How to Use WinUSB to Communicate with a USB Device

March 30, 2009

Abstract

Independent hardware vendors (IHVs) who manufacture USB devices must often provide a way for applications to access the device's features. Historically, this has meant using the Windows® Driver Model (WDM) to implement a function driver for the device and installing the driver in the device stack above system-supplied protocol drivers. The Windows Driver Foundation (WDF) is now the preferred model for USB drivers. It provides IHVs with three options for providing access to a USB device:

- Implementing a user-mode driver by using the WDF user-mode driver framework (UMDF).
- Implementing a kernel-mode driver by using the WDF kernel-mode driver framework (KMDF).
- Installing WinUsb.sys as the device's function driver and providing an application that accesses the device by using the WinUSB API.

This white paper provides guidelines for when to use each option and includes a detailed walkthrough of how to install WinUsb.sys as a device's function driver and use the WinUSB API to communicate with the device.

This information applies for the following operating systems:

Windows 7
Windows Server® 2008
Windows Vista®
Windows XP

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

For the latest information, see:

http://www.microsoft.com/whdc/connect/usb/winusb_howto.mspx

Revision History

Date	Change
March 30, 2009	▪ Added additional information for communicating with endpoints.
December 6, 2007	▪ Added a section on DFU to the WinUSB FAQ. ▪ Removed the note from Table 1 indicating that WinUSB does not support WinUSB selective suspend on Windows XP.
August 30, 2007	Created

Contents

March 30, 2009

Introduction

Independent hardware vendors (IHVs) who manufacture USB devices must typically provide a way for applications to access the device's features. Historically, this has meant using the Windows® Driver Model (WDM) to implement a function driver for the device and installing the driver in the device stack above system-supplied protocol drivers such as Usbhub.sys or Usbccgp.sys. The function driver exposes a device interface that applications use to obtain the device's file handle. They can then use the handle to communicate with the driver by calling Windows API functions such as **ReadFile** and **DeviceIoControl**.

Drivers are the most flexible way to provide access to a USB device and allow the device to be accessed by any application, including multiple concurrent applications. However, drivers require a significant development effort, and some devices are simple enough that they do not require the full support of a custom function driver. For example, devices such as machine controllers or data loggers are typically accessed only by a single application that was written specifically for the associated device. In these cases, WinUSB provides a simpler alternative to implementing a custom USB driver.

WinUSB was developed concurrently with the Windows Driver Foundation (WDF) and is available for Windows XP and later versions of Windows. It includes a kernel-mode driver, WinUsb.sys, which is an integral part of WDF user-mode driver framework (UMDF) support for USB drivers. However, for USB devices that are accessed by only a single application, vendors can often install WinUsb.sys as their device's function driver instead of implementing a custom driver. The application can then configure the device and access its endpoints by using the WinUSB API.

For those USB devices that require the features of a custom function driver, the preferred approach is WDF. The WDF programming model and device driver interface (DDI) makes WDF USB drivers much easier to implement than equivalent WDM drivers. You can implement WDF USB drivers in either of the following ways:

- Use the user-mode driver framework (UMDF) to implement user-mode USB drivers for most USB devices for Windows XP and later.
- Use the kernel-mode driver framework (KMDF) to implement kernel-mode USB drivers for any USB device for Windows 2000 and later.

This white paper describes how to choose the best way to provide applications with access to a USB device and answers some common questions about WinUSB. The bulk of the paper is a detailed walkthrough—including code samples—of how to install WinUsb.sys as a USB device's function driver and how to use the WinUSB API to communicate with the device from an application.

The examples in this paper are based on the OSR USB FX2 Learning Kit device, but you can easily extend the procedures to other USB devices. Figure 1 is a simplified diagram of the FX2 device and shows its key features.

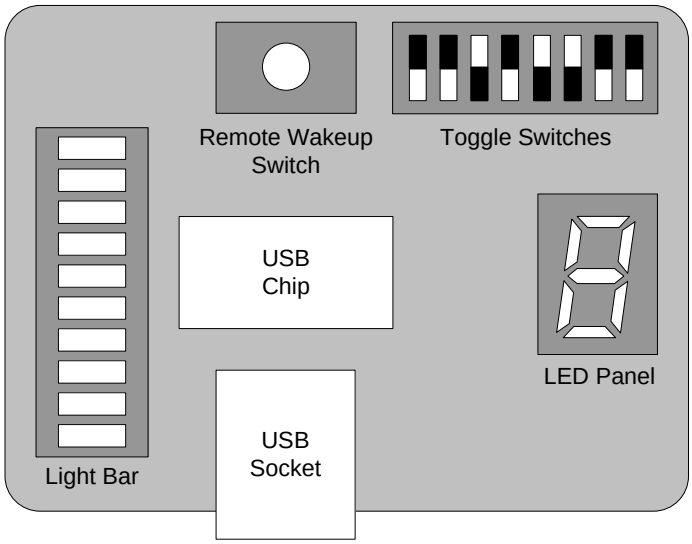

Figure 1. OSR USB FX2 Learning Kit device

Summary of WinUSB, UMDF, and KMDF Capabilities

Table 1 summarizes the capabilities of WinUSB, UMDF USB drivers, and KMDF USB drivers.

Table 1. WDF USB Feature Support

Requirements	WinUSB	UMDF	KMDF
Supports multiple concurrent applications.	No	Yes	Yes
Isolates driver address space from application address space.	No	Yes	No
Supports bulk, interrupt, and control transfers.	Yes	Yes	Yes
Supports isochronous transfers.	No	No	Yes
Supports the installation of kernel-mode drivers such as filter drivers above the USB driver.	No	No	Yes
Supports selective suspend and wait/wake	Yes	Yes	Yes

Table 2 summarizes which WDF options are supported by different versions of Windows.

Table 2. WDF USB Windows Support

Requirements	WinUSB	UMDF	KMDF
Windows Vista and later	Yes[1]	Yes[1]	Yes
Windows Server 2003	No	No	Yes
Windows XP	Yes[2]	Yes[2]	Yes
Windows 2000	No	No	Yes[3]

Notes:
[1]WinUSB and UMDF are supported only on x86 and x64 versions of Windows.
[2]WINUSB and UMDF are supported on Windows XP with service pack 2 or later.
[3]KMDF is supported on Windows 2000 with Service Pack 4 or later.

