Universal Serial Bus
Understanding WDM Power Management, Version 1.1

August 7, 2000

Kosta Koeman
Intel Corporation
kosta.koeman@intel.com

Abstract

This white paper provides an overview of power management in the WDM architecture and the necessary code required to implement minimal support. This paper makes the assumption that the reader is experienced writing WDM USB drivers and is familiar with USB bus analyzer tools such as the CATC™ product line (see http://www.catc.com for more information).

Contributors

I would like to thank the following people for providing valuable input and information that improved the content of this white paper.

John Keys, Intel Corporation

Mark McCoy, Cypress Semiconductor Corporation

Phong Minh, Lexar Media Inc.

Walter Oney, Walter Oney Software

Revision History

Revision Number	Release Notes
1.0	Original Release
1.1	Updates for Windows™ 98 Millennium Edition Code enhancements

Table Of Contents

Introduction

White Paper Overview

This white paper is a brief tutorial for proper power management implementation. This paper covers handling of the IRPs, generated by the device driver and the operating system's I/O and power managers, that relate to power management. The I/O manager dispatches various PnP IRPs to the device stack and the power manager dispatches various power IRPs. All of these IRPs will be discussed in greater detail later in this paper.

This paper begins with a brief overview of the power states and their respective definitions for systems and devices. The power state overview is followed by the parameters (device capabilities, power state information, etc.) that must be stored in the device extension in order to simplify supporting power management. The reader is then informed how to acquire the device capabilities information.

The minor power IRP codes are then discussed individually. In addition to the proper implementation of the minor power functions, issues are addressed that go beyond available documentation, and solutions will be provided to work around and accommodate these problems. The code provided in this paper is a mixture of *Windows® 2000* DDK (March 9, 2000 build) sample code and code derived from the documentation and through knowledge of existing issues.

Before beginning the overview of power states, it is important to remember the layering of drivers in the WDM architecture. Figure 1 shows a simplified view of the device driver layering. The functional device object is the device object that the under control of the device driver. The corresponding physical device object is the physical abstraction created by the hub's functional device object. There can be multiple layers of hub functional and physical device objects (depending on the number between the device and the root hub). The bottom of the stack is the (USB) bus functional device object. This paper will now refer to this stack of device objects as the USB Stack. However, there are some IRPs that travel to and from deeper in the stack, such as to the ACPI driver.

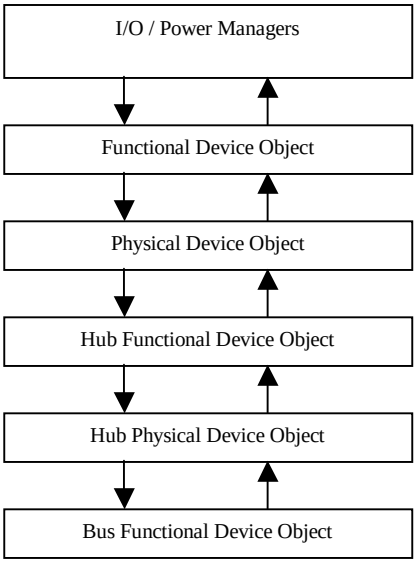


Figure 1. Simplified Device Layering View

Overview of Power States

In the WDM architecture, there are five system and four device power states that range from fully on, to sleeping or suspended, to fully off. The names and meanings of these power states are summarized in **Table 1** and **Table 2**. For more information on the ACPI system sleep states, see [1].

System Power States

System State	Meaning
PowerSystemWorking (S0)	System fully on
PowerSystemSleeping1 (S1)	• System fully on, but sleeping • Most APM machines go to this state
PowerSystemSleeping2 (S2)	• Processor is off • Memory is on, • PCI is on
PowerSystemSleeping3 (S3)	• Processor is off • Memory is in refresh • PCI receives auxiliary power
PowerSystemHibernate (S4)	OS saves context before power off
PowerSystemShutdown (S5)	System fully off, no context saved

Table 1. Summary of System Power States

The first and highest state, *PowerSystemWorking* or *S0*, corresponds to the state in which the system is in normal operation with all devices on. If a machine in *S0* is idle for long enough, the system may power off the monitor and power down the harddrive(s). In this state, the USB bus is fully on. See [2] for more information.

The first sleep state, *PowerSystemSleeping1* or *S1*, corresponds to the system fully on but sleeping. At this point, the harddrive(s) are powered down and the monitor is powered off. Most APM machines power down to this state when suspended. In this state, the USB bus is suspended, and V_{bus} is still powered to 5 volts. See [3] for more information.

The second sleep state, *PowerSystemSleeping2* or *S2*, corresponds the processor being turned off. Main memory is still active and the PCI bus is powered. This state is typically not used in the Windows® operating systems. V_{bus} is still powered to 5 volts. See [3] for more information.

The third sleep state, *PowerSystemSleepingS3* or *S3*, corresponds to the sleep state in which memory is placed in the refresh state (memory is still refreshed periodically, but no memory access occurs) and the PCI bus is powered with auxiliary power. This suspend state is the goal of ACPI machines. While in the *S3* sleep state, older platforms power off the USB bus. Newer motherboards power the USB bus using auxiliary power while in *S3*. Examples of motherboards that have this capability use Intel's 82820 and 82815 chipsets. See [3] for more information.

