Peripheral Bus (SPB) device class. The SPB class includes synchronous serial buses such as I2C and SPI. A driver triple for an I2C bus controller would look like this:

- The Framework handles general tasks that are common to most all drivers.
- SpbCx.sys handles tasks that are specific to the SPB bus class. These are tasks that are common to all SPB busses.
- The KMDF driver handles tasks that are specific to an I2C bus. Let's call this driver MyI2CBusDriver.sys.



The three drivers in the driver triple (MyI2CBusDriver.sys, SpbCx.sys, Wdf01000.sys) combine to form a single WDM driver that serves as the function driver for the I2C bus controller. Wdf01000.sys (the Framework) owns the dispatch table for this driver, so when someone sends an IRP to the driver triple, it goes to the wdf01000.sys. If the wdf01000.sys can handle the IRP by itself, SpbCx.sys and MyI2CBusDriver.sys are not involved. If wdf01000.sys cannot handle the IRP by itself, it gets help by calling an event handler in SbpCx.sys.

Here are some examples of event handlers that might be implemented by MyI2CBusDriver.sys:

- EvtSpbControllerLock
- EvtSpbIoRead
- EvtSpbIoSequence

Here are some examples of event handlers that are implemented by SpbCx.sys

- EvtIoRead

Upper and lower edges of drivers

6/25/2019 • 3 minutes to read • Edit Online

The sequence of drivers that participate in an I/O request is called the driver stack for the request. A driver can call into the upper edge of a lower driver in the stack. A driver can also call into the lower edge of a higher driver in the stack.

Before you read this topic, you should understand the ideas presented in Device nodes and device stacks and Driver stacks.

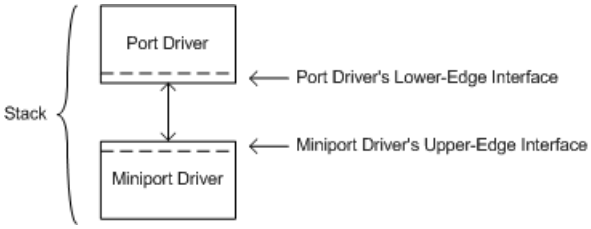
I/O requests are processed first by the top driver in the driver stack, then by the next lower driver, and so on until the request is fully processed.

When a driver implements a set of functions that a higher driver can call, that set of functions is called the *upper edge* of the driver or the *upper-edge interface* of the driver.

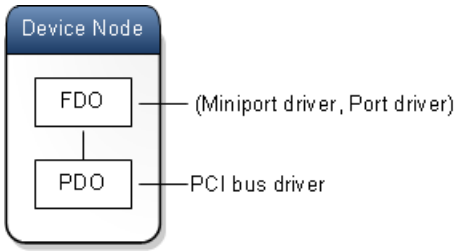
When a driver implements a set of functions that a lower driver can call, that set of functions is called the *lower edge* of the driver or the *lower-edge interface* of the driver.

Audio example

We can think of an audio miniport driver sitting below an audio port driver in a driver stack. The port driver makes calls to the miniport driver's upper edge. The miniport driver makes calls to the port driver's lower edge.

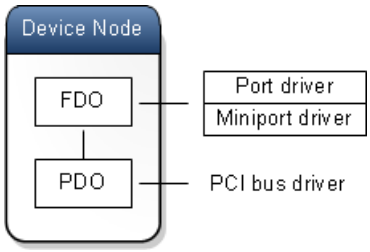


The preceding diagram illustrates that it is sometimes useful to think of a port driver sitting above a miniport driver in a driver stack. Because I/O requests are processed first by the port driver and then by the miniport driver, it is reasonable to think of the port driver as being above the miniport driver. Keep in mind, however, that a (miniport, port) driver pair usually sits at a single level in a device stack, as shown here.



Note that a *device stack* is not the same thing as a *driver stack*. For definitions of these terms, and for a discussion of how a pair of drivers can form a single WDM driver that occupies one level in a device stack, see Minidrivers and driver pairs.