Guidelines for Providing Access to USB Devices

Generally, start with the simplest approach, WinUSB, and move to more complex solutions only if it is necessary.

If your device does not support isochronous transfers and is accessed by only a single application, the application can use WinUSB to configure the device and access its endpoints. For example, WinUSB is the preferred approach to use for an electronic weather station that is accessed only by an application that is packaged with the device. WinUSB is also useful for diagnostic communication with a device and for flashing firmware.

Some types of devices require a custom function driver. For example, you must implement a driver for devices that are accessed by multiple concurrent applications. The general guidelines for implementing WDF USB drivers are as follows:

- UMDF is the preferred approach and is suitable for most USB devices. For example, UMDF drivers are the preferred option for music players or serial dongles.
- Devices with features that are not supported by UMDF or that must run on versions of Windows earlier than Windows XP require a KMDF driver. For example, a USB network adapter requires a KMDF driver because the driver's upper edge must communicate with the kernel-mode network interface device standard (NDIS) stack. Such communication can be done only from kernel mode. KMDF drivers are also required for devices that support isochronous transfers.

For a detailed discussion about how to implement UMDF and KMDF drivers, see the Microsoft Press book *Developing Drivers with the Windows Driver Foundation* or the WHDC "Windows Driver Foundation" Web page. The rest of this white paper is devoted to WinUSB.

Introduction to WinUSB

WinUSB consists of two primary components:

- WinUsb.sys is a kernel-mode driver that can be installed as either a filter or function driver, above the protocol drivers in a USB device's kernel-mode device stack.
- WinUsb.dll is a user-mode DLL that exposes the WinUSB API. Applications can use this API to communicate with WinUsb.sys when it is installed as a device's function driver.

For devices that do not require a custom function driver, WinUsb.sys can be installed in the device's kernel-mode stack as the function driver. User-mode processes can then communicate with WinUsb.sys through a set of device I/O control requests.

The WinUSB API—exposed by WinUSB.dll—simplifies this communication process. Instead of constructing device I/O control requests to perform standard USB operations—such as configuring the device, sending control requests, and transferring data to or from the device—applications call equivalent WinUSB API functions. Internally, WinUsb.dll uses the data that the application passes to the WinUSB function to construct the appropriate device I/O control request and sends

the request to WinUsb.sys for processing. When the request is complete, the WinUSB function passes any information returned by WinUsb.sys—such as data from a read request—back to the calling process.

Using the WinUSB API to communicate with a device is much simpler than implementing a driver but has the following corresponding limitations:

- The WinUSB API lets only one application at a time communicate with the device. If more than one application must be able to communicate concurrently with a device, you must implement a function driver.
- The WinUSB API does not support streaming data to or from isochronous endpoints. Isochronous transfers require a kernel-mode function driver.
- The WinUSB API does not support devices that already have kernel-mode support. Examples of such devices include modems and network adaptors, which are supported by the telephony API (TAPI) and NDIS, respectively.
- For multifunction devices, you can use the device's INF to specify either an in-box kernel-mode driver or WinUsb.sys for each USB function separately. However, you can specify only one of these options for a particular function, not both.

WinUsb.sys is also a key part of the link between a UMDF function driver and the associated device. WinUsb.sys is installed in the device's kernel-mode stack as an upper filter driver. An application communicates with the device's UMDF function driver to issue read, write, or device I/O control requests. The driver interacts with the framework, which passes the request to WinUsb.sys, which processes the request and passes it to the protocol drivers and ultimately to the device. Any response returns by the reverse path. WinUsb.sys also serves as the device stack's Plug and Play and power owner (PPO).

WinUSB FAQ

This FAQ answers several common questions about WinUSB.

Which versions of Windows support WinUSB?

WinUSB is supported by:

- All Windows Vista SKUs.
- All client SKUs of the 32-bit versions of Windows XP SP2 and later service packs.

Note: WinUSB is not native to Windows XP; it must be installed with the WinUSB co-installer. For details, see "How to Install WinUsb.sys as a Function Driver" later in this paper.

Which USB features are supported by WinUSB?

Table 3 shows which high-level USB features are supported by WinUSB in Windows Vista and Windows XP.

Table 3. WinUSB Feature Support

Feature	Windows XP	Windows Vista
Device I/O control requests	Supported	Supported
Isochronous transfers	Not supported	Not supported
Bulk, control, and interrupt transfers	Supported	Supported
Selective suspend	Supported	Supported
Remote wake	Supported	Supported

How do I get permission to redistribute WinUSB?

WinUSB is included in the Windows Driver Kit (WDK) in the form of a co-installer package, WinUSBCoInstaller.dll. Separate DLLs for x86 and x64 systems are located under the WinDDK*BuildNumber*\Redist\Winusb folder. These DLLs are signed. IHVs can redistribute these DLLs.

Does WinUSB support the DFU profile for USB devices?

Microsoft has not implemented a specific driver for firmware updates. WinUSB does not support host-initiated reset port and cycle port operations. However, some devices expose a vendor-defined control code for initiating a reset. In that case, you can use WinUSB to initiate a reset by sending a control transfer request that contains the vendor-defined control code to the device. For a discussion of how to use WinUSB to send control transfer requests, see "Control Requests" later in this paper.

How do I report WinUSB bugs or make feature requests?

To report bugs or make feature requests, use http://support.microsoft.com to contact your Microsoft Technical Account Manager or Product Support Engineer.

How to Install WinUsb.sys as a Function Driver

Before your application can use the WinUSB API to communicate with a device, you must install WinUsb.sys as the device's function driver. To do so, create a package that includes the following:

- The WinUSB co-installer, which installs WinUSB on the target system, if necessary.

  The WDK includes three versions of the co-installer: one for x86 systems, one for x64 systems, and one for Itanium systems. They are all named WinUSBCoInstaller.dll and are located in the appropriate subdirectory of the WinDDK*BuildNumber*\redist\winusb folder.

