

Windows Driver Design Device Interfaces

主講人：虞台文

Content

- ≈ Overview
- ≈ Device Names
- ≈ Introduction to GUIDs
- ≈ Device Setup Classes
- ≈ Device Interface Classes
- ≈ Functions of Device Interface Class
- ≈ User-Mode Device Interface Functions

- ≈ Example

Windows Driver Design Device Interfaces

Overview

Access a Device

- ≈ Any driver of a physical, logical, or virtual device to which **user-mode** code can **direct I/O requests** must supply some sort of **name** for its user-mode clients.

- ≈ Using the **name**, a user-mode application (or other system component) identifies the device from which it is requesting I/O.

Register Device Interface

```
NTSTATUS XxxAddDevice(  
    IN PDRIVER_OBJECT  DriverObject,  
    IN PDEVICE_OBJECT  PhysicalDeviceObject )  
{  
    1. Call IoCreateDevice to create an FDO or FIDO for the device being added.  
    2. Create one or more symbolic links (device interfaces) to the device.  
    3. Initialize your device extension and the Flags member of the device object.  
    4. Call IoAttachDeviceToDeviceStack to Attach the device object to the device stack.  
    5. Clear the DO_DEVICE_INITIALIZING flag in the FDO or filter DO.  
}
```

Named Device Objects

- ≈ A **device object**, like all Object Manager objects, can be **named** or **unnamed**.

- ≈ When a user-mode application makes an I/O request, it specifies the **target** of the operation **by name**,
- e.g., the **filename** parameter of **CreateFile** API.

- ≈ The **Object Manager** **resolves** the **name** to determine the destination of the I/O request.

IoCreateDevice Review

```

NTSTATUS IoCreateDevice(
    IN PDRIVER_OBJECT DriverObject,
    IN ULONG DeviceExtensionSize,
    IN PUNICODE_STRING DeviceName OPTIONAL,
    IN DEVICE_TYPE DeviceType,
    IN ULONG DeviceCharacteristics,
    IN BOOLEAN Exclusive,
    OUT PDEVICE_OBJECT *DeviceObject
);
    
```

Symbolic Links

⚡ In **Windows NT® 4.0** and earlier versions of the NT-based operating system, drivers **named** their device objects and then set up **symbolic links** in the **registry** between **device names** and a user-visible Win32® logical name.

Symbolic Links

- ⚡ A driver can specify a **name** of a device object by calling **IoCreateDevice** or **IoCreateDeviceSecure** to create the device object.
- ⚡ A named device object can also have an **MS-DOS** device name, which is a **symbolic link** created by **IoCreateSymbolicLink** or **IoCreateSymbolicLinkUnprotected**.
- ⚡ **WDM** drivers do **not** in general require an MS-DOS device name.

Callers' IRQL = **PASSIVE_LEVEL**

IoCreateSymbolicLink Routine

```

NTSTATUS IoCreateSymbolicLink(
    IN PUNICODE_STRING SymbolicLinkName,
    IN PUNICODE_STRING DeviceName
);
    
```

- ⚡ Sets up a **symbolic link** between a **device object name** and a **user-visible name** for the device.
- ⚡ **PnP** drivers do **not** name device objects and therefore should not use this routine.
- ⚡ Instead, a **PnP** driver should call **IoRegisterDeviceInterface** to set up a **symbolic link**.

IoCreate

Callers' IRQL = **PASSIVE_LEVEL**

Points to a buffered Unicode string that is the **user-visible name**.

In the form of **\\DosDevices\\SymbolicLinkName**

```

NTSTATUS IoCreateSymbolicLink(
    IN PUNICODE_STRING SymbolicLinkName,
    IN PUNICODE_STRING DeviceName
);
    
```

Points to a buffered Unicode string that is the **name of the driver-created device object**.

