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

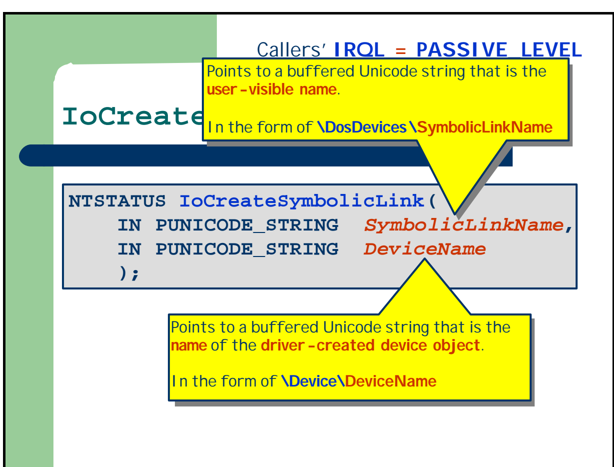
- A driver can specify a **name** of a device object by calling **IoCreateDevice** or **IoCreateDeviceSecure** to create the device object.
- A named device object can also have an **MS-DOS** device name, which is a **symbolic link** created by **IoCreateSymbolicLink** or **IoCreateSymbolicLinkUnprotected**.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

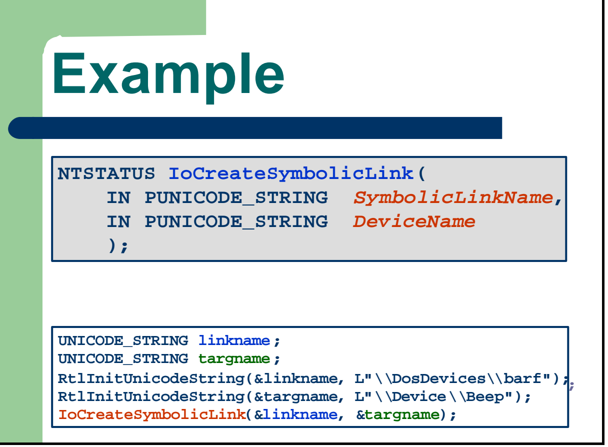
Callers' **IRQL = PASSIVE_LEVEL**

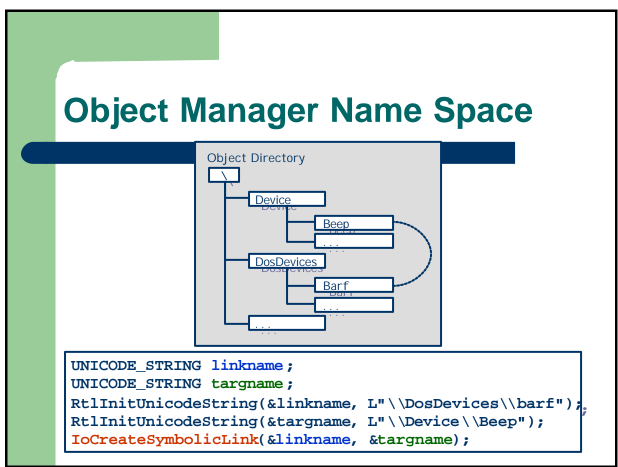
IoCreateSymbolicLink Routine

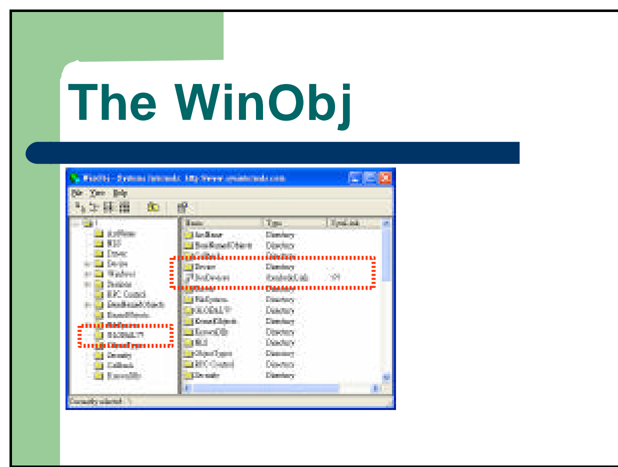
```
NTSTATUS IoCreateSymbolicLink(
    IN PUNICODE_STRING  SymbolicLinkName,
    IN PUNICODE_STRING  DeviceName
    );
```

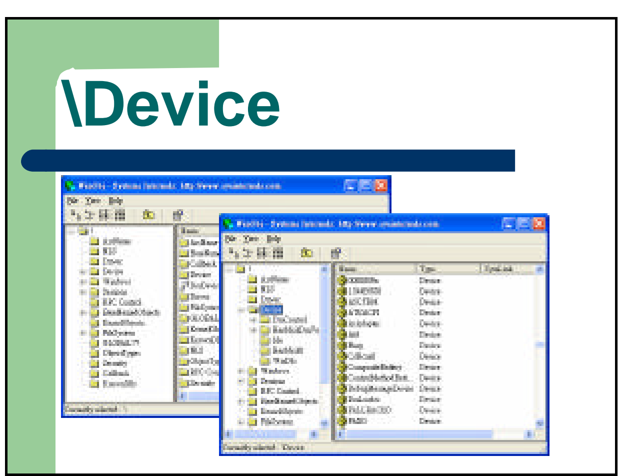
- Sets up a **symbolic link** between a **device object name** and a **user-visible name** for the device.
- **PnP** drivers do **not** name device objects and therefore should not use this routine.
- Instead, a **PnP** driver should call **IoRegisterDeviceInterface** to set up a **symbolic link**.