- The KMDF co-installer, which installs the correct version of KMDF on the target system, if necessary.

  This co-installer is required because WinUsb.sys depends on KMDF. The version of WinUSB on which this paper is based depends on KMDF version 1.5, and the associated co-installer is WdfCoInstaller01005.dll. The x86 and x64 versions of WdfCoInstaller01005.dll are included with the WDK under the WinDDK*BuildNumber*\redist\wdf folder.

- An INF that installs WinUsb.sys as the device's function driver.
- A signed catalog file for the package.

  This file is required to install WinUSB on x64 versions of Windows Vista. For more information on how to create and test signed catalog files, see "Kernel-Mode Code Signing Walkthrough" on the WHDC Web site.

Note: As new WDF versions are released, the co-installer names will change to reflect the WDF version number. For example WDF version 1.7 is currently under development, and the KMDF co-installer for that version is named WdfCoInstaller01007.dll.

The following example is a simple INF that installs WinUsb.sys as the function driver for the Fx2 device:

```
 [Version]
Signature = "$Windows NT$"
Class = MyDeviceClass
ClassGuid={78A1C341-4539-11d3-B88D-00C04FAD5171}
Provider = %ProviderName%
CatalogFile=MyCatFile.cat

; ================== Class section ==================

[ClassInstall32]
Addreg=MyDeviceClassReg

[MyDeviceClassReg]
HKR,,,0,%ClassName%
HKR,,Icon,,-1

; ========== Manufacturer/Models sections ===========

[Manufacturer]
%ProviderName% = MyDevice_WinUSB,NTx86,NTamd64,NTia64

[MyDevice_WinUSB.NTx86]
%USB\MyDevice.DeviceDesc% =USB_Install, USB\VID_0547&PID_1002

[MyDevice_WinUSB.NTamd64]
%USB\MyDevice.DeviceDesc% =USB_Install, USB\VID_0547&PID_1002

[MyDevice_WinUSB.NTia64]
%USB\MyDevice.DeviceDesc% =USB_Install, USB\VID_0547&PID_1002
; ================== Installation ==================

;[1]
[USB_Install]
Include=winusb.inf
Needs=WINUSB.NT

;[2]
[USB_Install.Services]
Include=winusb.inf
AddService=WinUSB,0x00000002,WinUSB_ServiceInstall

;[3]
[WinUSB_ServiceInstall]
DisplayName     = %WinUSB_SvcDesc%
ServiceType     = 1
StartType       = 3
ErrorControl    = 1
ServiceBinary   = %12%\WinUSB.sys

;[4]
[USB_Install.Wdf]
KmdfService=WINUSB, WinUsb_Install
```

```
[WinUSB_Install]
KmdfLibraryVersion=1.5

;[5]
[USB_Install.HW]
AddReg=Dev_AddReg

[Dev_AddReg]
HKR,,DeviceInterfaceGUIDs,0x10000,"{b35924d6-3e16-4a9e-9782-
5524a4b79bac}"

;[6]
[USB_Install.CoInstallers]
AddReg=CoInstallers_AddReg
CopyFiles=CoInstallers_CopyFiles

[CoInstallers_AddReg]
HKR,,CoInstallers32,0x00010000,"WdfCoInstaller01005.dll,WdfCoInstaller
","WinUSBCoInstaller.dll"

[CoInstallers_CopyFiles]
WinUSBCoInstaller.dll
WdfCoInstaller01005.dll

[DestinationDirs]
CoInstallers_CopyFiles=11

; ================ Source Media Section =====================
;[7]

[SourceDisksNames]
1 = %DISK_NAME%,,\i386
2 = %DISK_NAME%,,\amd64
3 = %DISK_NAME%,,\ia64

[SourceDisksFiles.x86]
WinUSBCoInstaller.dll=1
WdfCoInstaller01005.dll=1

[SourceDisksFiles.NTamd64]
WinUSBCoInstaller.dll=2
WdfCoInstaller01005.dll=2

[SourceDisksFiles.ia64]
WinUSBCoInstaller.dll=3
WdfCoInstaller01005.dll=3
; ================== Strings ==================

[Strings]
ProviderName="MyWinUsbTest"
USB\MyDevice.DeviceDesc="Test using WinUSB only"
WinUSB_SvcDesc="WinUSB Test"
DISK_NAME="My Install Disk"
ClassName="MyDeviceClass"
```

This INF can be used for most USB devices, with some straightforward modifications. Generally, you should change “USB_Install” in section names to an appropriate *DDInstall* value. You should also make straightforward changes to the version, manufacturer, and model sections, such as providing an appropriate manufacture’s name, the name of your signed catalog file, the correct device class, and the vendor identifier (VID) and product identifier (PID) for the device.

The device-specific values that should be changed are shown in the example in bold. Values that might need to be changed, such as those that depend on version number, are in italic.

Note: For more information on USB device classes, see “Supported USB Classes” in the WDK.

Apart from device-specific values and several issues that are noted in the following list, you can use these sections and directives without modification to install WinUSB for any USB device. The following notes correspond to the numbered comments in the INF.

1. The **Include** and **Needs** directives in the **USB_Install** section are required for installing WinUSB on Windows Vista systems. Windows XP systems ignore these directives. These directives should not be modified.
2. The **Include** directive in the **USB_Install.Services** section includes the system-supplied INF for WinUSB. This INF is installed by the WinUSB co-installer if it is not already on the target system. The **AddService** directive specifies WinUsb.sys as the device's function driver. These directives should not be modified.
3. The **WinUSB_ServiceInstall** section contains the data for installing WinUsb.sys as a service. This section should not be modified.
4. The **KmdfService** directive in the **USB_Install.Wdf** section installs WinUsb.sys as a kernel-mode service. The referenced **WinUSB_Install** section specifies the KMDF library version. This example is based on the Windows Vista version of the WDK (build 6000), which includes KMDF version 1.5. Later versions of WinUSB might require a later KMDF version.
5. **USB_Install.HW** is the key section in the INF. It specifies the device interface globally unique identifier (GUID) for your device. The **AddReg** directive puts the interface GUID in a standard registry value. When WinUsb.sys is loaded as the device's function driver, it reads the registry value and uses the specified GUID to represent the device interface. You should replace the GUID in this example with one that you create specifically for your device. If the protocols for the device change, you should create a new device interface GUID.
6. The **USB_Install.CoInstallers** section, including the referenced **AddReg** and **CopyFiles** sections, contains data and instructions to install the WinUSB and KMDF co-installers and associate them with the device. Most USB devices can use these sections and directives without modification.
7. The x86 and x64 versions of Windows have separate co-installers. This example stores them on the installation disk in folders that are named i386 and amd64, respectively. Figure 2 (on the following page) shows an example of what an IHV's driver package might contain.