Powering down to the fourth sleep state, *PowerSystemHibernate* or *S4*, involves first saving the operating system context to hard disk before powering off the system. The purpose of this sleep state is to provide a means for quick reboot of a PC. USB is powered off in this state. *Windows® 98 ME* (a.k.a. Millennium) and *Windows® 2000* both support *S4*. See [3] for more information.

The final sleep state, *PowerSystemShutdown* or *S5*, corresponds simply to power being turned off. No operating system is saved before entering this state. The user places the PC in this state by selecting *Start→Shutdown→Shutdown*. USB is powered off in this state. See [4] for more information. *Windows® 2000* provides extra information when setting this state. See [5] for more information.

Simplified System Power State View

These five system power states can be categorized or viewed into three states by USB devices: powered on, suspended, and off. The mapping of these three simplified state is shown in the table below. The purpose of introducing these states is to simply the view of these power states as they apply to USB.

Simplified State	System Power States
Powered On	PowerSystemWorking
Suspended	• PowerSystemSleepingS1 • PowerSystemSleepingS2 • (PowerSystemSleepingS3)
Powered Off	• (PowerSystemSleepingS3) • PowerSystemHibernate • PowerSystemShutdown

Table 2. Simplified System Power State View

It is important to note that the *PowerSystemSleepingS3* state is listed in both the suspended and powered off simplified state. The reason for this dual assignment is the fact that new ACPI systems maintain bus power (V_{bus} at 5 volts) where others do not. Also, all current PCI-USB add-in cards do not amplify the PCI bus voltage (3.3 volts) to the level needed (5 volts) to power the USB bus. Therefore, if V_{bus} is maintained to 5 volts on a system, the simplified state will be suspended. If not, the simplified state will be powered off since from an operating system point of view, all USB devices have been removed.

Device Power States

The four device power states consist of a full power state, *PowerDeviceD0* or simply *D0*, and three sleep states, *PowerDeviceD1* or *D1*, *PowerDeviceD2* or *D2*, *PowerDeviceD3* or *D3*. According to the DDK, the difference between the sleep states is in the latency in returning to the full power state, *PowerDeviceD0*. As will be seen later, *PowerDeviceD3* will be used as an "off" state when the USB bus is powered off to maintain consistency with the DDK documentation. For more information on ACPI device power states, see [6].

Device Power State	Meaning
PowerDeviceD0 (D0)	Full Power. Device fully on.
PowerDeviceD1 (D1)	Low sleep state with lowest latency in returning to PowerDeviceD0 state.
PowerDeviceD2 (D2)	Medium sleep state
PowerDeviceD3 (D3)	Full sleep state with longest latency in returning to PowerDeviceD0. (Note: Commonly referred to as "off," however a USB device's parent port is not powered off, but rather suspended.)

Table 3. Summary of Devices Power States

Simplified View of Device Power States

For device power states, there is no difference in the status of a device's parent port between the three device sleep states: *PowerDeviceD1*, *PowerDeviceD2*, and *PowerDeviceD3*. Powering down into any of these power states result in the device being suspended. For USB, there are basically two power categories: fully powered or suspended. However, the driver should still set device power states during system power state changes according to the power state mapping in the device capabilities structure. The state of powered off (i.e., USB power is off) involves the driver being unloaded and does not correspond to the *PowerDeviceD3* state.

Supporting Power Management

Supporting power management requires cooperation between the power manager and the device driver. The power manager [7] manages system power and tracks power IRPs travelling through the system. The device driver must implement a certain amount of functionality in order to support power management properly [8].

This white paper covers the necessary components and code for supporting power management. First, the information needed to store in the device extension, which include the device capabilities structure filled in by the bus driver. Second, the relevant components of the device capabilities structure are discussed, followed by the appropriate means of acquiring and storing that information.

Next, the four power IRPs are discussed and sample code is provided for supporting these IRPs. The next section focuses on IRP sequences that a driver will receive and/or generate in specific power events. How to generate power IRPs is covered in the next section.

Finally, the worst-case scenario is discussed followed by the consequences of not supporting power management properly.