In the form of **\\Device\\DeviceName**

Example

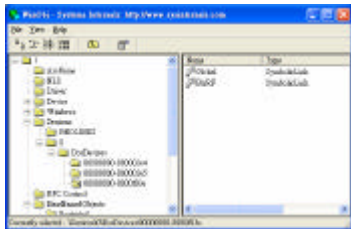
```

NTSTATUS IoCreateSymbolicLink(
    IN PUNICODE_STRING SymbolicLinkName,
    IN PUNICODE_STRING DeviceName
);
    
```

```

UNICODE_STRING linkname;
UNICODE_STRING targname;
RtlInitUnicodeString(&linkname, L"\\DosDevices\\barf");
RtlInitUnicodeString(&targname, L"\\Device\\Beep");
IoCreateSymbolicLink(&linkname, &targname);
    
```


The Result



Exercise

- Write an Windows application based on Win32 device management functions.
 - You can write one similar to WinObj with search capability.
- Write slides to describe the basic method to use the Win32 device management functions.
 - It is better to give some simple examples.

Windows Driver Design Device Interfaces

Introduction to GUIDs

Example:

6B29FC40-CA47-1067-B31D-00DD010662DA

GUIDs

- Globally Unique IDentifier.
- It is a **128-bit** unique identification string used to identify various class objects.
- The valid format for a GUID is
{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}
where X is a hex digit (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F).

```
typedef struct _GUID {  
    DWORD Data1;  
    WORD Data2;  
    WORD Data3;  
    BYTE Data4[8];  
} GUID;
```

Tools

UuidGen.exe

GuidGen.exe

```
// {92E88AF3-E426-41c8-AC0F-B120CA71089D}  
DEFINE_GUID(<name>,  
0x92e88af3, 0xe426, 0x41c8, 0xac, 0xf, 0xb1, 0x20, 0xca, 0x71, 0x8, 0x9d);
```

Including GUIDs in Driver Code

- Include the **initguid.h** header file that **redefines** the **DEFINE_GUID** macro.
- Include this header file in the driver source file **where** the GUIDs should be **instantiated**.
- User-mode** applications include **objbase.h** before including header files containing GUID definitions.

Including GUIDs in Driver Code

```

:
// include system headers here such as wdm.h

#include <initguid.h>

// include system and driver-specific header files here
// that contain GUID definitions

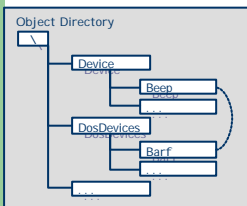
#include <guids>
:
    
```

Put the above statements **in one module** of the driver; typically the main module. When the above statements are present, the driver refers to a GUID using its symbolic name.

Windows Driver Design Device Interfaces

Device Names

Device Names



- ⚡ NT Device Names
 - **\Device\DeviceName.**
 - Known as the **NT device name** of the device object.
- ⚡ MS-Dos Device Names
 - **\DosDevices\DosDeviceName.**
 - A named device object created by a **non-WDM driver** typically has an MS-DOS device name.
 - It is a **symbolic link** in the Object Manager, e.g., **\DosDevices\COM1,** **\DosDevices\C:**

Device Names for WDM Drivers

- ⚡ WDM drivers do **not** name their device objects directly.

The system uses a **uniform naming scheme** as follows:

- ⚡ The **PDO** for a device is **named.**
 - The **bus driver requests named PDOs** for the devices it enumerates.
 - The bus driver uses **FILE_AUTOGENERATED_DEVICE_NAME** device characteristic when it creates the device object.
 - The system then **automatically generates** the device name.
- ⚡ The **FDOs** and **filter DOs** are **not named.**
 - **Function** and **filter** drivers do **not** request a name when creating the device object

Device Names for WDM Drivers

- ⚡ Any **I/O request** to a named device object automatically goes to the **top** object in that device object's **stack.**
 - Thus, **only** the **PDO** is required to be **named.**
 - **User-mode** applications do **not** refer to WDM device objects by **name;**
 - Instead, applications access the device object through its **device interface.**
- ⚡ Driver writers **must not** name more than one object in a device stack.
 - The OS checks **security settings** based on the named object.
 - If **two** different objects are named and have different security descriptors, the I/O requests **that are sent to the object with the weaker security descriptor** can reach the device object with the **stronger security descriptor.**

Device Names for non-WDM Drivers

- ⚡ A **non-WDM** driver must **explicitly** specify a **name** for any named device objects.
- ⚡ The driver must create **at least one named device object** in the **\Device** object directory to receive I/O requests.
- ⚡ The driver specifies the device name as the **DeviceName** parameter to **IoCreateDeviceSecure** when creating the device object.