   Note: Each co-installer has free and checked versions. Use the free version to install WinUSB on free builds of Windows, including all retail versions. Use the checked version—which has the "_chk" suffix—to install WinUSB on checked builds of Windows.

The INF typically also contains directives to install the associated application.

You install WinUsb.sys exactly like any other driver. The simplest approach is to plug in the device and use the Add New Hardware Wizard or Device Manager to install the driver by using the INF that is discussed in this section. For more details on INFs and how to install device drivers, see "Device and Driver Installation" in the WDK.

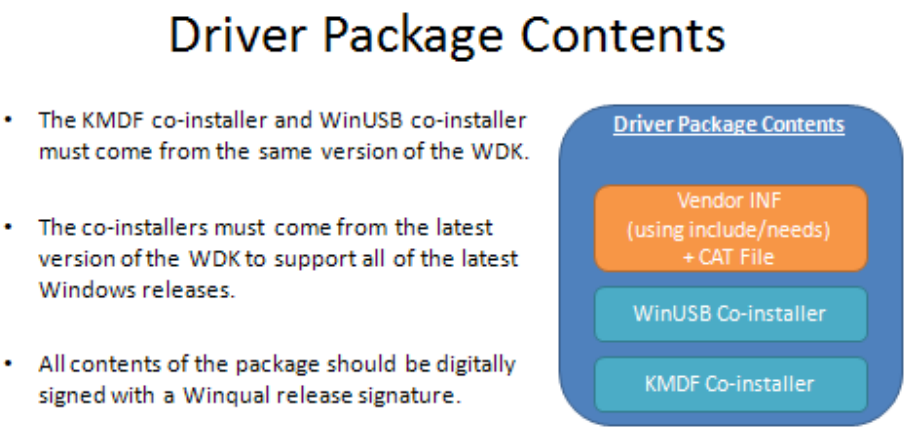

Figure 2. Example driver package contents

How to Use the WinUSB API

If a USB device has WinUsb.sys as its function driver, the associated application communicates with the device by calling various WinUSB API functions. To use the WinUSB API in an application:

- Include WinUsb.h. It is included with the WDK, under WINDDK*BuildNumber*\inc\ddk.
- Add WinUsb.lib to the list of libraries that are linked to your application. WinUsb.lib is included with the WDK. The version for Windows XP is located under WINDDK*BuildNumber*\lib\wxp\i386. There are separate versions of WinUsb.lib for Windows Vista for each supported CPU architecture. They are located under the WINDDK*BuildNumber*\lib\wlh folder.
- Include Usb100.h, which is also under WINDDK*BuildNumber*\inc\ddk. This header is not required, but it contains declarations for some useful macros.

To access the device, an application must:

1. Use the device interface GUID to obtain a handle to the device.
2. Use the handle to initialize WinUSB.
3. Use the WinUSB API to configure the device.
4. Use the WinUSB API to communicate with the endpoints.

This section shows how to perform these key tasks, based on a simple application that accesses the Fx2 device.

Obtain a Handle to the Device and Initialize WinUSB

To use the WinUSB API, you must first obtain a file handle for the device and use that handle to initialize WinUSB. The first two steps of the procedure are similar to those steps that are used to obtain a file handle for any device:

1. Use the device interface GUID to obtain the device path. The correct GUID is the one that you specified in the INF that was used to install WinUsb.sys.
2. Use the device path from step 1 to obtain a file handle for the device.

3. Pass the file handle to **WinUsb_Initialize** to initialize WinUSB and obtain a WinUSB handle. You use the device's WinUSB handle to identify the device when you call WinUSB API functions, not the device's file handle.

Obtain the Device Path

The application's GetDevicePath function, which is shown in the following example, uses the device interface GUID to obtain the device path. It is similar to the Setup API code that is used for most devices, but is included here for completeness. For more information, see "Setup API" on MSDN. Note that some code—mostly routine error-handling code—has been omitted for clarity:

```
BOOL GetDevicePath(LPGUID InterfaceGuid,
                   PCHAR DevicePath,
                   size_t BufLen)
{
  BOOL bResult = FALSE;
  HDEVINFO deviceInfo;
  SP_DEVICE_INTERFACE_DATA interfaceData;
  PSP_DEVICE_INTERFACE_DETAIL_DATA detailData = NULL;
  ULONG length;
  ULONG requiredLength=0;
  HRESULT hr;

// [1]
  deviceInfo = SetupDiGetClassDevs(InterfaceGuid,
                    NULL, NULL,
                    DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
...//Error handling code omitted.

// [2]
  interfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);

  bResult = SetupDiEnumDeviceInterfaces(deviceInfo,
                                        NULL,
                                        InterfaceGuid,
                                        0,
                                        &interfaceData);
...//Error handling code omitted.

// [3]
  SetupDiGetDeviceInterfaceDetail(deviceInfo,
                                  &interfaceData,
                                  NULL, 0,
                                  &requiredLength,
                                  NULL);

  detailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)
     LocalAlloc(LMEM_FIXED, requiredLength);

  if(NULL == detailData)
  {
    SetupDiDestroyDeviceInfoList(deviceInfo);
    return FALSE;
  }

  detailData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
  length = requiredLength;

  bResult = SetupDiGetDeviceInterfaceDetail(deviceInfo,
                                            &interfaceData,
                                            detailData,
                                            length,
                                            &requiredLength,
                                            NULL);
```

```
  if(FALSE == bResult)
  {
    LocalFree(detailData);
    return FALSE;
  }

// [4]
  hr = StringCchCopy(DevicePath,
                     BufLen,
                     detailData->DevicePath);
  if(FAILED(hr))
  {
    SetupDiDestroyDeviceInfoList(deviceInfo);
    LocalFree(detailData);
  }

  LocalFree(detailData);

  return bResult;
}
```