Power Information To Store In The Device Extension

For simplifying power management implementation, it is necessary to store the device capabilities [9], as well as the current system and device states, and a flag for signaling device power transitions. A portion of the device extension is shown below. The reason for saving this information in the device is to record current system and device power states (due to `IRP_MN_SET_POWER` IRPs), monitoring power state transitions (due to `IRP_MN_QUERY_POWER` IRPs), and managing device power according to system power state change. Each of these will be discussed further in the discussion of the four minor power IRP codes.

```
typedef struct _DEVICE_EXTENSION {
.
.
.
DEVICE_CAPABILITIES DeviceCapabilities;
POWER_STATE CurrentSystemState;
POWER_STATE CurrentDeviceState;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Sample Code 1. Parameters stored the device extension for proper power management support

Overview of Device Capabilities Structure

The device capabilities structure stores power management related parameters. The host controller driver fills in this structure based upon the BIOS's ability to support power management for the host controller. For basic support of power management, only a few components of the `DEVICE_CAPABILITIES` are used: *DeviceState*, an array of `DEVICE_POWER_STATE` values that link system power states to corresponding device power states; *SystemWake*, a `SYSTEM_POWER_STATE` value that indicates what is the lowest system power state in which the device may wake up the system; and *DeviceWake*, the lowest device power state make wakeup itself or the system.

The *DeviceState* array provides a mapping on which device power state a device should set itself to when a system is entering some sleep state. The mapping of a USB device is shown in **Table 3**.

System Power State	Device State (Wakeup supported)	Device State (Wakeup not supported)
PowerSystemWorking	PowerDeviceD0	PowerDeviceD0
PowerSystemSleeping1	PowerDeviceD2	PowerDeviceD3
PowerDeviceSleeping2	PowerDeviceD2	PowerDeviceD3
PowerSystemSleeping3	PowerDeviceD2	PowerDeviceD3
PowerSystemHibernate	PowerDeviceD3	PowerDeviceD3
PowerSystemShutdown	PowerDeviceD3	PowerDeviceD3

Table 3. *DeviceState* Values

The *SystemWake* value is used to determine whether or not to issue a wait/wake IRP on a suspend event. If the USB device supports remote wakeup, this value should be *PowerSystemSleeping3* since this will be the minimal power state in which the USB bus could be powered.

The *DeviceWake* value is used to determine which state to enter when entering a low power state, for example, when aggressively managing power. If the USB device supports remote wakeup, this value should be *PowerDeviceD2*.

Acquiring Device Capabilities

After receiving an IRP_MN_START_DEVICE PnP IRP, the I/O manager will issue an IRP_MN_QUERY_CAPABILITIES IRP. The driver should attach a completion routine (as shown below) in which the device capabilities' power values are modified and stored in the device extension. The modifications of the device capabilities include setting the Removable flag to TRUE and setting the SurpriseRemovalOK flag to TRUE. In *Windows® 2000*, if either of these values are set to FALSE, then the user must stop the device before disconnecting it. Setting these values to TRUE removes this requirement and allows for the device to be truly hot pluggable. Of course, the consequence is that the IRP_MN_SURPRISE_REMOVAL PnP must be properly supported.

```
// in the PnP dispatch routine
case IRP_MN_QUERY_CAPABILITIES: // 0x09
    IoCopyCurrentIrpStackLocationToNext(IRP);
    IoSetCompletionRoutine(IRP,
                           QueryCapabilitiesCompletionRoutine,
                           DeviceObject,
                           TRUE,
                           TRUE,
                           TRUE);

    ntStatus = IoCallDriver(deviceExtension->StackDeviceObject, Irp);

    return ntStatus;
    break;
    .
    .
    .
    return ntStatus;
}
```

Sample Code 2. Attaching Completion Routine to the IRP_MN_QUERY_CAPABILITIES IRP

```
NTSTATUS
QueryCapabilitiesCompletionRoutine(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    )
{
    PDEVICE_OBJECT deviceObject       = (PDEVICE_OBJECT) Context;
    PDEVICE_EXTENSION deviceExtension = deviceObject->DeviceExtension;
    NTSTATUS ntStatus = STATUS_SUCCESS;
    PIO_STACK_LOCATION IRPStack = IoGetCurrentIRPStackLocation (Irp);
    PDEVICE_CAPABILITIES Capabilities =
                           IrpStack->Parameters.DeviceCapabilities.Capabilities;
    ULONG ulPowerLevel;

    // If the lower driver returned PENDING, mark our stack location
    // as pending also.
    if (Irp->PendingReturned)
    {
        IoMarkIrpPending(Irp);
    }
    ASSERT(IrpStack->MajorFunction == IRP_MJ_PNP);
    ASSERT(IrpStack->MinorFunction == IRP_MN_QUERY_CAPABILITIES);

    // Modify the PnP values here as necessary
    Capabilities->Removable = TRUE;           // Make sure the systems sees this
    Capabilities->SurpriseRemovalOK = TRUE;   // No need to display window if not
                                              // removing via applet (Win2k)
```

```
    // Save the device capabilities in the device extension
    RtlCopyMemory(&deviceExtension->DeviceCapabilities,
                  DeviceCapabilities,
                  sizeof(DEVICE_CAPABILITIES));

    // print out capabilities info
    KdPrint(("************ Device Capabilites ************\n"));
    KdPrint(("SystemWake = %s (0x%x)\n",
                 SystemPowerStateString[deviceExtension->DeviceCapabilities.SystemWake],
                 deviceExtension->DeviceCapabilities.SystemWake));

    KdPrint(("DeviceWake = %s\n",
                 DevicePowerStateString[deviceExtension->DeviceCapabilities.DeviceWake],
                 deviceExtension->DeviceCapabilities.SystemWake));

    for (ulPowerLevel=PowerSystemUnspecified;
         ulPowerLevel< PowerSystemMaximum;
         ulPowerLevel++)
    {
        KdPrint(("Dev State Map: sys st %s = dev st %s\n",
                    SystemPowerStateString[ulPowerLevel],
                    DevicePowerStateString[
                        deviceExtension->DeviceCapabilities.DeviceState[ulPowerLevel]]));
    }
    Irp->IoStatus.Status = STATUS_SUCCESS;