Here's another way to draw a diagram of the same device node and device stack:

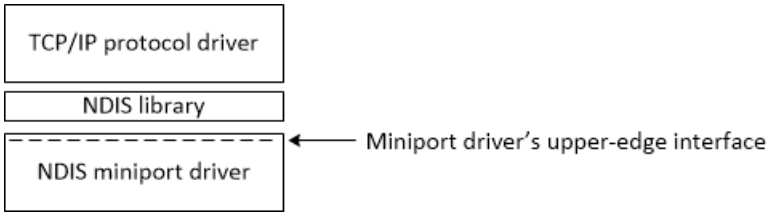


In the preceding diagram, we see that the (miniport, port) pair forms a single WDM driver that is associated with a single device object (the FDO) in the device stack; that is, the (miniport, port) pair occupies only one level in the device stack. But we also see a vertical relationship between the miniport and port drivers. The port driver is shown above the miniport driver to indicate that the port driver processes I/O requests first and then calls into the miniport driver for additional processing.

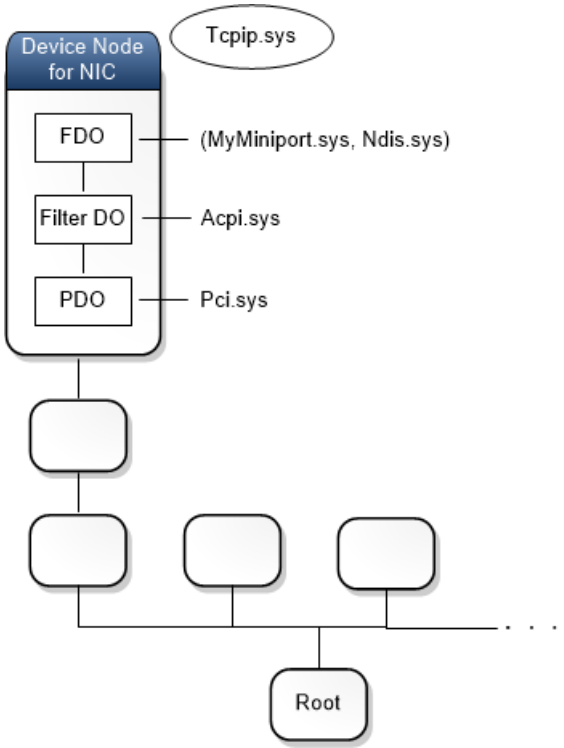
The key point is that when the port driver calls into the miniport driver's upper-edge interface, that is not the same as passing an I/O request down the device stack. In a driver stack (not device stack) you can choose to draw a port driver above a miniport driver, but that does not mean that the port driver is above the miniport driver in the device stack.

NDIS example

Sometimes a driver calls the upper edge of a lower driver indirectly. For example, suppose a TCP/IP protocol driver sits above an NDIS miniport driver in a driver stack. The miniport driver implements a set of *MiniportXxx* functions that form the miniport driver's upper edge. We say that the TCP/IP protocol driver *binds* to the upper edge of the NDIS miniport driver. But the TCP/IP driver does not call the *MiniportXxx* functions directly. Instead, it calls functions in the NDIS library, which then call the *MiniportXxx* functions.



The preceding diagram shows a driver stack. Here's another view of the same drivers.



The preceding diagram shows the device node for a network interface card (NIC). The device node has a position in

the Plug and Play (PnP) device tree. The device node for the NIC has a device stack with three device objects. Notice that the NDIS miniport driver and the NDIS library work as a pair. The pair (MyMiniport.sys, Ndis.sys) forms a single WDM driver that is represented by the functional device object (FDO).

Also notice that the protocol driver Tcpip.sys is not part of the device stack for the NIC. In fact, Tcpip.sys is not part of the PnP device tree at all.

Summary

The terms *upper edge* and *lower edge* are used to describe the interfaces that drivers in a stack use to communicate with each other. A *driver stack* is not the same thing as *device stack*. Two drivers that are shown vertically in a driver stack might form a driver pair that sits at a single level in a device stack. Some drivers are not part of the PnP device tree.

Related topics

Concepts for all driver developers

Device nodes and device stacks

Driver stacks

Audio Devices

Network Drivers Starting with Windows Vista

Header files in the Windows Driver Kit

10/23/2019 • 2 minutes to read • Edit Online

The Windows Driver Kit (WDK) contains all the header files (.h files) that you need to build kernel-mode and user-mode drivers. Header files are in the Include folder in your WDK installation folder. Example: C:\Program Files (x86)\Windows Kits\10\Include.