The basic procedure is as follows:

1. Get a handle to the device information set by passing the device interface GUID that you defined in the INF to **SetupDiGetClassDevs**. The function returns an HDEVINFO handle.
2. Call **SetupDiEnumDeviceInterfaces** to enumerate the system's device interfaces and obtain information on your device interface. To do so:
   - Initialize a SP_DEVICE_INTERFACE_DATA structure by setting its **cbSize** member to the size of the structure.
   - Pass the HDEVINFO handle from step 1, the device interface GUID, and a reference to the initialized SP_DEVICE_INTERFACE_DATA structure to **SetupDiEnumDeviceInterfaces**.
   - When the function returns, the SP_DEVICE_INTERFACE_DATA structure contains basic data for the interface.
3. Call **SetupDiGetDeviceInterfaceDetail** to get detailed data for the device interface. The information is returned in a SP_DEVICE_INTERFACE_DETAIL_DATA structure.
   - Because the size of the SP_DEVICE_INTERFACE_DETAIL_DATA structure varies, you must first obtain the correct buffer size by calling **SetupDiGetDeviceInterfaceDetail** with the *DeviceInterfaceDetailData* parameter set to NULL.
   - The function returns the correct buffer size in the *requiredlength* parameter. Use that value to correctly allocate memory for a SP_DEVICE_INTERFACE_DETAIL_DATA structure.
   - Call **SetupDiGetDeviceInterfaceDetail** again and pass it a reference to the initialized structure. When the function returns, the structure contains detailed information about the interface.
4. The device path is in the SP_DEVICE_INTERFACE_DETAIL_DATA structure's **DevicePath** member.

Obtain a File Handle for the Device

The application's OpenDevice function, which is shown in the following example, obtains a file handle for the device by passing the device path to **CreateFile**.

1. Call the GetDevicePath utility function to obtain the device path. GetDevicePath was discussed in the previous section.
2. Pass the device path to **CreateFile** to obtain a file handle for the device. This example obtains a file handle that supports synchronous read and write access to the device. For details on how to open a file handle for asynchronous I/O, see the "**CreateFile** Function" reference page on MSDN. Be sure to set the FILE_FLAG_OVERLAPPED flag. WinUSB depends on this setting:

```
HANDLE OpenDevice(BOOL bSync)
{
  HANDLE hDev = NULL;
  char devicePath[MAX_DEVPATH_LENGTH];
  BOOL retVal;

  retVal = GetDevicePath( (LPGUID) &GUID_DEVINTERFACE_OSRUSBFX2,
                          devicePath,
                          sizeof(deviceName));
...//Error-handling code omitted.

  hDev = CreateFile(devicePath,
                    GENERIC_WRITE | GENERIC_READ,
                    FILE_SHARE_WRITE | FILE_SHARE_READ,
                    NULL,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
                    NULL);

...//Error-handling code omitted.
  return hDev;
}
```

Initialize WinUSB

The handle that was obtained in the preceding section is the file handle for the device. However, the WinUSB API uses a WinUSB handle to identify the target device instead of the file handle. To obtain a WinUSB handle, initialize WinUSB by passing the file handle to **WinUsb_Initialize**. This function returns the WinUSB handle for the device.

The following example initializes WinUSB with the file handle that the OpenDevice utility function obtained, as discussed in the preceding section. It then stores the corresponding WinUSB handle for later use in a privately defined global structure:

```
deviceHandle = OpenDevice(TRUE);
bResult = WinUsb_Initialize(deviceHandle, &usbHandle);

if(bResult)
{
  devInfo.winUSBHandle = usbHandle;
}
```

Configure the Device

After an application initializes WinUSB, it must configure the USB device. The procedure is similar to the one that USB device drivers use. However, it is accomplished by calling WinUSB API functions instead of the WDF framework libraries or any of the Windows WDM USB client support routines.

The application's InitializeDevice function, which is shown in the following example, configures the Fx2 device. It also includes the code shown in the preceding example to initialize WinUSB:

```
BOOL Initialize_Device()
{
  BOOL bResult;
  WINUSB_INTERFACE_HANDLE usbHandle;
  USB_INTERFACE_DESCRIPTOR ifaceDescriptor;
  WINUSB_PIPE_INFORMATION pipeInfo;
  UCHAR speed;
  ULONG length;

  deviceHandle = OpenDevice(TRUE);
  bResult = WinUsb_Initialize(deviceHandle, &usbHandle);

//[1]
  if(bResult)
  {
    devInfo.winUSBHandle = usbHandle;
    length = sizeof(UCHAR);
    bResult = WinUsb_QueryDeviceInformation(devInfo.winUSBHandle,
                                            DEVICE_SPEED,
                                            &length,
                                            &speed);
  }

//[2]
  if(bResult)
  {
    devInfo.deviceSpeed = speed;
    bResult = WinUsb_QueryInterfaceSettings(devInfo.winUSBHandle,
                                            0,
                                            &ifaceDescriptor);
  }
  if(bResult)
  {
    for(int i=0;i<ifaceDescriptor.bNumEndpoints;i++)
    {
//[3]
      bResult = WinUsb_QueryPipe(devInfo.winUSBHandle,
                                 0,
                                 (UCHAR) i,
                                 &pipeInfo);
//[4]
      if(pipeInfo.PipeType == UsbdPipeTypeBulk &&
                 USB_ENDPOINT_DIRECTION_IN(pipeInfo.PipeId))
      {
        devInfo.bulkInPipe = pipeInfo.PipeId;
      }
      else if(pipeInfo.PipeType == UsbdPipeTypeBulk &&
                 USB_ENDPOINT_DIRECTION_OUT(pipeInfo.PipeId))
      {
        devInfo.bulkOutPipe = pipeInfo.PipeId;
      }
      else if(pipeInfo.PipeType == UsbdPipeTypeInterrupt)
      {
        devInfo.interruptPipe = pipeInfo.PipeId;
      }
      else
      {
        bResult = FALSE;
        break;
      }
    }
  }

  return bResult;
}
```

The basic procedure is as follows:

1. If necessary, call **WinUsb_QueryDeviceInformation** to obtain the device's speed. The function returns one of three values: **LowSpeed** (0x01), **FullSpeed** (0x02), or **HighSpeed** (0x03).
2. Pass the device's interface handles to **WinUsb_QueryInterfaceSettings** to obtain the corresponding interface descriptors. The WinUSB handle corresponds to the first interface. Because the Fx2 device supports only one interface that has no alternative settings, the *AlternateSettingNumber* parameter is set to zero and the function is called only once. If the device supports multiple interfaces, call **WinUsb_GetAssociatedInterface** to obtain interface handles for associated interfaces.