Windows Driver Design Device Interfaces

Device Classes

Device Classes

- ⌘ Two device classes are used to place devices and drivers into groups whose members have similar characteristics:
- ⌘ Device Setup Classes
 - Used to group together devices that are installed and configured in a similar manner.
- ⌘ Device Interface Classes
 - Used to group together devices that provide similar capabilities.

Windows Driver Design Device Interfaces

Device Setup Classes

Device Setup Classes

- ⌘ To facilitate device installation
 - devices that are set up and configured in the same way are grouped into a device setup class.
- ⌘ A device setup class defines the class installer and class co-installers involved in installing the device.
- ⌘ For example, SCSI media changer devices are grouped into the MediumChanger device setup class.

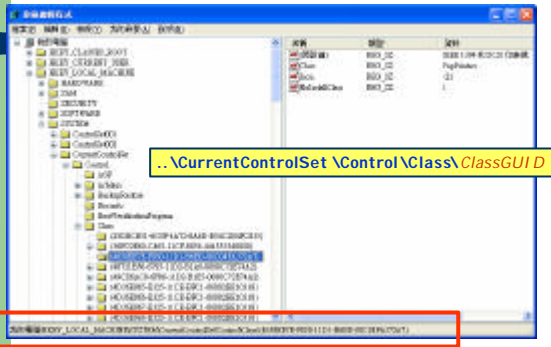
Who Defines the Device Setup Classes

- ⌘ Microsoft defines setup classes for most devices.
- ⌘ IHVs and OEMs can define new device setup classes, but only if none of the existing classes apply.
- ⌘ For example, a camera vendor doesn't need to define a new setup class because cameras fall under the Image setup class.
- ⌘ Similarly, uninterruptible power supply (UPS) devices fall under the Battery class.

GUIDs of Device Setup Classes

- ⌘ A GUID is associated with each device setup class.
- ⌘ System-defined setup class GUIDs are defined in devguid.h
- ⌘ Typically, as GUID_DEVCLASS_XXX.

Registry Keys for Device Setup Classes



Creating a New Device Setup Class?

- ⚡ Create one *only if* absolutely necessary.
 - It is usually possible to assign your device to one of the system-supplied device setup classes.
- ⚡ Use the system-supplied one if
 - the device's **installation and configuration requirements** match those of an existing class.
 - Your device's **capabilities** match those of an existing class.

Creating a New Device Setup Class?

- ⚡ Consider providing a device **co-installer** if
 - the device has installation requirements that are **not** supported by an existing device type-specific INF file.
 - Installation of your device requires device **property pages** that are **not** provided by an existing class.

Creating a New Device Setup Class?

- ⚡ If the device's capabilities are **significantly different** from those provided by existing classes, it might merit a new device setup class.
- ⚡ Before creating, contact **Microsoft** to find out if a new system-supplied device setup class is **being planned** for your device type.

How to Create a Device Setup Class?

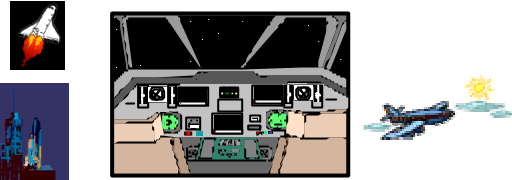
- ⚡ Create a new device setup class using an **INF** file.
- ⚡ Besides installing support for a device,
 - an INF file can initialize a new device setup class for the device.
 - such an INF file has an INF ClassInstall32 section.

Windows Driver Design Device Interfaces

Device Interface Classes

Interfaces to Access Device Objects

- Device interface classes are the means by which drivers make devices available to applications and other drivers.



Naming the Device Objects

- To allow user-mode applications or other system components to identify the device for direct I/O requests, a driver must supply some sort of name for its user-mode clients.
- Methods to export device name:
 - Windows NT® 4.0: via symbolic links
 - Windows® 2000 and later: via device interface classes

Device Interface Classes

- Exporting device and driver functionality to
 - user-mode applications
 - other system components, including other drivers
- Two steps to exporting a device interface:
A driver can
 - register a device interface class, then
 - enable an instance of the class for each device object to which user-mode I/O requests might be sent.