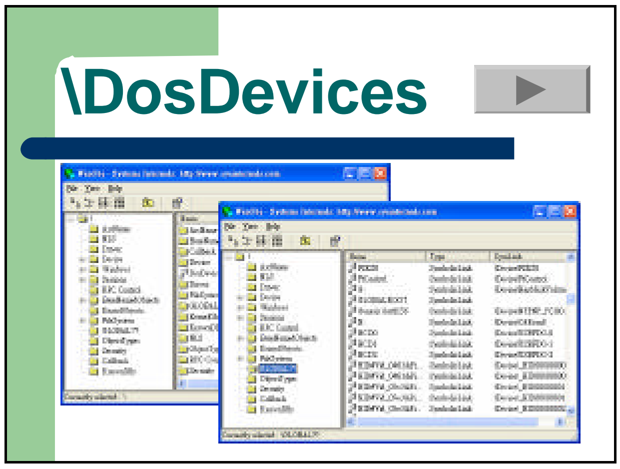


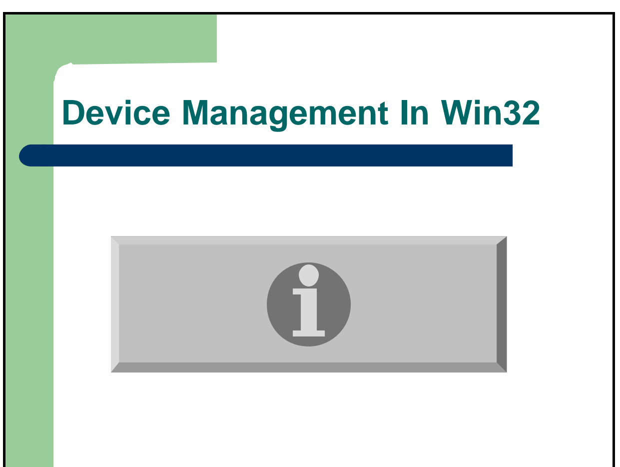






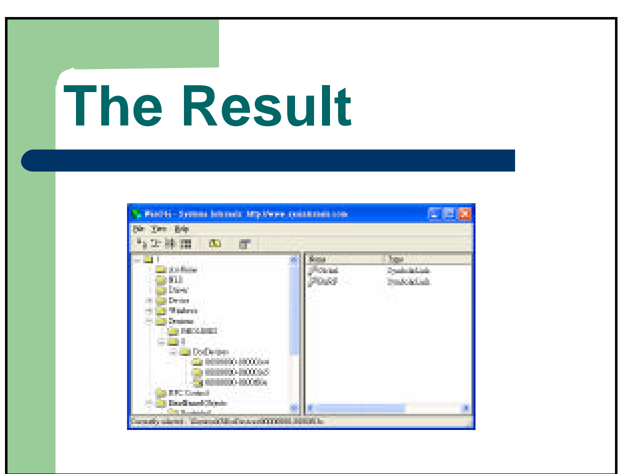






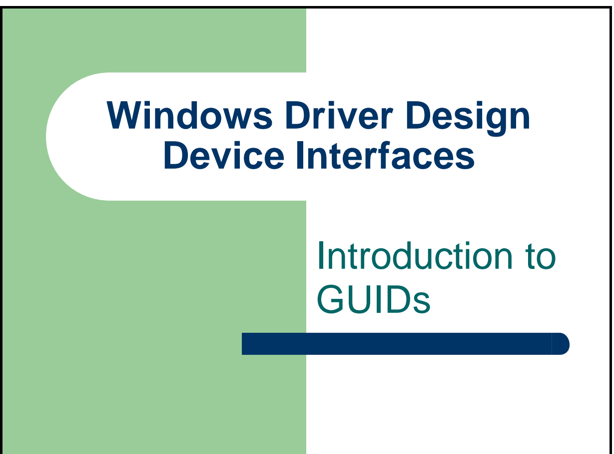
DefineDosDevice

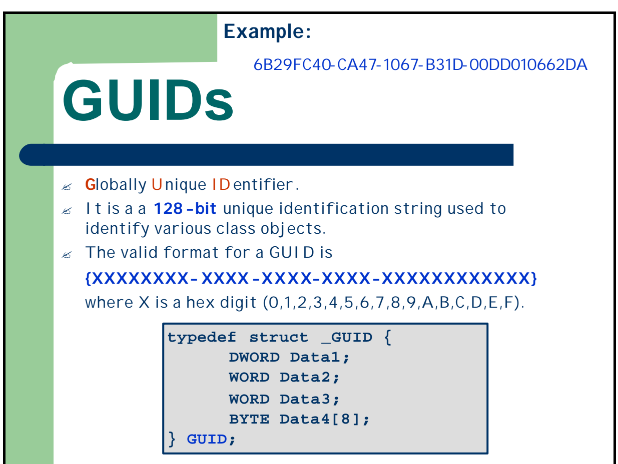
```
#include "stdafx.h"
#include <Windows.h>

int main(int argc, char* argv[])
{
  BOOL okay = DefineDosDevice(
            DDD_RAW_TARGET_PATH,
            "barf",
            "\\Device\\Beep");
  printf("\\DosDevice\\barf -> \\Device\\Beep\n");
  return 0;
}
```

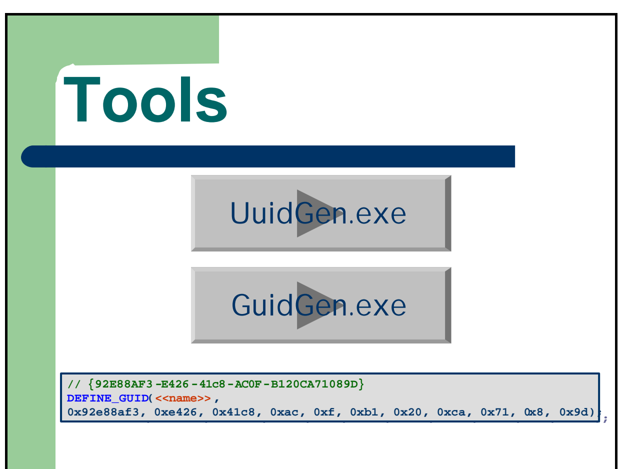


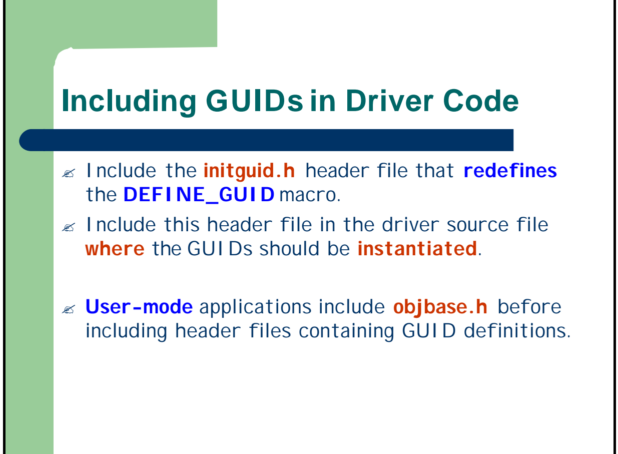
Exercise

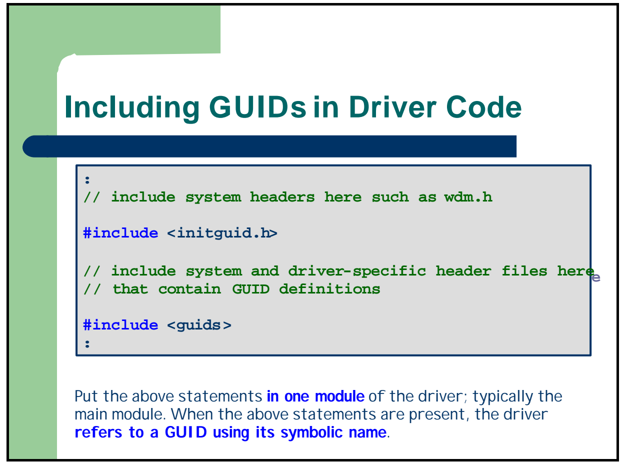
- Write an Windows application based on Win32 device management functions.
  - You can write one similar to WinObj with search capability.
- Write slides to describe the basic method to use the Win32 device management functions.
  - It is better to give some simple examples.