   WinUsb_QueryInterfaceSettings returns a USB_INTERFACE_DESCRIPTOR structure that contains information about the interface. The structure contains, in particular, the number of endpoints in the interface.
3. To obtain information about each endpoint, call **WinUsb_QueryPipe** once for each endpoint on each interface. **WinUsb_QueryPipe** returns a WINUSB_PIPE_INFORMATION structure that contains information about the specified endpoint. The Fx2 device has one interface with three endpoints, so the function's *AlternateInterfaceNumber* parameter is set to 0 and the value of the *PipeIndex* parameter is varied from 0 to 2.
4. To determine the pipe type, examine the WINUSB_PIPE_INFORMATION structure's **PipeInfo** member. This member is set to one of the USBD_PIPE_TYPE enumeration values: **UsbdPipeTypeControl**, **UsbdPipeTypeIsochronous**, **UsbdPipeTypeBulk**, or **UsbdPipeTypeInterrupt**.

   The three endpoints that the Fx2 device supports are an interrupt pipe, a bulk-in pipe, and a bulk-out pipe, so **PipeInfo** will be set to either **UsbdPipeTypeInterrupt** or **UsbdPipeTypeBulk**.

   The **UsbdPipeTypeBulk** value identifies bulk pipes, but does not give the direction. That information is encoded in the high bit of the pipe address, which is stored in the WINUSB_PIPE_INFORMATION structure's **PipeId** member. The simplest way to determine direction is to pass the **PipeId** value to one of the following macros from Usb100.h:

   - USB_ENDPOINT_DIRECTION_IN returns TRUE if the direction is in.
   - USB_ENDPOINT_DIRECTION_OUT returns TRUE if the direction is out.

   The application uses the **PipeId** value to identify the desired pipe in calls to WinUSB functions such as **WinUsb_ReadPipe**, so the example stores all three **PipeId** values for later use.

Communicate with Endpoints

After you have **PipeId** values for the device's endpoints, you can communicate with the device. This paper discusses how to issue control, read, and write requests.

Control Requests

All USB devices have a control endpoint in addition to the endpoints that are associated with interfaces. The primary purpose of the control endpoint is to provide

a default endpoint that applications can use to configure the device. However, devices can also use the control endpoint for device-specific purposes. The Fx2 device uses the control endpoint to control the light bar and seven-segment digital display.

Control commands consist of an 8-byte setup packet—which includes a request code that specifies the particular request—and an optional data buffer. The request codes and buffer formats are vendor defined. To issue a control request:

1. Allocate a buffer, if necessary.
2. Construct a setup packet.
3. Call **WinUsb_ControlTransfer** to send the request and buffer to the control endpoint.

The applications SetBar function, which is shown in the following example, sends a control request to the Fx2 device to control the lights on the light bar:

```
BOOL SetBar(HWND hWnd)
{
  BOOL bResult;
  ULONG bytesReturned;
  WINUSB_SETUP_PACKET setupPacket;
  UCHAR lightedBars = 0;

//[1]
  numBars = 8;
  for(int i = 0;i<numBars; i++)
  {
    if((int) SendMessage(hwndSetBarCheckBox[i], BM_GETCHECK, 0,0))
    {
      lightedBars += 1 << (UCHAR) i;
    }
  }

//[2]
  setupPacket.RequestType = 0; //Host to Device
  setupPacket.Request = SET_BARGRAPH_DISPLAY;
  setupPacket.Index = 0;
  setupPacket.Length = sizeof(UCHAR);
  setupPacket.Value = 0;

//[3]
  bResult = WinUsb_ControlTransfer(devInfo.winUSBHandle,
                                   setupPacket,
                                   &lightedBars,
                                   sizeof(UCHAR),
                                   &bytesReturned,
                                   NULL);
  return bResult;
}
```

The code to set the light bar is 0xD8, which is defined for convenience as SET_BARGRAPH_DISPLAY. The procedure for issuing the request is as follows:

1. Load the data into the buffer. The device requires a 1-byte data buffer that specifies which elements should be lit by setting the appropriate bits. The user interface (UI) for the application includes a set of eight CheckBox controls that are used to specify which elements of the light bar should be lit. The example queries the user interface to determine which lights should be lit and sets the appropriate bits in the buffer.

2. Construct a setup packet by assigning values to a WINUSB_SETUP_PACKET structure:
   - The **RequestType** member specifies request direction. It is set to 0, which indicates host-to-device data transfer. For device-to-host transfers, set **RequestType** to 1.
   - The **Request** member is set to the vendor-defined code for this request, 0xD8. It is defined for convenience as SET_BARGRAPH_DISPLAY.
   - The **Length** member is set to the size of the data buffer.

   The **Index** and **Value** members are not required for this request, so they are set to zero.
3. Transmit the request by passing the device's WinUSB handle, the setup packet, and the data buffer to **WinUsb_ControlTransfer**. The function returns the number of bytes that were transferred to the device.

WinUSB I/O Requests

The Fx2 device has bulk-in and bulk-out endpoints that can be used for read and write requests, respectively. These two endpoints are configured for loopback, so the device simply moves data from the bulk-in endpoint to the bulk-out endpoint. It does not change the value of the data or create any new data, so a read request reads the data that was sent by the most recent write request.