    return ntStatus;
}
```

Sample Code 3. `IRP_MN_QUERY_CAPABILITIES` Completion Routine

The Four Power IRP Minor Functions

There are four power IRP codes, as shown in the table below, may be generated by either the power manager [10], the device driver [11] or both. Only the device driver may generate the two optional IRPs, power sequence [12] and wait/wake [13]. The device driver may also generate set power and query power IRPs, but only of with of device type, not system. The power manager will generally generate only the system query power and set system power IRPs to control system power. However, if a driver registers for idle detection via the *PoRegisterDeviceForIdleDetection()* [14], then the power manager will issue a set device power IRP if the device has not called *PoSetDeviceBusy()* [15] within the maximum idle time.

Minor IRP Code	Optional/Required	Generated By
IRP_MN_POWER_SEQUENCE	Optional	Device
IRP_MN_QUERY_POWER (System)	Required	Power Manager
IRP_MN_QUERY_POWER (Device)	Required	Both
IRP_MN_SET_POWER (Device)	Required	Both
IRP_MN_SET_POWER (System)	Required	Power Manager
IRP_MN_WAIT_WAKE	Optional	Device

Table 4. Minor Function Power IRP Codes

IRP_MN_POWER_SEQUENCE

This optional IRP is used for optimizing device state change processing. The device driver generates this IRP to determine the number of times the device has entered the different device power states, possibly keeping track by storing this information in the device extension. This paper defers to reader to [12] for more information.

IRP_MN_QUERY_POWER

In general, the power manager is the policy owner of this minor IRP code [16]. However, a device driver may generate this IRP as well. The power manager uses this IRP to request approval from all device drivers for the system to enter a specific sleep state. This IRP is not sent when powering up the system (always to *S0*) or under critical situations where system power is about to be lost. If the system wishes to change sleep states, the system will first power up to *S0*, and then go into the other low power state.

The rest of this section focuses on advising on the driver's policy for handling suspend to *S1* through *S4* requests. All devices must accept system power requests to *S4* and *S5*. It should be noted that when a system is shut down, the OS might not issue a query IRP before a set system power IRP.

When the operating systems issues a query IRP to enter *S1* or *S3*, the device driver has two options when transmitting data: reject the IRP, thus preventing the system from entering that sleep state; or stop all data transmissions, queue up further data transmitting IRPs, and pass down

the query IRP down the stack. If idle, the device driver must perform the latter option to allow the system to enter the sleep state.

In some cases, the vendor may wish to provide the consumer the option of which choice the driver makes whenever the preferred option is not obvious. This can be done by the device driver's corresponding application toggling a registry value that is read by the device driver.

Cases in which the vendor may wish to grant the consumer this option may include scanners, video cameras, even mass storage devices, and other devices transmitting non-critical data. Upon system resume, a well-written device driver would be capable of completing a file transfer, completing the scan, resume video transfer, or whatever task it had before the system entered the sleep state.

Cases in which the device driver should reject the system's request enter a sleep state are those in which data would be lost or corrupted. The best examples of this case are printers (print job may be ruined by stopping/resuming) and communication devices, such as modems, that are transmitting data (as opposed to maintaining a connection). Communication devices that are simply maintaining a connection should allow the system to enter a suspend state and cause a remote wakeup event whenever actual data is received.

Ultimately, vendors know their customers best and must decide what is the *suspend-while-transmitting* policy and if it is appropriate to provide their customers the option of changing that policy. The following source code provides the flexibility of having consumer preferences dictate the policy regarding suspend while the device is transmitting data.

```
NTSTATUS
DispatchPower(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{
    PDEVICE_EXTENSION  deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS           ntStatus        = STATUS_SUCCESS;

    KdPrint(("DispatchPower() IRP_MJ_POWER\n"));

    switch (irpStack->MinorFunction)
    {
    .
    .
    .
    case IRP_MN_QUERY_POWER:

        if (irpStack->Parameters.Power.Type == SystemPowerState)
        {
            HandleSystemQueryIrp(DeviceObject, Irp);
        }
        else if (irpStack->Parameters.Power.Type == DevicePowerState)
        {
            HandleDeviceQueryIrp(DeviceObject, Irp);
        }
        break;
```

Sample Code 4. IRP_MN_QUERY_POWER

```
NTSTATUS
HandleSystemQueryIrp(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PDEVICE_EXTENSION  deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS           ntStatus        = STATUS_SUCCESS;

    BOOLEAN fNoCompletionRoutine = FALSE;
    BOOLEAN fPassDownIrp         = TRUE;

    KdPrint(("=====================================\n"));
    KdPrint(("Power() IRP_MN_QUERY_POWER to %s\n",
             SystemPowerStateString[irpStack->Parameters.Power.State.SystemState]));

    // first determine if we are transmitting data
    if (deviceExtension->ulOutStandingIrps > 0)
    {
        BOOLEAN fOptionDetermined = FALSE;
        ULONG   ulStopDataTransmissionOnSuspend = 1;
        BOOLEAN fTransmittingCritialData = FALSE;

        // determine if driver should stop transmitting data
        fOptionDetermined = (BOOLEAN)GetRegistryDword(&gRegistryPath,
                                            L"StopDataTransmissionOnSuspend",
                                            &ulStopDataTransmissionOnSuspend);
        // Note COMPANY_X_PRODUCT_Y_REGISTRY_PATH is the absolute registry path
        // which would be defined as:       L"\\REGISTRY\\Machine\\System…
        // …\\CurrentControlSet\\Services\\COMPANY_X\\PRODUCT_Y and corresponds
        // to the following section in the registry (created by an .inf file or
        // installation routine): HKLM\\System\\CurrentControlSet\\Services…
        // …\\COMPANY_X\\PRODUCT_Y which would contain the DWORD entry
        // StopDataTransmissionOnSuspend

        if (ulStopDataTransmissionOnSuspend ||
            !fOptionDetermined ||
            irpStack->Parameters.Power.State.SystemState == PowerSystemShutdown)
            // stop data transmission if the option was set to do so or if the
            // option could not be read or if the system is entering S5
        {
            // Set a flag used to queue up incoming irps
            deviceExtension->PowerTransitionPending = TRUE;

            // attach a completion routine to wait for
            IoCopyCurrentIrpStackLocationToNext(Irp);
            IoSetCompletionRoutine(Irp,
                                   PoSystemQueryCompletionRoutine,
                                   DeviceObject,
                                   TRUE,
                                   TRUE,
                                   TRUE);

            fNoCompletionRoutine = FALSE;
        }
        else
            fPassDownIrp = FALSE;
    }

    ntStatus = PassDownPowerIrp(DeviceObject, Irp, fPassDownIrp, fNoCompletionRoutine);

    return ntStatus;
}
```

Sample Code 5. Handle System Query Irps routine

The following section of code provides a generic routine that reads the specified DWORD value from the designated registry path.