The header files contain version information so that you can use the same set of header files regardless of which version of Windows your driver will run on.

Constants that represent Windows versions

Header files in the WDK contain conditional statements that specify programming elements that are available only in certain versions of the Windows operating system. The versioned elements include functions, enumerations, structures, and structure members.

To specify the programming elements that are available in each operating system version, the header files contain preprocessor conditionals that compare the value of NTDDI_VERSION with a set of predefined constant values that are defined in Sdkddkver.h.

Here are the predefined constant values that represent versions of the Microsoft Windows operating system.

CONSTANT	OPERATING SYSTEM VERSION
NTDDI_WIN10	Windows 10
NTDDI_WINBLUE	Windows 8.1
NTDDI_WIN8	Windows 8
NTDDI_WIN7	Windows 7
NTDDI_WS08SP4	Windows Server 2008 with SP4
NTDDI_WS08SP3	Windows Server 2008 with SP3
NTDDI_WS08SP2	Windows Server 2008 with SP2
NTDDI_WS08	Windows Server 2008

You can see many examples of version-specific DDI elements in the WDK header files. This conditional declaration appears in Wdm.h, which is a header file that might be included by a kernel-mode driver.

```
#if (NTDDI_VERSION >= NTDDI_WIN7)
_Must_inspect_result_
NTKERNELAPI
NTSTATUS
KeSetTargetProcessorDpcEx (
    _Inout_ PKDPC Dpc,
    _In_ PPROCESSOR_NUMBER ProcNumber
    );
#endif
```

In the example you can see that the **KeSetTargetProcessorDpcEx** function is available only in Windows 7 and later versions of Windows.

This conditional declaration appears in Winspool.h, which is a header file that might be included by a user-mode driver.

```
#if (NTDDI_VERSION >= NTDDI_WIN7)
...
BOOL
WINAPI
GetPrintExecutionData(
    _Out_ PRINT_EXECUTION_DATA *pData
    );

#endif // (NTDDI_VERSION >= NTDDI_WIN7)
```

In the example can see that the **GetPrintExecutionData** function is available only in Windows 7 and later versions of Windows.

Header files for the Kernel Mode Driver Framework

The WDK supports several versions of Windows, and it also supports several versions of the Kernel Mode Driver Framework (KMDF) and User Mode Driver Framework (UMDF). The versioning information in the WDK header files pertains to Windows versions, but not to KMDF or UMDF versions. Header files for different versions of KMDF and UMDF are placed in separate directories.

Writing drivers for different versions of Windows

10/23/2019 • 4 minutes to read • Edit Online

When you create a driver project, you specify the minimum target operating system, which is the minimum version of Windows that your driver will run on. For example, you could specify that Windows 7 is the minimum target operating system. In that case, your driver would run on Windows 7 and later versions of Windows.

Note If you develop a driver for a particular minimum version of Windows and you want your driver to work on later versions of Windows, you must not use any undocumented functions, and you must not use documented functions in any way other than how it is described in the documentation. Otherwise your driver might fail to run on the later versions of Windows. Even if you have been careful to use only documented functions, you should test your driver on the new version of Windows each time one is released.

Writing a multiversion driver using only common features

When you design a driver that will run on multiple versions of Windows, the simplest approach is to allow the driver to use only DDI functions and structures that are common to all versions of Windows that the driver will run on. In this situation, you set the minimum target operating system to the earliest version of Windows that the driver will support.

For example, to support all versions of Windows, starting with Windows 7, you should:

1. Design and implement the driver so that it uses only those features that are present in Windows 7.
2. When you build your driver, specify Windows 7 as the minimum target operating system.

While this process is simple, it might restrict the driver to use only a subset of the functionality that is available on later versions of Windows.

Writing a multiversion driver that uses version-dependent features

A kernel-mode driver can dynamically determine which version of Windows it is running on and choose to use features that are available in that version. For example, a driver that must support all versions of Windows, starting with Windows 7, can determine, at run time, the version of Windows that it is running on. If the driver is running on Windows 7, it must use only the DDI functions that Windows 7 supports. However, the same driver can use additional DDI functions that are unique to Windows 8, for example, when its run-time check determines that it is running on Windows 8.