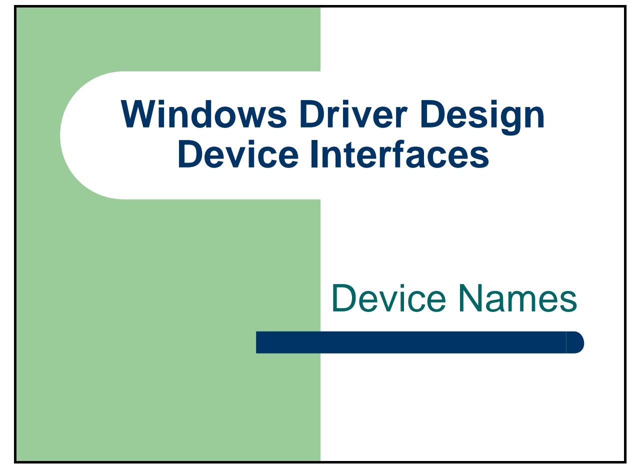


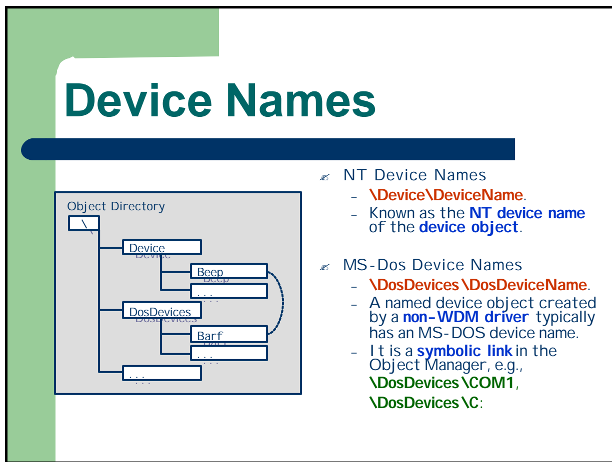


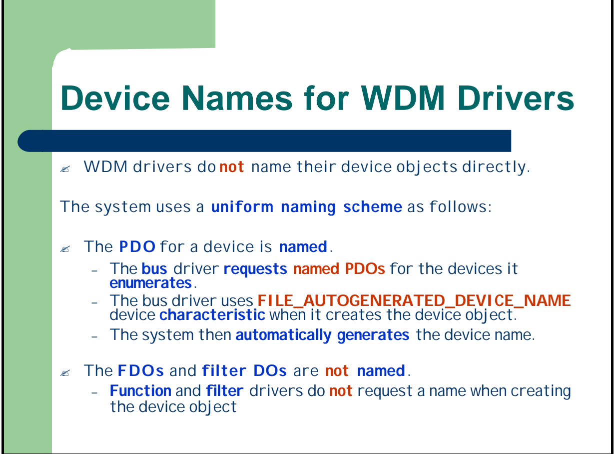












Device Names for WDM Drivers

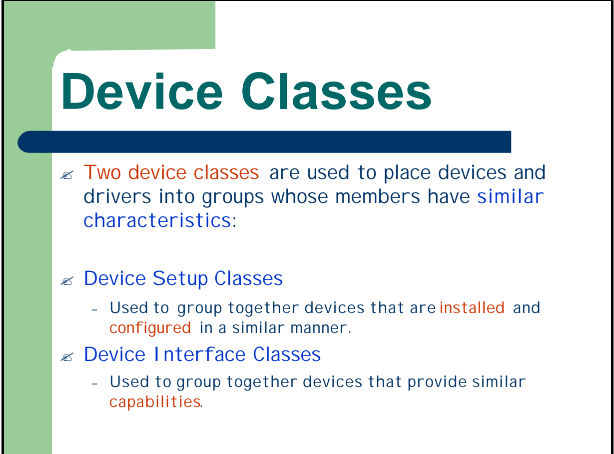
- Any **I/O request** to a named device object automatically goes to the **top** object in that device object's **stack**.
  - Thus, **only** the **PDO** is required to be **named**.
  - **User-mode** applications do **not** refer to WDM device objects by **name**;
  - instead, applications access the 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



Windows Driver Design Device Interfaces

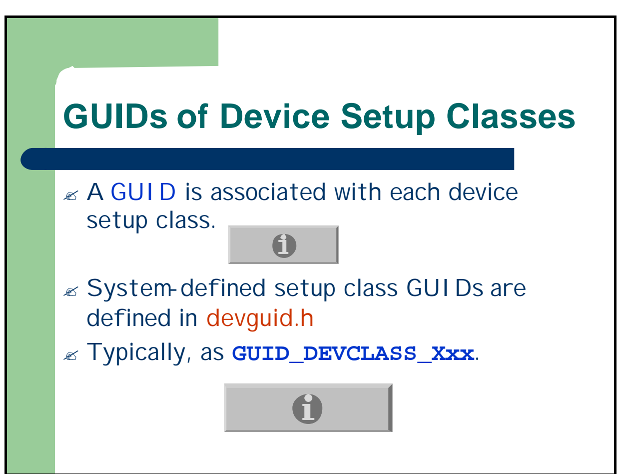
Device Setup Classes

Device Setup Classes

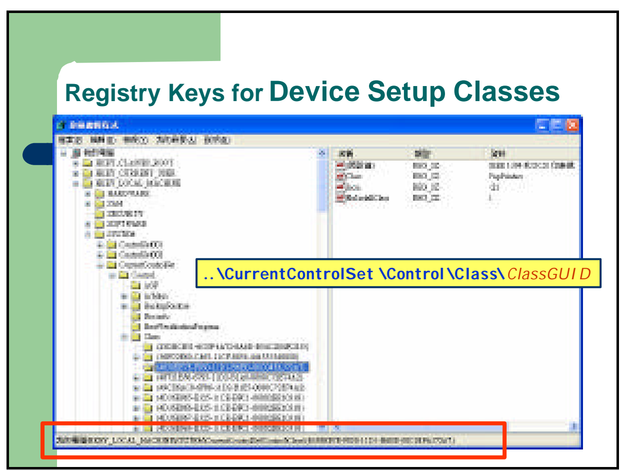
- To facilitate device installation
  - devices that are set up and configured in the same way are grouped into a device setup class.
- A device setup class defines the class installer and class co-installers involved in installing the device.
- For example, SCSI media changer devices are grouped into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



Registry Keys for Device Setup Classes

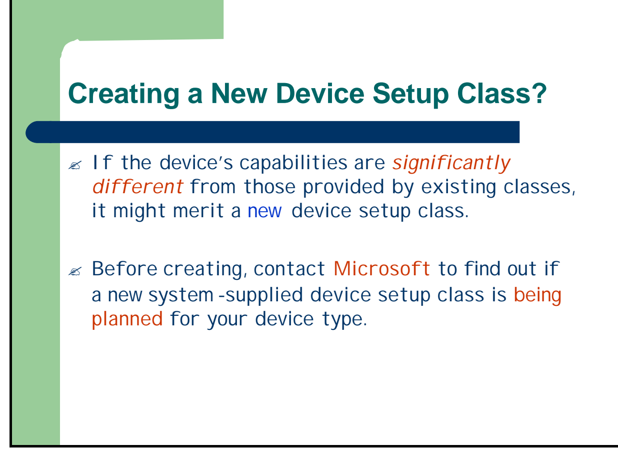


Creating a New Device Setup Class?