```
BOOLEAN
Usb_GetRegistryDword(
    IN      PWCHAR    RegPath,
    IN      PWCHAR    ValueName,
    IN OUT  PULONG    Value
    )
{
    UNICODE_STRING path;
    RTL_QUERY_REGISTRY_TABLE paramTable[2];  //zero'd second table terminates parms
    ULONG lDef = *Value;                     // default
    NTSTATUS status;
    BOOLEAN fres;
    WCHAR wbuf[ MAXIMUM_FILENAME_LENGTH ];

    path.Length = 0;
    path.MaximumLength = MAXIMUM_FILENAME_LENGTH * sizeof( WCHAR );
                                             // MAXIMUM_FILENAME_LENGTH defined in wdm.h
    path.Buffer = wbuf;

    RtlZeroMemory(path.Buffer, path.MaximumLength);
    RtlMoveMemory(path.Buffer, RegPath, wcslen( RegPath) * sizeof( WCHAR ));

    RtlZeroMemory(paramTable, sizeof(paramTable));

    paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
    paramTable[0].Name = ValueName;
    paramTable[0].EntryContext = Value;
    paramTable[0].DefaultType = REG_DWORD;
    paramTable[0].DefaultData = &lDef;
    paramTable[0].DefaultLength = sizeof(ULONG);


    status = RtlQueryRegistryValues(RTL_REGISTRY_ABSOLUTE | RTL_REGISTRY_OPTIONAL,
                                    path.Buffer, paramTable, NULL, NULL);