Determining the Windows version

RtlIsNtDdiVersionAvailable is a function that drivers can use to determine, at run time, if the features that are provided by a particular version of Windows are available. The prototype for this function is as follows:

```
BOOLEAN RtlIsNtDdiVersionAvailable(IN ULONG Version)
```

In this prototype, *Version* is a value that indicates the required version of the Windows DDI. This value must be one of the DDI version constants, defined in sdkddkver.h, such as NTDDI_WIN8 or NTDDI_WIN7.

RtlIsNtDdiVersionAvailable returns TRUE when the caller is running on a version of Windows that is the same as, or later than, the one that is specified by *Version.*

Your driver can also check for a specific service pack by calling the **RtlIsServicePackVersionInstalled** function. The prototype for this function is as follows:

```
BOOLEAN RtlIsServicePackVersionInstalled(IN ULONG Version)
```

In this prototype, *Version* is a value that indicates the required Windows version and service pack. This value must be one of the DDI version constants, defined in sdkddkver.h, such as NTDDI_WS08SP3.

Note that RtlIsServicePackVersionInstalled returns TRUE only when the operating system version exactly matches the specified version. Thus, a call to **RtlIsServicePackVersionInstalled** with *Version* set to NTDDI_WS08SP3 will fail if the driver is not running on Windows Server 2008 with SP4.

Conditionally calling Windows version-dependent functions

After a driver determines that a specific operating system version is available on the computer, the driver can use the MmGetSystemRoutineAddress function to dynamically locate the routine and call it through a pointer. This function is available in Windows 7 and later operating system versions.

Note To help preserve type checking and prevent unintentional errors, you should create a typedef that mirrors the original function type.

Example: Determining the Windows version and conditionally calling a version-dependent function

This code example, which is from a driver's header file, defines the PAISQSL type as a pointer to the KeAcquireInStackQueuedSpinLock function. The example then declares a `AcquireInStackQueuedSpinLock` variable with this type.

```
...
 //
// Pointer to the ordered spin lock function.
// This function is only available on Windows 7 and
// later systems
 typedef (* PAISQSL) (KeAcquireInStackQueuedSpinLock);
PAISQSL AcquireInStackQueued = NULL;
 ...
```

This code example, which is from the driver's initialization code, determines whether the driver is running on Windows 7 or a later operating system. If it is, the code retrieves a pointer to KeAcquireInStackQueuedSpinLock.

```
...

//
// Are we running on Windows 7 or later?
//
 if (RtlIsNtDdiVersionAvailable(NTDDI_WIN7) ) {

 //
  // Yes... Windows 7 or later it is!
  //
     RtlInitUnicodeString(&funcName,
                 L"KeAcquireInStackQueuedSpinLock");

 //
  // Get a pointer to Windows implementation
  // of KeAcquireInStackQueuedSpinLock into our
  // variable "AcquireInStackQueued"
     AcquireInStackQueued = (PAISQSL)
                 MmGetSystemRoutineAddress(&funcName);
 }

...
// Acquire a spin lock.

 if( NULL != AcquireInStackQueued) {
  (AcquireInStackQueued)(&SpinLock, &lockHandle);
} else {
    KeAcquireSpinLock(&SpinLock);
}
```

In the example the driver calls **RtlIsNtDdiVersionAvailable** to determine whether the driver is running on Windows 7 or later. If the version is Windows 7 or later, the driver calls **MmGetSystemRoutineAddress** to get a pointer to the **KeAcquireInStackQueuedSpinLock** function and stores this pointer in the variable named `AcquireInStackQueued` (which was declared as a PAISQSL type).

Later, when the driver must acquire a spin lock, it checks to see whether it has received a pointer to the **KeAcquireInStackQueuedSpinLock** function. If the driver has received this pointer, the driver uses the pointer to call **KeAcquireInStackQueuedSpinLock**. If the pointer to **KeAcquireInStackQueuedSpinLock** is null, the driver uses **KeAcquireSpinLock** to acquire the spin lock.