GUIDs of the Device Interface Classes

- Each device interface class is associated with a GUID.
- The system defines GUIDs for common device interface classes in device-specific header files.
- You can create additional device interface classes if needed.

Windows Driver Design Device Interfaces

Functions of
Device Interface Class

I/O Manager Routines

- `IoRegisterDeviceInterface`
- `IoSetDeviceInterfaceState`
- `IoOpenDeviceInterfaceRegistryKey`
- `IoGetDeviceInterfaces`
- `IoGetDeviceObjectPointer`

Callers must be running at IRLQ **PASSIVE_LEVEL** in the context of a **system thread**.

Register a Device Interface Class

```
NTSTATUS IoRegisterDeviceInterface(
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN CONST GUID *InterfaceClassGuid,
    IN PUNICODE_STRING ReferenceString OPTIONAL,
    OUT PUNICODE_STRING SymbolicLinkName
);
```

Registers a device interface class, if it has not been previously registered, and **creates a new instance** of the interface class.
A driver can subsequently **enable for use** by applications or other **system components**.

Callers must be running at IRLQ **PASSIVE_LEVEL** in the context of a **system thread**.

Register a Device Interface Class

```
NTSTATUS IoRegisterDeviceInterface(
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN CONST GUID *InterfaceClassGuid,
    IN PUNICODE_STRING ReferenceString OPTIONAL,
    OUT PUNICODE_STRING SymbolicLinkName
);
```

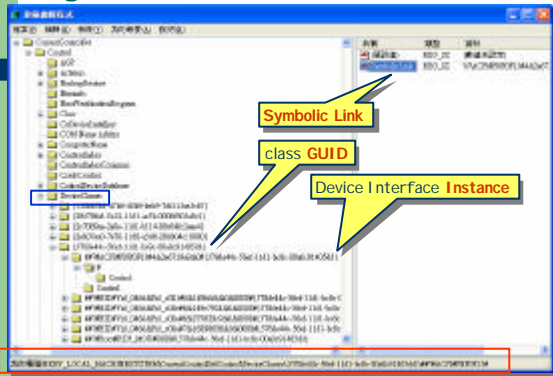
Points to the PDO for the device.

Points to the class GUID that identifies the functionality (the device interface class) being registered.

Optionally points to a reference string. Function drivers typically specify **NULL**. Filter drivers must specify **NULL**.

Points to a unicode string structure allocated by the caller. The caller is responsible to free it with **RtlFreeUnicodeString** when it is no longer needed.

Register a Device Interface Class



Callers must be running at IRLQ **PASSIVE_LEVEL** in the context of a **system thread**.

Enable/Disable a Device Interface Instance

```
NTSTATUS IoSetDeviceInterfaceState(
    IN PUNICODE_STRING SymbolicLinkName,
    IN BOOLEAN Enable
);
```

Enables or **disables** an instance of a previously registered device interface class.

Callers must be running at IRLQ **PASSIVE_LEVEL** in the context of a **system thread**.

Enable/Disable a Device Interface Instance

```
NTSTATUS IoSetDeviceInterfaceState(
    IN PUNICODE_STRING SymbolicLinkName,
    IN BOOLEAN Enable
);
```

TRUE/FALSE

Points to a string identifying the device interface instance being enabled or disabled. This string was obtained from a previous call to **IoRegisterDeviceInterface** or **IoGetDeviceInterfaces**.

Callers must be running at IRLQ **PASSIVE_LEVEL** in the context of a **system thread**.

Open Device Interface Registry Key

```
NTSTATUS IoOpenDeviceInterfaceRegistryKey(
    IN PUNICODE_STRING SymbolicLinkName,
    IN ACCESS_MASK DesiredAccess,
    OUT PHANDLE DeviceInterfaceKey
);
```

Returns a **handle** to a **registry key** for accessing information about a particular device interface instance.

Callers must be running at IRQL **PASSIVE_LEVEL** in the context of a **system thread**.