WinUSB Write Requests

WinUSB has separate functions for sending write and read requests: **WinUsb_WritePipe** and **WinUsb_ReadPipe**. The application's WriteToDevice function, which is shown in the following example, writes a simple string to the Fx2 device:

```
BOOL WriteToDevice(HWND hWnd)
{
  USHORT bufSize = 12;
  UCHAR szBuffer[12];
  BOOL bResult;
  ULONG bytesWritten;

//[1]
  SendMessage(hwndWriteEdit, EM_GETLINE, 0, (LPARAM) szBuffer);

//[2]
  bResult = WinUsb_WritePipe(devInfo.winUSBHandle,
                             devInfo.bulkOutPipe,
                             szBuffer,
                             24,
                             &bytesWritten,
                             NULL);

  return bResult;
}
```

To send a write request:

1. Create a buffer and fill it with the data that you want to write to the device. The sample obtains a 12-character Unicode string from an Edit control on the application's UI.

   The application is responsible for managing the size of the buffer appropriately. As long as an application is not using RAW_IO, there is no limitation on buffer

size. WinUSB divides the buffer into appropriately sized chunks, if necessary. For RAW_IO, the size of the buffer is limited by the controller. For read requests, the buffer must be a multiple of the maximum packet size.

2. Write the buffer to the device by calling **WinUsb_WritePipe**. Pass the interface handle, the **PipeId** value for the bulk-out pipe, and the buffer. In this case, the interface handle is the WinUSB handle. The function returns the number of bytes that are actually written to the device in the *bytesWritten* parameter.

   The final parameter of **WinUsb_WritePipe**, *Overlapped*, is set to NULL to request a synchronous operation. To perform an asynchronous write, *Overlapped* should be a pointer to an OVERLAPPED structure.

Default WinUSB Write Behavior

Zero-length writes are forwarded down the stack. If the transfer length is greater than a maximum transfer length, WinUSB divides the request into smaller requests of maximum transfer length and submits them serially.

WinUSB Read Requests

Read operations are similar to write operations. Pass to **WinUsb_ReadPipe** the interface handle, the **PipeId** value for the bulk-in endpoint, and an appropriately sized empty buffer. When the function returns, the buffer contains the data that was read from the device. The number of bytes that were read is returned in the function's *bytesRead* parameter. The following example reads a character string from the Fx2 device and displays it in a Static control on the application's UI:

```
BOOL ReadFromDevice(HWND hWnd)
{
  USHORT bufSize = 12;
  UCHAR szBuffer[12];
  BOOL bResult;
  ULONG bytesRead;

  bResult = WinUsb_ReadPipe(devInfo.winUSBHandle,
                            devInfo.bulkInPipe,
                            szBuffer,
                            24,
                            &bytesRead,
                            NULL);

  SendMessage(hwndReadStatic, WM_SETTEXT, 0, (LPARAM) szBuffer);

  return bResult;
}
```

Default WinUSB Read Behavior

Zero-length reads complete immediately with success and are not sent down the stack. If the transfer length is greater than a maximum transfer length, WinUSB divides the request into smaller requests of maximum transfer length and submits them serially. If the transfer length is not a multiple of the endpoint's **MaxPacketSize**, WinUSB increases the size of the transfer to the next multiple of **MaxPacketSize**. If a device returns more data than was requested, WinUSB saves the excess data. If data remains from a previous read, WinUSB copies it to the beginning of the next read and completes the read, if necessary.

Pipe Policies

WinUSB allows modifications to its default behavior through policies that can be applied to a pipe (endpoint). Using policies can help IHVs tune WinUSB to best match their device to its capabilities. Table 4 provides a list of the pipe policies that WinUSB supports.

Table 4. WinUSB Pipe Policies

Policy number	Policy name	Default
0x01	SHORT_PACKET_TERMINATE	Off
0x02	AUTO_CLEAR_STALL	Off
0x03	PIPE_TRANSFER_TIMEOUT	5 seconds for control, 0 for others
0x04	IGNORE_SHORT_PACKETS	Off
0x05	ALLOW_PARTIAL_READS	On
0x06	AUTO_FLUSH	Off
0x07	RAW_IO	Off
0x09	RESET_PIPE_ON_RESUME	Off

Table 5 provides information about when you should use each of the pipe policies and describes the resulting behavior when each policy is enabled.

Table 5. Pipe Policy Behavior

Policy	When to use	Behavior
SHORT_PACKET_TERMINATE	▪ Use if your device requires that OUT transfers be terminated with a short packet (most devices do not require this). ▪ This policy is valid only for bulk and interrupt OUT endpoints (setting this policy on IN endpoints has no effect).	▪ If enabled, all writes to the endpoint are terminated with a short packet.
AUTO_CLEAR_STALL	▪ Use if you want failed transfers not to leave the endpoint in a stalled state. ▪ This policy is valid only for bulk and interrupt IN endpoints. ▪ This policy is generally useful only when you pend multiple reads to the endpoint when RAW_IO is disabled.	▪ If enabled, when an IN transfer fails and returns a status other than STATUS_CANCELLED or STATUS_DEVICE_NOT_CONNECTED, WinUSB resets the endpoint before completing the failed transfer. ▪ If disabled, subsequent transfers to the endpoint fail until the endpoint is manually reset by calling WinUsb_ResetPipe(). ▪ No significant performance difference exists between enabling or disabling this policy.