- 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.

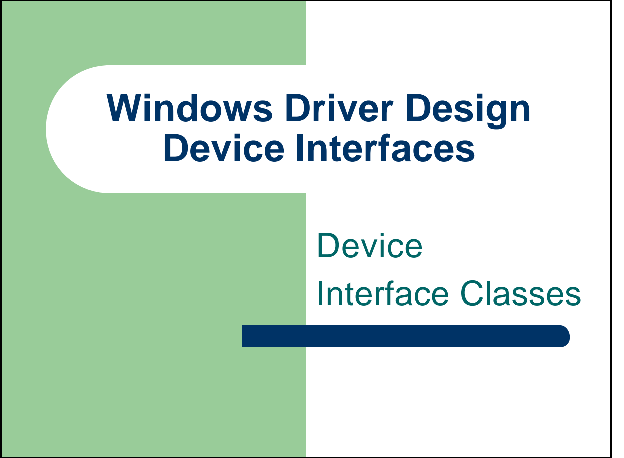
Creating a New Device Setup Class?

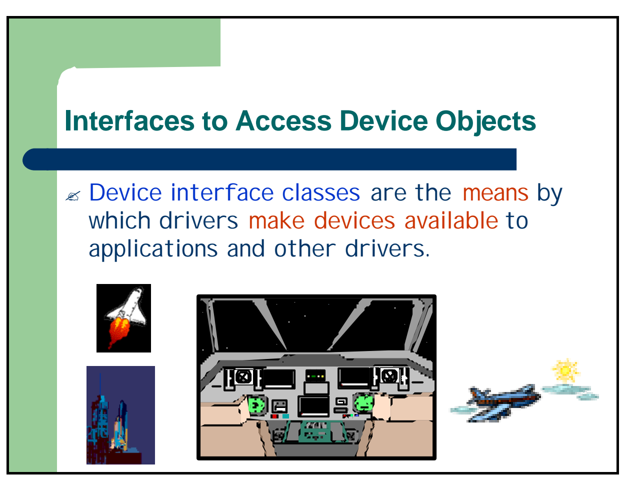
- Consider providing a device co-installer if
  - the device has installation requirements that are *not* supported by an existing device type-specific INF file.
  - Installation of your device requires device property pages that are *not* provided by an existing class.



How to Create a Device Setup Class?

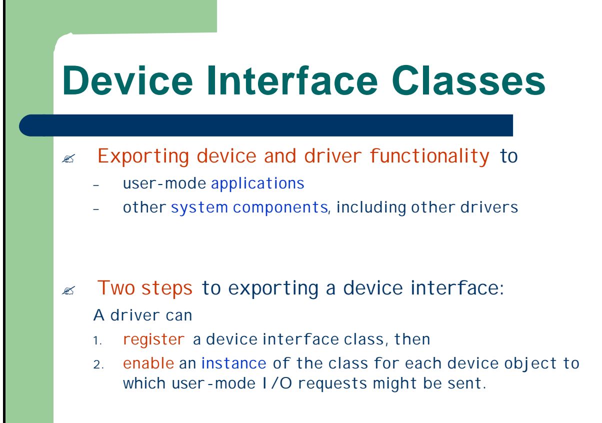
- Create a new device setup class using an INF file.
- Besides installing support for a device,
  - an INF file can initialize a new device setup class for the device.
  - such an INF file has an INF ClassInstall32 section.