    fres = (NT_SUCCESS(status)) ? TRUE : FALSE;
    return fres;
}
```

Sample Code 6. Reading a DWORD entry from the registry

The author has noted in some driver writing books the reader is instructed to set the IRP status to STATUS_SUCCESS, call *PoStartNexPowerIRP*(), and then complete the IRP with a call to *IoCompleteRequest*(). This is incorrect since this does not allow other devices in the stack to respond to this IRP.

According to the DDK documentation [17], the driver has the option of failing this IRP if one of the following conditions is true:

- The device is enabled for remote wakeup (i.e., has a wait/wake IRP pending), but the system state is lower in which the device supports remote wakeup (as indicated in the device capabilities). For this reason, the author recommends canceling pending wait/wake IRPs on system resume, and for system suspend, generate a fresh wait/wake IRP only if the device is capable of waking the system from that suspend state.
- Going into the sleep state will cause a loss of data (DDK example: modem loses connection). The application often will handle this situation.

The author recommends that device drivers only issue wait/wake IRPs when the system enters a sleep state (or when self-suspending for power savings) and cancelling that IRP upon system resume (assuming that it was not responsible for waking the system).

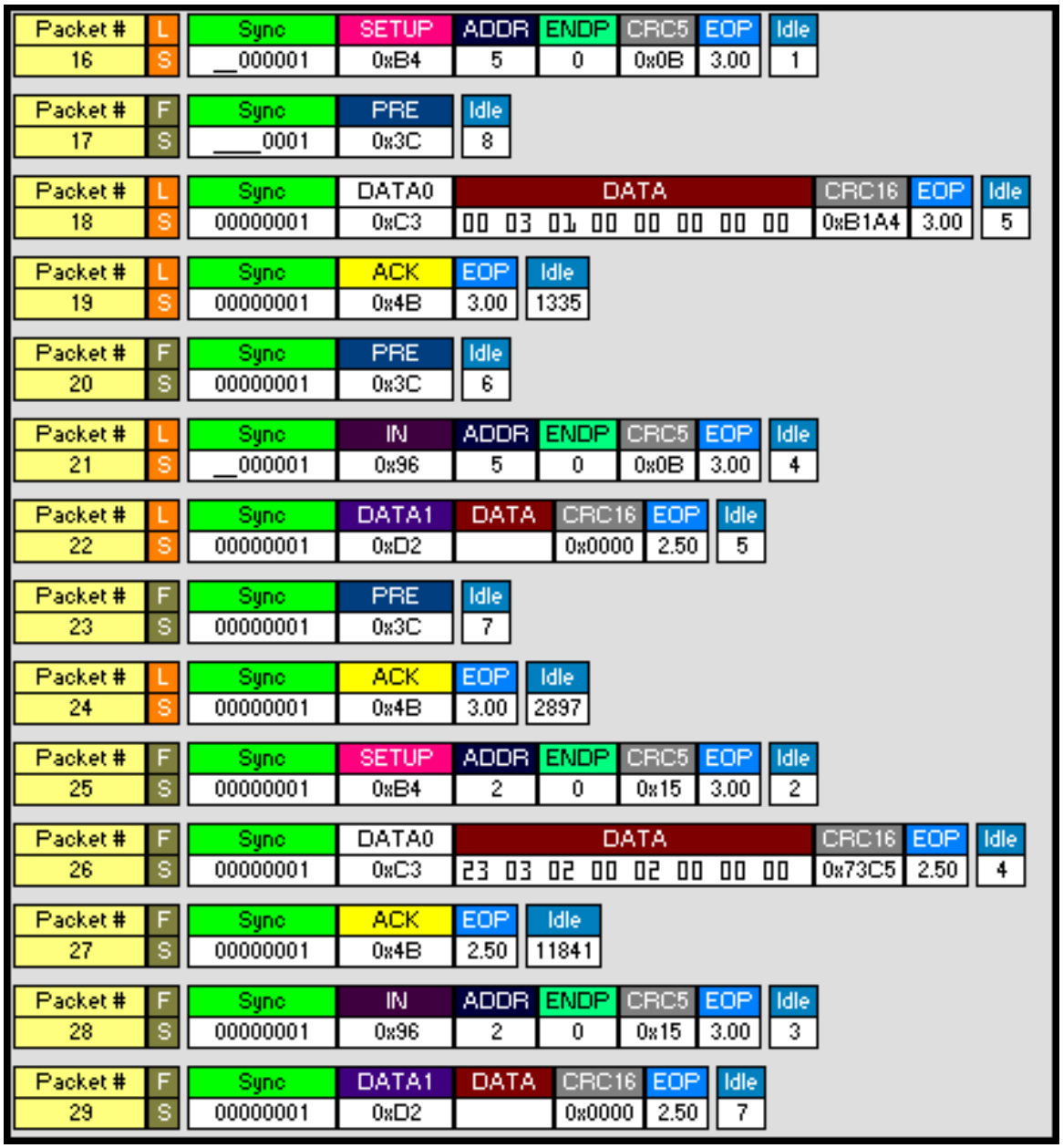
Ideally, when receiving a query IRP, the device driver would gracefully stop transmitting data, queue any incoming IRPs [18], and pass the query IRP down the stack. However, there are conditions in which this may not be the desired action. It is most important that the driver handles the query IRP in a manner that prevents blue screens and that satisfies the expectations of customers.

It should be noted that failing a query IRP might not prevent the system from following up with a set power IRP to the designated sleep state (for example, in a notebook where the OS is shutting down due to loss of power). Therefore, the author will repeat his recommendation for the driver to gracefully stop transmitting data, queue any incoming IRPs, and pass this IRP (except in extreme cases such as the DDK example). See [17] for more information.

IRP_MN_WAIT_WAKE

Device drivers of remote wakeup capable devices generate this optional IRP [13] to notify the system that the driver is capable of waking itself and/or the system from a sleep state, specifying a completion routine which is called if the device actually wakes up (drives "K" state on USB). The driver must issue this request while the device is in *D0*, since the system will issue an set device feature (remote-wakeup) before the device is set to a lower device power state, as seen in the CATC™ trace shown on the next page.

If a device is to wakeup the system, the driver must issue a wait/wake IRP, and a set device power IRP to a sleep state when receiving a system set power IRP to a system sleep state.



Packet #	L/S	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle
16	S	__000001	0x2D	5	0	0x0B	3.00	1

Packet #	F/S	Sync	PRE	Idle
17	S	____0001	0x3C	8

Packet #	L/S	Sync	DATA0	DATA	CRC16	EOP	Idle
18	S	00000001	0xC3	00 03 01 00 00 00 00 00	0xB1A4	3.00	5

Packet #	L/S	Sync	ACK	EOP	Idle
19	S	00000001	0x4B	3.00	1335

Packet #	F/S	Sync	PRE	Idle
20	S	00000001	0x3C	6

Packet #	L/S	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
21	S	__000001	0x96	5	0	0x0B	3.00	4

Packet #	L/S	Sync	DATA1	DATA	CRC16	EOP	Idle
22	S	00000001	0xD2		0x0000	2.50	5

Packet #	F/S	Sync	PRE	Idle
23	S	00000001	0x3C	7

Packet #	L/S	Sync	ACK	EOP	Idle
24	S	00000001	0x4B	3.00	2897

Packet #	F/S	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle
25	S	00000001	0x2D	2	0	0x15	3.00	2

Packet #	F/S	Sync	DATA0	DATA	CRC16	EOP	Idle
26	S	00000001	0xC3	23 03 02 00 02 00 00 00	0x73C5	2.50	4

Packet #	F/S	Sync	ACK	EOP	Idle
27	S	00000001	0x4B	2.50	11841

Packet #	F/S	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
28	S	00000001	0x96	2	0	0x15	3.00	3

Packet #	F/S	Sync	DATA1	DATA	CRC16	EOP	Idle
29	S	00000001	0xD2		0x0000	2.50	7

CATC™ Trace 1. Result Of Generating Wait/Wake IRP Before Self-Suspend

To generate a wait/wake IRP, the driver must call *PoRequestPowerIRP*() [19], as shown below. The following code will cause the Power Manager to dispatch a wait/wake IRP to the power dispatch routine.

```
powerState.SystemState = deviceExtension->SystemWake;
ntStatus = PoRequestPowerIrp(deviceExtension->PhysicalDeviceObject,
                             IRP_MN_WAIT_WAKE,
                             powerState,
                             RequestWaitWakeCompletion,
                             DeviceObject,
                             &deviceExtension->WaitWakeIrp);
```

Sample Code 7. Generating a Wait/Wake IRP

Handling wait/wake IRPs in the power dispatch routine is simple. The driver simply passes the IRP down the stack. Note that no completion routine is specified in the sample code below. However, if the driver writer wishes to include a power completion routine different than the one used when generating the wait/wake IRP, that power completion routine will be called before the completion routine corresponding to the wait/wake IRP generation call.