Policy	When to use	Behavior
PIPE_TRANSFER_TIMEOUT	▪ Use if you have an endpoint for which you expect transfers to complete within a specific time.	▪ If set to 0, transfers will pend indefinitely until they are manually canceled or they are completed normally. ▪ If set to a nonzero value, a timer starts when the request is sent down to the core USB stack (requests do not time out while in a WinUSB queue). When the timer expires, the request is canceled. ▪ There is a minor performance penalty because of managing timers.
IGNORE_SHORT_PACKETS	▪ Use if you do not want short packets to complete your read requests (this is rare). ▪ This policy is valid only for bulk and interrupt IN endpoints with RAW_IO disabled.	▪ Reads complete only if an error occurs, the request is canceled, or all the requested bytes have been received.
ALLOW_PARTIAL_READS	▪ Use if your device can legally send more data than was requested (possible only if you are requesting nonmultiples of **MaxPacketSize**). ▪ Use if the application wants to read a few bytes to determine how many total bytes to read. ▪ This policy is valid only for bulk and interrupt IN endpoints.	▪ If disabled and the device returns more data than was requested, the request completes with an error. ▪ If enabled, WinUSB immediately completes read requests for zero bytes with success and does not send the requests down the stack. ▪ If enabled and the device returns more data than was requested, WinUSB saves the excess data and prepends it to the next read request.
AUTO_FLUSH	▪ Use if your device can send more data than was requested and the application does not require any extra data beyond what was requested (possible only if requesting non-multiples of **MaxPacketSize**). ▪ This policy is valid only for bulk and interrupt IN endpoints where the ALLOW_PARTIAL _READS policy is enabled.	▪ If an endpoint returns more bytes than requested, WinUSB discards those bytes without error. ▪ Has no effect if ALLOW_PARTIAL_READS is disabled.

Policy	When to use	Behavior
RAW_IO	▪ Use if read performance is a priority and the application submits multiple simultaneous reads to the same endpoint. ▪ This policy is valid only for bulk and interrupt IN endpoints (setting the policy on OUT endpoints has no effect).	▪ Requests to the endpoint that are not a multiple of **MaxPacketSize** fail. ▪ Requests to the endpoint that are greater than the maximum transfer size (which is retrievable through the MAXIMUM_TRANSFER_SIZE pipe policy) fail. ▪ All well-formed reads to the endpoint are immediately sent down the stack to be scheduled in the host controller. ▪ Significant performance improvement occurs when you submit multiple read requests. ▪ The delay between the last packet of one transfer and the first packet of the next transfer is reduced.
RESET_PIPE_ON_RESUME	▪ Use if the device does not preserve its data toggle state across suspend. ▪ This policy is valid only for bulk and interrupt endpoints.	▪ On resume from suspend, WinUSB resets the endpoint before it lets requests be sent to the endpoint.

For more information about WinUSB policies, see "**WinUsb_GetPipePolicy**" and "**WinUsb_SetPipePolicy**" in the WDK.

WinUSB Power Management

WinUSB uses the KMDF state machines for power management. Power policy is managed through calls to **WinUsb_SetPowerPolicy**, as well as defaults that are set in the registry. To allow a device to wake the system, you must add the **SystemWakeEnabled** DWORD registry value and set it to a nonzero value. A check box in the device **Properties** page is automatically enabled so that the user can override the setting.

Selective Suspend

Selective suspend can be disabled by any of several system or WinUSB settings. A single setting cannot force WinUSB to enable selective suspend. The following Power Policy settings affect the behavior of selective suspend:

AUTO_SUSPEND
: When set to zero, does not selectively suspend the device.

SUSPEND_DELAY
: Sets the time between the device becoming idle and WinUSB requesting the device to selective suspend.

The following registry keys affect the behavior of selective suspend:

DeviceIdleEnabled
: When set to zero, does not selectively suspend the device.

DeviceIdleIgnoreWakeEnable
: When set to a nonzero value, suspends the device even if it does not support **RemoteWake**.

UserSetDeviceIdleEnabled
When set to a nonzero value, enables a check box in the device **Properties** page that allows the user to override the idle defaults.

DefaultIdleState
Sets the default value of the AUTO_SUSPEND power policy setting. This registry key is used to enable or disable selective suspend when a handle is not open to the device.

DefaultIdleTimeout
Sets the default state of the SUSPEND_DELAY power policy setting.

Detecting Idle

All writes and control transfers force the device into the D0 power state and reset the idle timer. The IN endpoint queues are not power managed. Reads wake the device when they are submitted. However, a device can become idle while a read request pends.

Future Feature Considerations

Please submit your requests for additional extensions and features to WinUSBFB@microsoft.com. Some possible future features include the following:

- Isochronous endpoint support.
- USB3 feature extensions.

Summary

By using WinUSB, IHVs provide a solid Windows driver solution for their USB hardware devices. WinUSB supports Windows XP and later versions of Windows and supports both 32-bit and 64-bit versions of Windows. Existing Windows 32-bit and 64-bit applications can be easily modified to take advantage of the WinUSB API. WinUSB eliminates all driver-related issues and lets IHVs provide a Windows driver solution for their USB devices in much less time and with a lot less effort than writing their own Windows driver.

Resources

Windows Hardware Developer Central (WHDC):

Kernel-Mode Code Signing Walkthrough
http://www.microsoft.com/whdc/winlogo/drvsign/kmcs_walkthrough.mspx

Windows Driver Foundation (WDF)
http://www.microsoft.com/whdc/driver/wdf/default.mspx

Writing USB Drivers with WDF
http://www.microsoft.com/whdc/driver/wdf/USB_WDF.mspx

Windows Driver Kit:

Device and Driver Installation
http://go.microsoft.com/fwlink/?LinkId=98295

March 30, 2009

Device and Driver Installation Design Guide
http://go.microsoft.com/fwlink/?LinkId=146855

Finish-Install Actions (Windows Vista and later)
http://go.microsoft.com/fwlink/?LinkId=146856

Finish-Install Wizard Pages
http://go.microsoft.com/fwlink/?LinkId=146857

Supported USB Classes
http://go.microsoft.com/fwlink/?LinkId=98297

USB
http://go.microsoft.com/fwlink/?LinkId=98298

WinUSB
http://go.microsoft.com/fwlink/?LinkId=98299

WinUSB User-Mode Client Support Routines
http://go.microsoft.com/fwlink/?LinkId=98300

WinUsb_GetPipePolicy
http://go.microsoft.com/fwlink/?LinkId=146858

WinUsb_SetPipePolicy
http://go.microsoft.com/fwlink/?LinkId=146859

MSDN:

CreateFile Function
http://go.microsoft.com/fwlink/?LinkId=98294

Setup API
http://go.microsoft.com/fwlink/?LinkId=98296

Microsoft Press:

Developing Drivers with the Windows Driver Foundation
http://www.microsoft.com/MSPress/books/10512.aspx