OpenDeviceInterfaceRegistryKey Points to a string identifying the device interface instance being enabled or disabled. This string was obtained from a previous call to `IoRegisterDeviceInterface` or `IoGetDeviceInterfaces`.

```
NTSTATUS IoOpenDeviceInterfaceRegistryKey(  
    IN PUNICODE_STRING SymbolicLinkName,  
    IN ACCESS_MASK DesiredAccess,  
    OUT PHANDLE DeviceInterfaceHandle  
);
```

Points to a **returned handle** to the requested registry key if the call is successful.

KEY_READ, KEY_WRITE, or KEY_ALL_ACCESS

Callers must be running at IRQL **PASSIVE_LEVEL** in the context of a **system thread**.

Get Device Interfaces

```
NTSTATUS IoGetDeviceInterfaces(  
    IN CONST GUID *InterfaceClassGuid,  
    IN PDEVICE_OBJECT PhysicalDeviceObject OPTIONAL,  
    IN ULONG Flags,  
    OUT PWSTR *SymbolicLinkList  
);
```

Returns a **list** of **device interface instances** of a **particular device interface class**, such as all devices on the system that support a HID interface.

Callers must be running at IRQL **PASSIVE_LEVEL** in the context of a **system thread**.

Get Device Interfaces

Points to the **class GUID** that identifies the functionality (the device interface class) being registered.

```
NTSTATUS IoGetDeviceInterfaces(  
    IN CONST GUID *InterfaceClassGuid,  
    IN PDEVICE_OBJECT PhysicalDeviceObject OPTIONAL,  
    IN ULONG Flags,  
    OUT PWSTR *SymbolicLinkList  
);
```

Points to an optional **PDO** that narrows the search to only the device interface instances.

Callers must be running at IRQL **PASSIVE_LEVEL** in the context of a **system thread**.

Get Device Interfaces

DEVICE_INTERFACE_INCLUDE_NONACTIVE: Return both **disabled** and **enabled** device interface instances.

```
NTSTATUS IoGetDeviceInterfaces(  
    IN CONST GUID *InterfaceClassGuid,  
    IN PDEVICE_OBJECT PhysicalDeviceObject OPTIONAL,  
    IN ULONG Flags,  
    OUT PWSTR *SymbolicLinkList  
);
```

Return a **list** of **symbolic link names**. Each unicode string in the list is **null-terminated**. The **end** of the whole list is marked by an **additional NULL**. The caller is responsible for **freeing** the buffer (`ExFreePool`) when it is no longer needed.

Callers' IRQL = **PASSIVE_LEVEL**

Get Device Object Pointer

```
NTSTATUS IoGetDeviceObjectPointer(  
    IN PUNICODE_STRING ObjectName,  
    IN ACCESS_MASK DesiredAccess,  
    OUT PFILE_OBJECT *FileObject,  
    OUT PDEVICE_OBJECT *DeviceObject  
);
```

Returns a **pointer** to a **named device object** and corresponding **file object**, if the requested access to the objects can be granted.

Windows Driver Design Device Interfaces

User-Mode Device
Interface Functions

Win32 Device Management

- When an application must **communicate** with a device, it searches for a device that exports the required **interface**.
- Use the following steps to perform the search:
 - Step 1.** Call the **SetupDiGetClassDevs** or **SetupDiGetClassDevsEx** function to obtain a list of all devices in a specified device class.
 - Step 2.** Call the **SetupDiEnumDeviceInterfaces** function to enumerate all devices of the specified class that export the interface.
 - Step 3.** Call the **SetupDiGetDeviceInterfaceDetail** function to get device information.
 - Step 4.** Call the **SetupDiDestroyDeviceInfoList** function to free memory.

Get Device Information of a Specific Class

```
HDEVINFO SetupDiGetClassDevs(
    IN LPGUID ClassGuid, OPTIONAL
    IN PCTSTR Enumerator, OPTIONAL
    IN HWND hwndParent, OPTIONAL
    IN DWORD Flags
);
```

Retrieves a **device information set** that contains **all** devices of a **specified class**.

Return value is a **handle** to a **device information set** containing all installed devices matching the specified parameters.