```
case IRP_MN_WAIT_WAKE:
    KdPrint(("=======================================\n"));
    KdPrint(("Power() Enter IRP_MN_WAIT_WAKE --\n"));
    // Optional IRP – generated by PoRequestPowerIRP
    // No completion routine is attached here since we already have one when we
    // generated the IRP
    IoSkipCurrentIrpStackLocation(Irp);
    PoStartNextPowerIrp(Irp);
    ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);
    break;
```

Sample Code 8. Handling a Wait/Wake IRP

It is important that developers of remote wakeup device be aware that an issue exists in *Windows® 98, Windows 98® SE,* and *Windows 98® ME* in which the wait/wake completion routine of the driver that issued the IRP is not called. This means that a device returns to a full power state, but the driver will not be aware of this transition to this state. This issue does not exist in *Windows® 2000*.

To workaround this issue, the author's recommendation is for only non-wakeup devices to suspend themselves when not in use for the benefit of potential power savings.

For system suspend, wakeup devices should issue a wait/wake IRP, then self-suspend before passing down the system set power IRP. For system resume, a wakeup device should cancel the pending wait/wake IRP, and then check the corresponding device if it woke up the system.

In *Windows® 2000*, this issue does not exist. Therefore, the driver should check the device only if the operating system is *Windows® 98 SE* and earlier, or *Windows® 98 ME*. To do this, the driver needs to acquire the version of the USB Driver Interface (`USBDI`). However, as noted below, this value is the same for *Windows® 98 ME* and *Windows® 2000*. In this case, the driver must determine which WDM version is supported.

```
BOOLEAN gHasRemoteWakeupIssue;
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
    NTSTATUS ntStatus = STATUS_SUCCESS;

    KdPrint (("entering (PM) DriverEntry (build time/date: %s/%s\n",__TIME__,__DATE__));
    KdPrint (("Registry path is %ws\n", RegistryPath->Buffer));

    // called when Ring 3 app calls CreateFile()
    DriverObject->MajorFunction[IRP_MJ_CREATE] = Create;

    // called when Ring 3 app calls CloseHandle()
    DriverObject->MajorFunction[IRP_MJ_CLOSE]  = Close;
    DriverObject->DriverUnload = Unload;

    // called when Ring 3 app calls DeviceIoCtrl
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ProcessIoctl;

    // handle WMI irps
    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = DispatchWMI;

    DriverObject->MajorFunction[IRP_MJ_PNP]   = DispatchPnP;
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
    // called when device is plugged in
    DriverObject->DriverExtension->AddDevice  = PnPAddDevice;

    // determine the os version and store in a global.
    USBD_GetUSBDIVersion(&gVersionInformation);

    KdPrint (("USBDI Version = 0x%x", gVersionInformation.USBDI_Version));

    // Note: the WDM major, minor version for Win98 ME is 1, 05 respectively
    gHasRemoteWakeupIssue =
                 ((gVersionInformation.USBDI_Version < USBD_WIN2K_WIN98_ME_VERSION) ||
                 ((gVersionInformation.USBDI_Version == USBD_WIN2K_WIN98_ME_VERSION)  &&
                 (IoIsWdmVersionAvailable((UCHAR)1, (UCHAR)05))));

    gRegistryPath.Buffer = (PWSTR)ExAllocatePool(NonPagedPool, RegistryPath->Length);

    ntStatus =  gRegistryPath.Buffer != NULL ? STATUS_SUCCESS : STATUS_NO_MEMORY;

    if (NT_SUCCESS(ntStatus))
    {
        RtlCopyMemory(gRegistryPath.Buffer,
                      RegistryPath->Buffer,
                      RegistryPath->Length);

        gRegistryPath.Length        = RegistryPath->Length;
        gRegistryPath.MaximumLength = RegistryPath->MaximumLength;
    }

    KdPrint (("exiting (PM) DriverEntry (%x)\n", ntStatus));
    return ntStatus;
}
```

Sample Code 9. Driver Entry Routine

The `USBDIVersion` has the following values for each of the following operating system values. Note that *Windows 98™ ME* and *Windows™ 2000* have the same `USBDIVersion` value. Therefore, if the `USBDIVersion` value is 0x300, then the driver must query if WDM version 1.05 is supported which corresponds to *Windows™ 98 ME*.

Operating System	USBDIVersion Value
Windows® 98	0x0101
Windows® 98 Second Edition	0x0200
Windows® 2000	0x0300
Windows® 98 ME	0x0300

Table 5. USBDIVersion Values

IRP_MN_SET_POWER

The Power Manager uses this IRP [20] to notify drivers of a system power state change. Also, drivers generate this IRP to change their device power state. Therefore the policy owner for set system power IRPs is the power manager while the device is the policy owner of set device power IRPs. There are two exceptions in which the power manager will issue a set device power IRP. First, the power manager will issue this IRP if the driver has been registered for idle detection that has timed out. Second, if a device is suspended when it is disconnected, the power manager will dispatch a set device power IRP to *PowerDeviceD0* before dispatching the PnP IRPs involved with device removal. It is important to note that for system power IRPs, the driver does not have the option to fail the IRP. The set system power IRP is therefore used as notification.

The following code is written to accommodate devices that may or may not support remote wakeup. Due to the issue with the wait/wake completion routine, extra code has been added changing system power states.

When the system is going into suspend, a wait/wake IRP is generated. If the operating system version is any version of *Windows® 98*, a system thread (a work item can be used instead) should be created. The purpose of this system thread (or work item) is to communicate with the device after system resume to determine whether the device caused a remote wakeup event by issuing a vendor specific command to endpoint zero. After the system and device have resumed to their full power states, the driver signals an event on which the system thread has blocked. The system thread then queries the device with a vendor specific command. In the sample code below, the system thread simply performs a get device descriptor call.