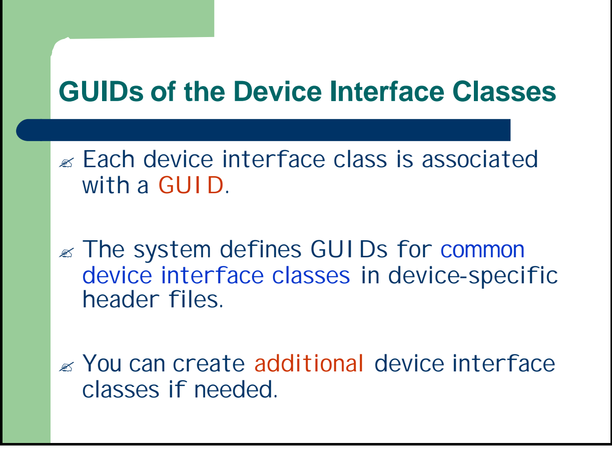


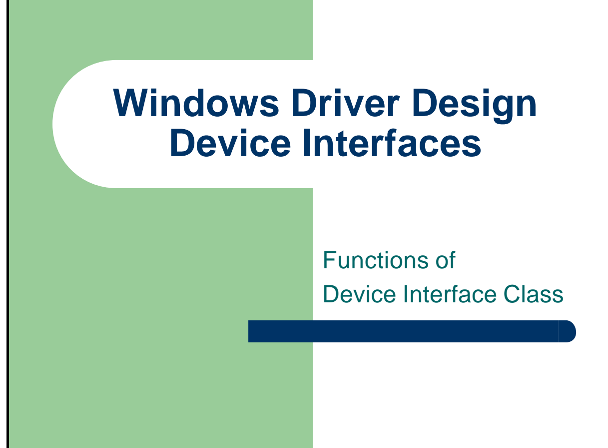


Naming the Device Objects

- 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

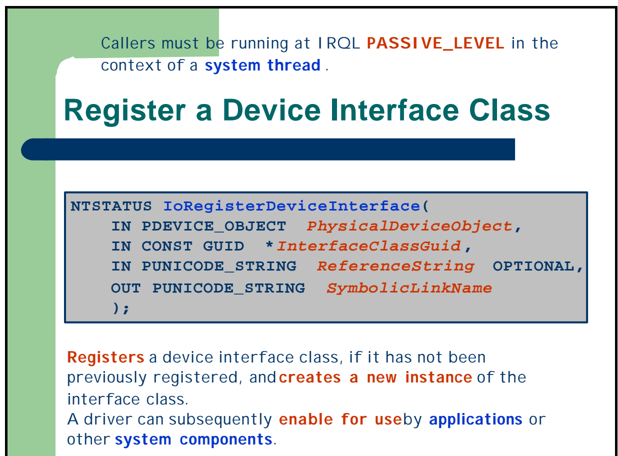


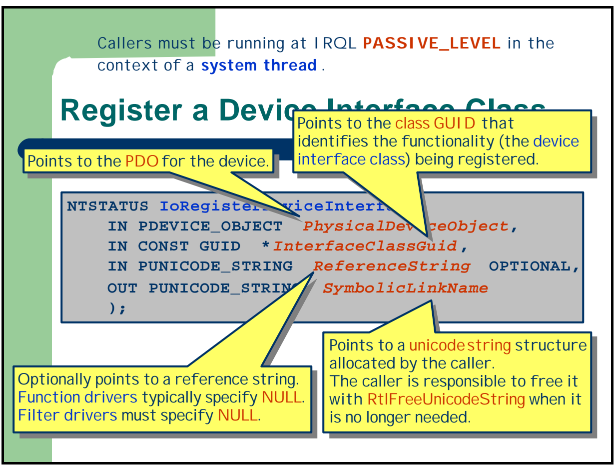


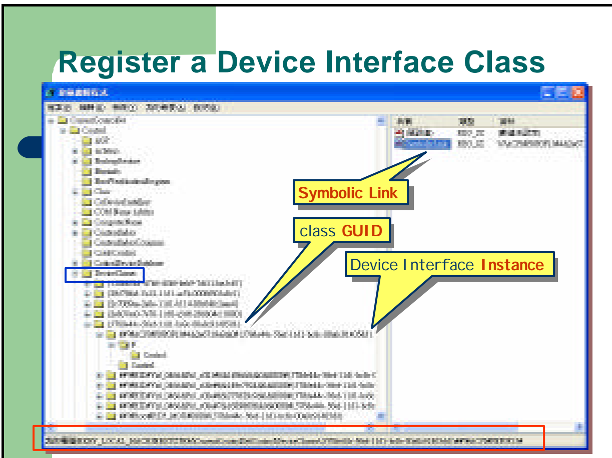


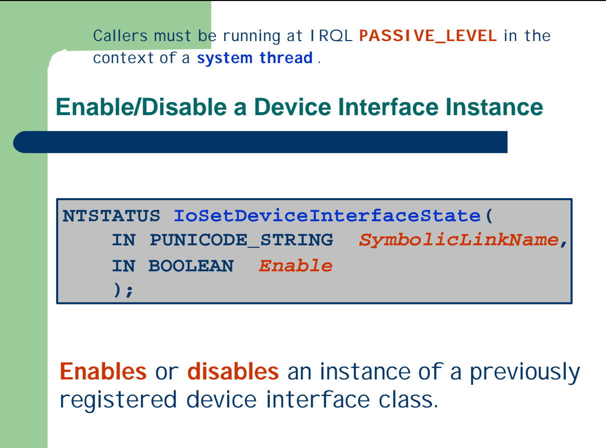
I/O Manager Routines

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









Callers must be running at IRQL **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

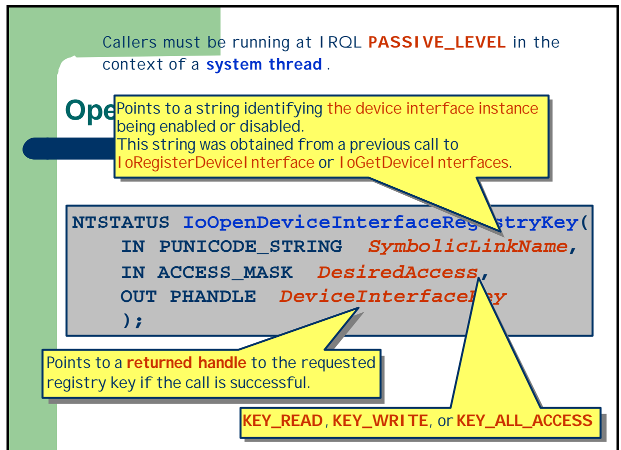
Points to a string identifying the device interface instance being enabled or disabled.
This string was obtained from a previous call to IoRegisterDeviceInterface or IoGetDeviceInterfaces.

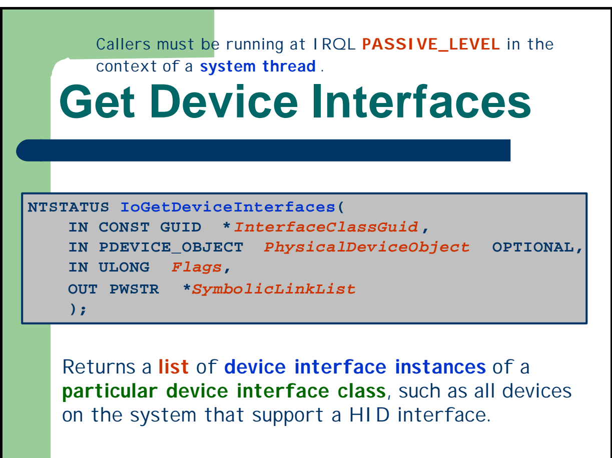
Callers must be running at IRQL **PASSIVE_LEVEL** in the context of a **system thread**.

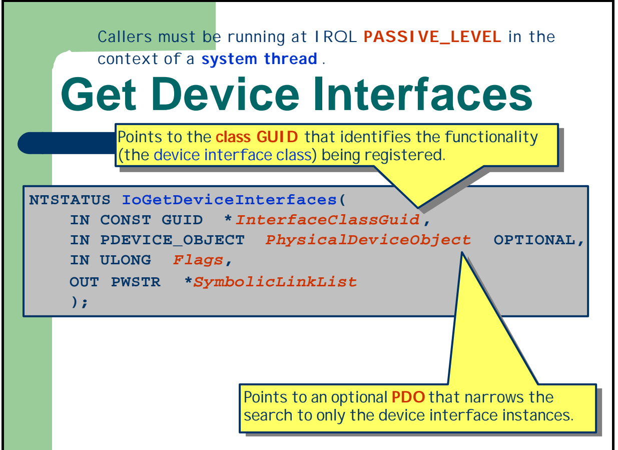
Open Device Interface Registry Key

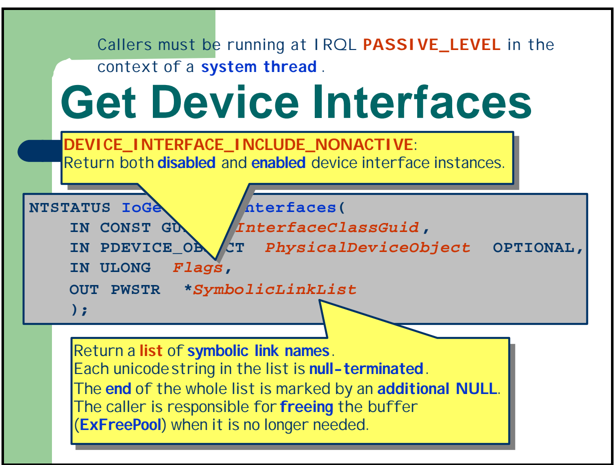
```
NTSTATUS IoOpenDeviceInterfaceRegistryKey(
    IN PUNICODE_STRING SymbolicLinkName,
    IN ACCESS_MASK DesiredAccess,
    OUT PHANDLE DeviceInterfaceKey
    );
```