Its meaning depends on **Flags**.

```
HDEVINFO SetupDiGetClassDevs(
    IN LPGUID ClassGuid, OPTIONAL
    IN PCTSTR Enumerator, OPTIONAL
    IN HWND hwndParent, OPTIONAL
    IN DWORD Flags
);
```

The handle of the **top-level window** to be used for any user interface relating to the members of this set.

Optionally points to a string that **filters** the devices to be returned. Its meaning also depends on **Flags**.

Get Device Information of a Specific Class

Value	Meaning
DIGCF_ALLCLASSES	Return a list of installed devices for all classes. If set , ClassGuid is ignored .
DIGCF_DEVICEINTERFACE	If set , ClassGuid specifies interface class ; otherwise, it specifies a setup class .
DIGCF_PRESENT	Return only devices that are currently present .
DIGCF_PROFILE	Return only devices that are a part of the current hardware profile .

```
IN DWORD Flags
);
```

Get Device Information of a Specific Class

```
HDEVINFO SetupDiGetClassDevs(
    IN LPGUID ClassGuid, OPTIONAL
    IN PCTSTR Enumerator, OPTIONAL
    IN HWND hwndParent, OPTIONAL
    IN DWORD Flags
);
```

The caller of this function must **delete** the returned **device information set** when it is no longer needed by calling **SetupDiDestroyDeviceInfoList**.

Destroy Device Information Set

```
WINSETUPAPI BOOL WINAPI
SetupDiDestroyDeviceInfoList(
    IN HDEVINFO DeviceInfoSet
);
```

Destroys a device information set and frees all associated memory.

Enumerate Device Interfaces

```
WINSETUPAPI BOOL WINAPI SetupDiEnumDeviceInterfaces(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVINFO_DATA DeviceInfoData, OPTIONAL
    IN LPGUID InterfaceClassGuid,
    IN DWORD MemberIndex,
    OUT PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData
);
```

- Returns a context structure for a device **interface element** of a device **information set**.
- Each call returns information about **one** device interface.
- Can be called repeatedly** to get information about several interfaces exposed by one or more devices.

Enumerate Device Interfaces

```
WINSETUPAPI BOOL WINAPI SetupDiEnumDeviceInterfaces(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVINFO_DATA DeviceInfoData, OPTIONAL
    IN LPGUID InterfaceClassGuid,
    IN DWORD MemberIndex,
    OUT PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData
);
```

Typically, this handle is returned by **SetupDiGetClassDevs**.

Optionally points to an **SP_DEVINFO_DATA** structure that **constrains the search** for interfaces to those of just one device in the device information set. This pointer is typically returned by **SetupDiEnumDeviceInfo**.

Enumerate Device Interfaces

```
WINSETUPAPI BOOL WINAPI SetupDiEnumDeviceInterfaces(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVINFO_DATA DeviceInfoData, OPTIONAL
    IN LPGUID InterfaceClassGuid,
    IN DWORD MemberIndex,
    OUT PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData
);
```

Points to a **GUID** that specifies the **device interface class**.

Specifies a **zero-based index** into the list of interfaces in the device information set.

Enumerate Device Interfaces

```
WINSETUPAPI BOOL WINAPI SetupDiEnumDeviceInterfaces(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVINFO_DATA DeviceInfoData, OPTIONAL
    IN LPGUID InterfaceClassGuid,
    IN DWORD MemberIndex,
    OUT PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData
);
```

Points to a **caller-allocated buffer** that contains a completed **SP_DEVICE_INTERFACE_DATA** structure that identifies an interface that meets the search parameters. The caller must set **DeviceInterfaceData.cbSize** to **sizeof(SP_DEVICE_INTERFACE_DATA)** before calling this function.

Get Device Interface Detail

```
WINSETUPAPI BOOL WINAPI SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData, OPTIONAL
    IN DWORD DeviceInterfaceDetailDataSize,
    OUT PDWORD RequiredSize, OPTIONAL
    OUT PSP_DEVINFO_DATA DeviceInfoData OPTIONAL
);
```

Returns **details** about a particular device interface.