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







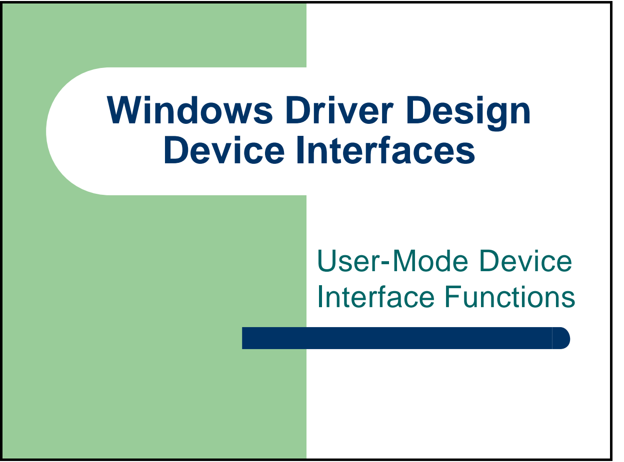


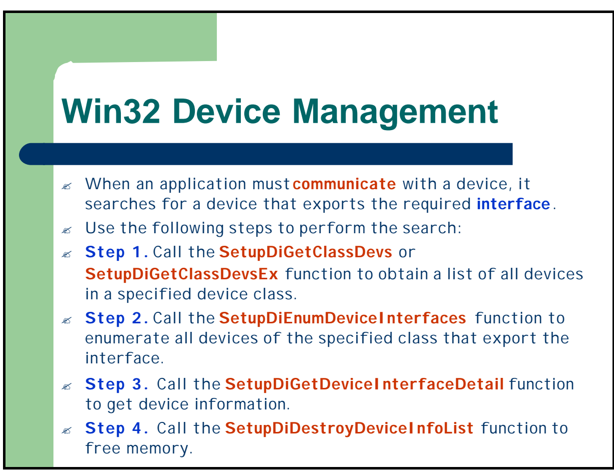
Callers' IRQL= **PASSIVE_LEVEL**

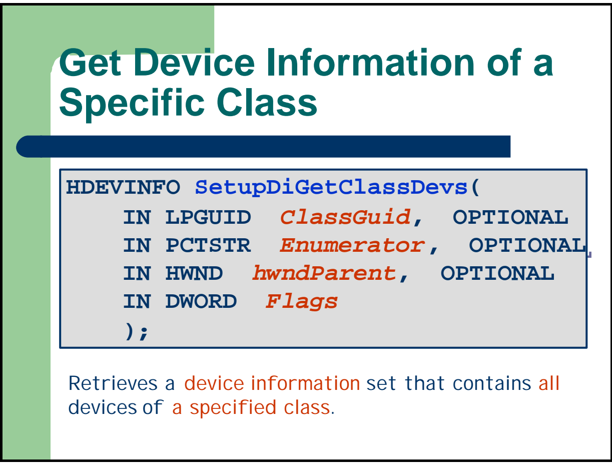
Get Device Object Pointer

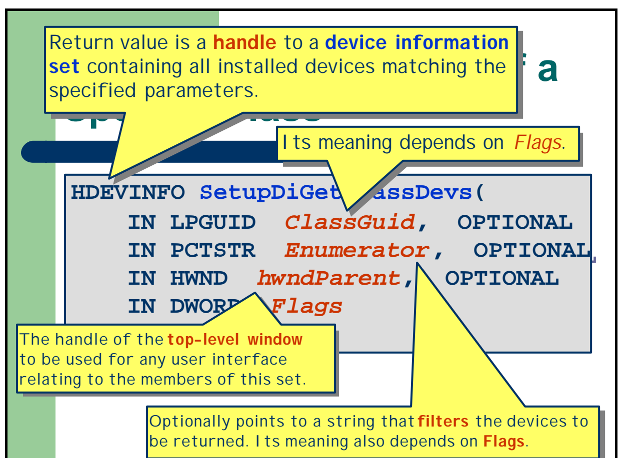
```
NTSTATUS IoGetDeviceObjectPointer(
    IN PUNICODE_STRING  ObjectName,
    IN ACCESS_MASK  DesiredAccess,
    OUT PFILE_OBJECT  *FileObject,
    OUT PDEVICE_OBJECT  *DeviceObject
);
```

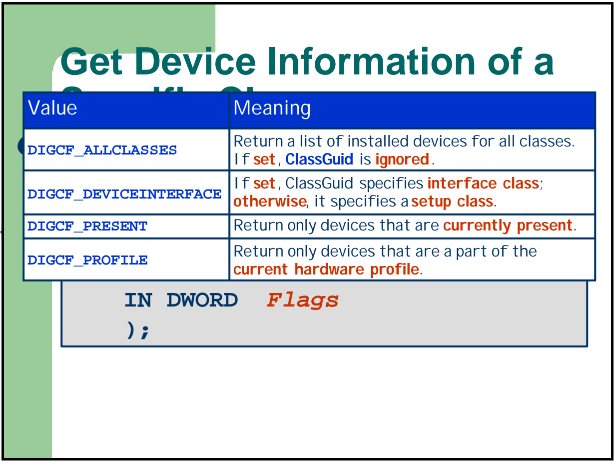
Returns a **pointer** to a **named device object** and corresponding **file object**, if the requested access to the objects can be granted.

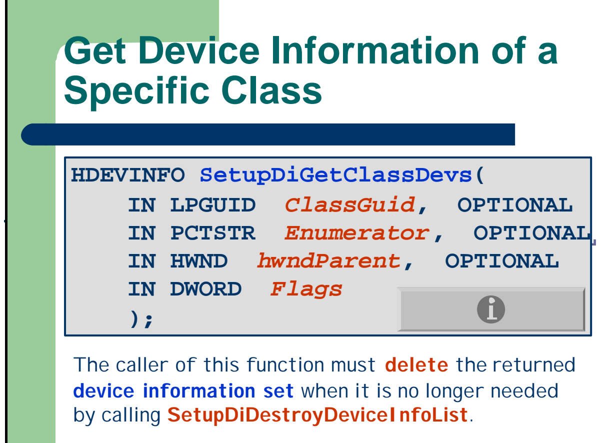



Win32 Device Management
When an application must communicate with a device, it searches for a device that exports the required interface.
Use the following steps to perform the search:
Step 1. Call the SetupDiGetClassDevs or SetupDiGetClassDevsEx function to obtain a list of all devices in a specified 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.

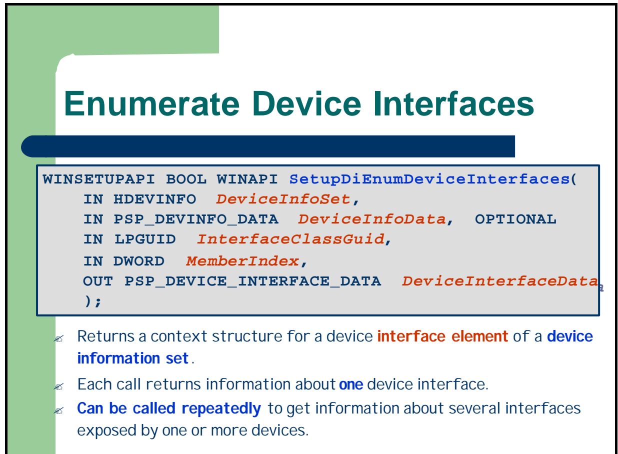

Get Device Information of a Specific Class
HDEVINFO SetupDiGetClassDevs(
IN LPGUID ClassGuid, OPTIONAL
IN PCTSTR Enumerator, OPTIONAL
IN HWND hwndParent, OPTIONAL
IN DWORD Flags
);
Retrieves a device information set that contains all devices of a specified class.

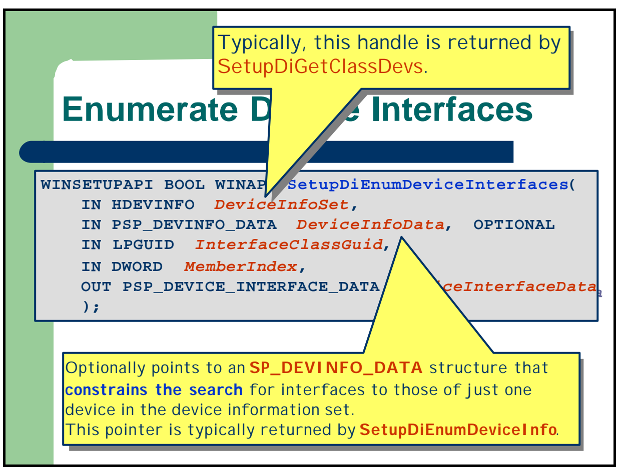

Return value is a handle to a device information set containing all installed devices matching the specified parameters.
Its meaning depends on Flags.
HDEVINFO SetupDiGet assDevs(
IN LPGUID ClassGuid, OPTIONAL
IN PCTSTR Enumerator, OPTIONAL
IN HWND hwndParent, OPTIONAL
IN DWORD Flags
The handle of the 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.

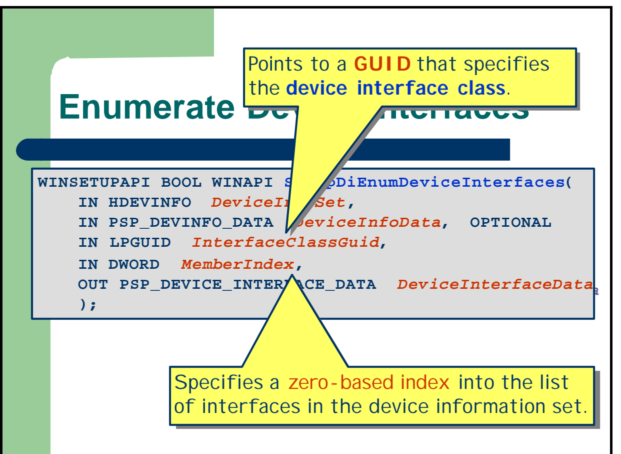

Get Device Information of a
Value
Meaning
DIGCF_ALLCLASSES
Return a list of installed devices for all classes. If set, ClassGuid is ignored.
DIGCF_DEVICEINTERFACE
If set, ClassGuid specifies interface class; otherwise, it specifies a setup class.
DIGCF_PRESENT
Return only devices that are currently present.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

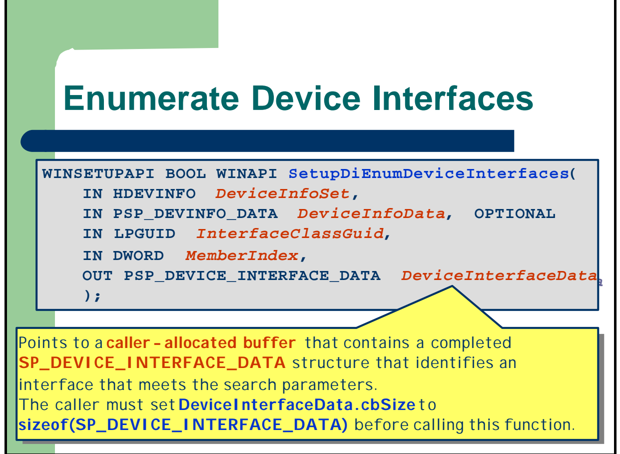

Get Device Information of a Specific Class
HDEVINFO SetupDiGetClassDevs(
IN LPGUID ClassGuid, OPTIONAL
IN PCTSTR Enumerator, OPTIONAL
IN HWND hwndParent, OPTIONAL
IN DWORD Flags
);
The caller of this function must delete the returned device information set when it is no longer needed by calling SetupDiDestroyDeviceInfoList.

Destroy Device Information Set
WINSETUPAPI BOOL WINAPI
SetupDiDestroyDeviceInfoList(
IN HDEVINFO DeviceInfoSet
);
Destroys a device information set and frees all associated memory.