Get Device Interface Detail

```
WINSETUPAPI BOOL WINAPI SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData, OPTIONAL
    IN DWORD DeviceInterfaceDetailDataSize,
    OUT PDWORD RequiredSize, OPTIONAL
    OUT PSP_DEVINFO_DATA DeviceInfoData OPTIONAL
);
```

Typically, this handle is returned by **SetupDiGetClassDevs**.

Pointer to a **SP_DEVICE_INTERFACE_DATA** structure that identifies the interface, typically returned by **SetupDiEnumDeviceInterfaces**.

Get

Optionally points to a **caller-allocated** **SP_DEVICE_INTERFACE_DETAIL_DATA** structure to receive information about the specified interface.

```
WINSETUPAPI BOOL WINAPI SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData, OPTIONAL
    IN DWORD DeviceInterfaceDetailDataSize,
    OUT PDWORD RequiredSize, OPTIONAL
    OUT PSP_DEVINFO_DATA DeviceInfoData OPTIONAL
);
```

Specifies the **size** of the **DeviceInterfaceDetailData** buffer. The buffer must be at least **(offsetof(SP_DEVICE_INTERFACE_DETAIL_DATA, DevicePath) + sizeof(TCHAR))** bytes, to contain the fixed part of the structure and a single N NULL to terminate an empty MULTI_SZ string. This parameter must be **zero** if **DeviceInterfaceDetailData** is **NULL**.

Optionally points to a **caller-allocated** variable to receive the required size of the **DeviceInterfaceDetailData** buffer.

Get Device Interface Detail

```
WINSETUPAPI BOOL WINAPI SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData, OPTIONAL
    IN DWORD DeviceInterfaceDetailDataSize,
    OUT PDWORD RequiredSize, OPTIONAL
    OUT PSP_DEVINFO_DATA DeviceInfoData OPTIONAL
);
```

Optionally points to a **caller-allocated** buffer to receive information about the device that exposes the requested interface. The caller must set **DeviceInfoData.cbSize** to **sizeof(SP_DEVINFO_DATA)**.

Get Device Interface Detail

```
WINSETUPAPI BOOL WINAPI SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData, OPTIONAL
    IN DWORD DeviceInterfaceDetailDataSize,
    OUT PDWORD RequiredSize, OPTIONAL
    OUT PSP_DEVINFO_DATA DeviceInfoData OPTIONAL
);
```

Calling this function is typically in **two** steps:

Step 1. Get the required buffer size.

Call **SetupDiGetDeviceInterfaceDetail** with a **NULL** **DeviceInterfaceDetailData** pointer, an **DeviceInterfaceDetailDataSize** of **zero**, and a valid **RequiredSize** variable. In response, this function returns the required buffer size at **RequiredSize** and fails with **GetLastError** returning **ERROR_INSUFFICIENT_BUFFER**.

Step 2. Allocate an appropriately sized buffer and **call the function** again to get the interface details.

Get Device Interface Detail

```
WINSETUPAPI BOOL WINAPI SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData, OPTIONAL
    IN DWORD DeviceInterfaceDetailDataSize,
    OUT PDWORD RequiredSize, OPTIONAL
    OUT PSP_DEVINFO_DATA DeviceInfoData OPTIONAL
);
```

```
typedef struct _SP_DEVICE_INTERFACE_DETAIL_DATA {
    DWORD cbSize;
    TCHAR DevicePath[ANYSIZE_ARRAY];
} SP_DEVICE_INTERFACE_DETAIL_DATA;
```

Get Device Interface Detail

```
WINSETUPAPI BOOL WINAPI SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData, OPTIONAL
    IN DWORD DeviceInterfaceDetailDataSize,
    OUT PDWORD RequiredSize, OPTIONAL
    OUT PSP_DEVINFO_DATA DeviceInfoData OPTIONAL
);
```

Call **CreateFile** using this as the filename.

```
typedef struct _SP_DEVICE_INTERFACE_DETAIL_DATA {
    DWORD cbSize;
    TCHAR DevicePath[ANYSIZE_ARRAY];
} SP_DEVICE_INTERFACE_DETAIL_DATA;
```

Windows Driver Design Device Interfaces

Example

Demonstrations



Device Management