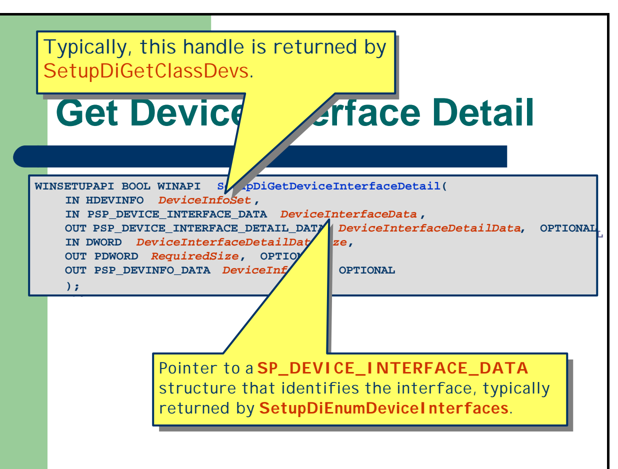


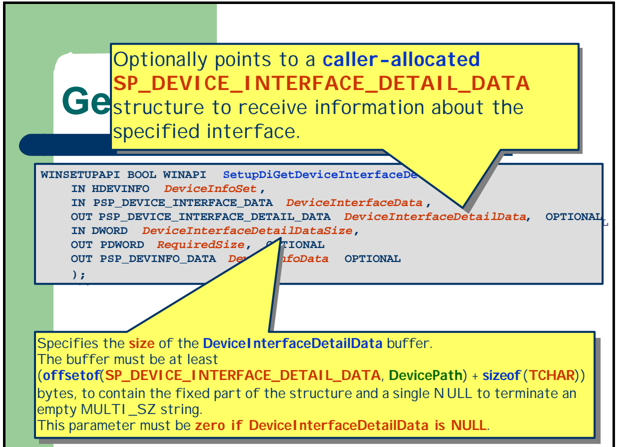


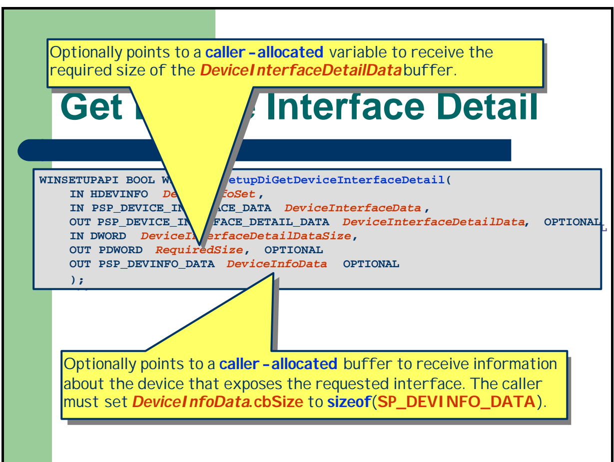
Get Device Interface Detail

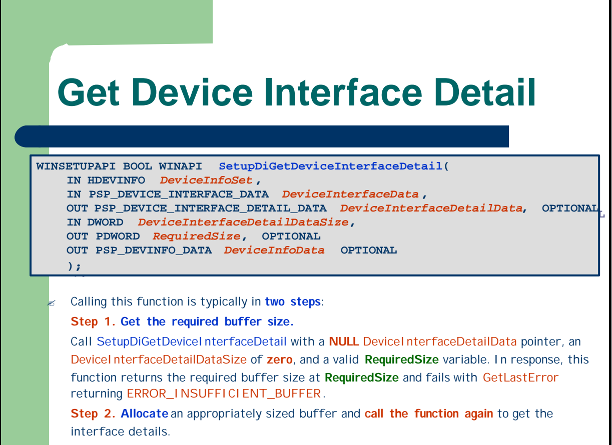
```
WINSETUPAPI BOOL WINAPI   SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA  DeviceInterfaceDetailData,  OPTIONAL
    IN DWORD  DeviceInterfaceDetailDataSize,
    OUT PDWORD  RequiredSize,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

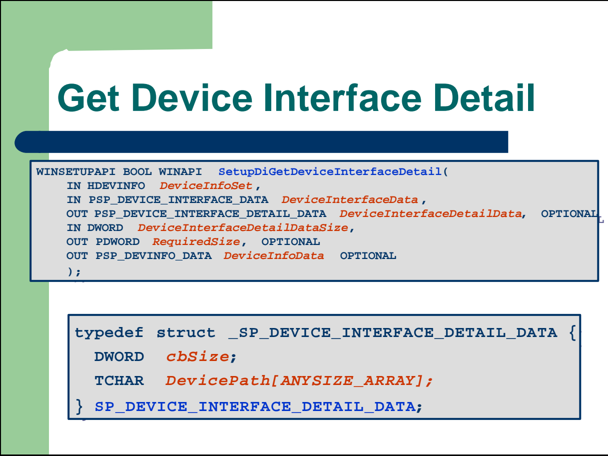
Returns details about a particular device interface.

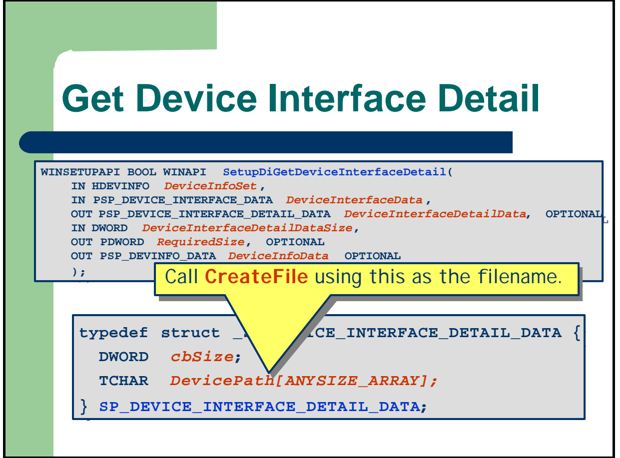


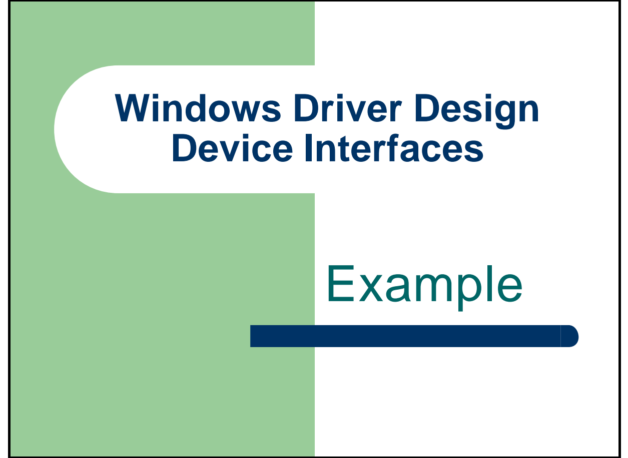

Ge
Optionally points to a caller-allocated SP_DEVICE_INTERFACE_DETAIL_DATA structure to receive information about the specified interface.
WINSETUPAPI BOOL WINAPI SetupDiGetDeviceInterfaceDe
IN HDEVINFO DeviceInfoSet ,
IN PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData ,
OUT PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData, OPTIONAL
IN DWORD DeviceInterfaceDetailDataSize,
OUT PDWORD RequiredSize,
OUT PSP_DEVINFO_DATA
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 NULL to terminate an empty MULTI_SZ string.
This parameter must be zero if DeviceInterfaceDetailData is NULL.

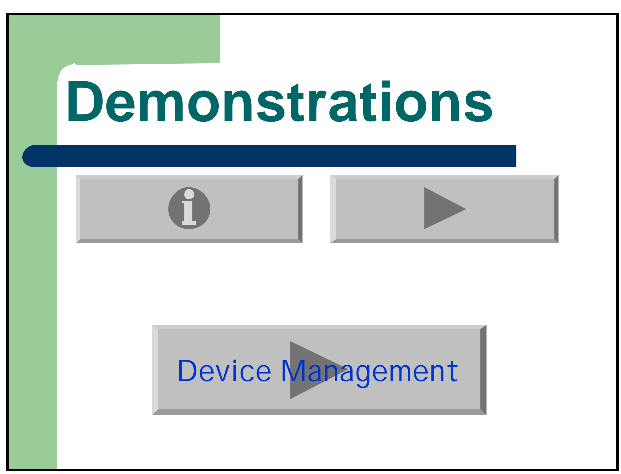

Optionally points to a caller-allocated variable to receive the required size of the DeviceInterfaceDetailData buffer.
Get
Interface Detail
OUT PDWORD RequiredSize, OPTIONAL
OUT PSP_DEVINFO_DATA DeviceInfoData OPTIONAL
);
Optionally points to a caller-allocated buffer to receive information about the 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.
DWORD cbSize;
} SP_DEVICE_INTERFACE_DETAIL_DATA;


Windows Driver Design
Device Interfaces
Example


Demonstrations
Device Management