```
case IRP_MN_SET_POWER:
    KdPrint(("Power() Enter IRP_MN_SET_POWER\n"));
    switch (irpStack->Parameters.Power.Type)
    {
    case SystemPowerState:
        ntStatus = HandleSetSytemPowerState(DeviceObject, Irp);
        break;

    case DevicePowerState:
        ntStatus = HandleSetDevicePowerState(DeviceObject, Irp);
        break;
    }
    break;
```

Sample Code 10. Handling IRP_MN_SET_POWER IRPs

```
NTSTATUS
HandleSetSytemPowerState(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp)
{
    PDEVICE_EXTENSION  deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS           ntStatus        = STATUS_SUCCESS;
    POWER_STATE        sysPowerState;

    // Get input system power state
    sysPowerState.SystemState = irpStack->Parameters.Power.State.SystemState;

    KdPrint(("Power() Set Power, type SystemPowerState = %s\n",
                    SystemPowerStateString[sysPowerState.SystemState] ));

    // If system is in working state always set our device to D0
    //  regardless of the wait state or system-to-device state power map
    if (sysPowerState.SystemState ==  PowerSystemWorking)
    {
        deviceExtension->NextDeviceState.DeviceState = PowerDeviceD0;
        KdPrint(("Power() PowerSystemWorking, will set D0, not use state map\n"));

        // cancel the pending wait/wake irp
        if (deviceExtension->WaitWakeIrp)
        {
            BOOLEAN bCancel = IoCancelIrp(deviceExtension->WaitWakeIrp);

            ASSERT(bCancel);
        }
    }
    else // powering down
    {
        NTSTATUS ntStatusIssueWW = STATUS_INVALID_PARAMETER; // assume that we won't be
                                                             // able to wake the system

        // issue a wait/wake irp if we can wake the system up from this system state
        // for devices that do not support wakeup, the system wake value will be
        // PowerSystemUnspecified == 0
        if (sysPowerState.SystemState <= deviceExtension->DeviceCapabilities.SystemWake)
        {
            ntStatusIssueWW = IssueWaitWake(DeviceObject);
        }


        if (NT_SUCCESS(ntStatusIssueWW) || ntStatusIssueWW == STATUS_PENDING)
        {
            // Find the device power state equivalent to the given system state.
            // We get this info from the DEVICE_CAPABILITIES struct in our device
            // extension (initialized in PnPAddDevice() )
            deviceExtension->NextDeviceState.DeviceState =
              deviceExtension->DeviceCapabilities.DeviceState[sysPowerState.SystemState];

            KdPrint(("Power() IRP_MN_WAIT_WAKE issued, will use state map\n"));

            if (gHasRemoteWakeupIssue)
            {
                StartThread(DeviceObject);
            }
        }
        else
        {
            // if no wait pending and the system's not in working state, just turn off
            deviceExtension->NextDeviceState.DeviceState = PowerDeviceD3;

            KdPrint(("Power() Setting PowerDeviceD3 (off)\n"));
        }
    } // powering down
```

```
        KdPrint(("Current Device State: %s\n",
            DevicePowerStateString[deviceExtension->CurrentDeviceState.DeviceState]));
        KdPrint(("Next Device State:    %s\n",
               DevicePowerStateString[deviceExtension->NextDeviceState.DeviceState]));

        // We've determined the desired device state; are we already in this state?
        if (deviceExtension->NextDeviceState.DeviceState !=
                            deviceExtension->CurrentDeviceState.DeviceState)
        {
            // attach a completion routine to change the device power state
            IoCopyCurrentIrpStackLocationToNext(Irp);
            IoSetCompletionRoutine(Irp,
                                   PoChangeDeviceStateRoutine,
                                   DeviceObject,
                                   TRUE,
                                   TRUE,
                                   TRUE);
        }
        else
        {
            // Yes, just pass it on to PDO (Physical Device Object)
            IoSkipCurrentIrpStackLocation(Irp);
        }
        PoStartNextPowerIrp(Irp);
        ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);

        KdPrint(("Power() Exit IRP_MN_SET_POWER (system) with ntStatus 0x%x\n", ntStatus));

        return ntStatus;
    }
```

Sample Code 11. Handle System Set Power Irps

```
    NTSTATUS
    HandleSetDevicePowerState(
        IN PDEVICE_OBJECT DeviceObject,
        IN PIRP           Irp
        )
    {
        PDEVICE_EXTENSION  deviceExtension = DeviceObject->DeviceExtension;
        PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation(Irp);
        NTSTATUS           ntStatus        = STATUS_SUCCESS;

        KdPrint(("Power() Set Power, type DevicePowerState = %s\n",
            DevicePowerStateString[irpStack->Parameters.Power.State.DeviceState]));

        IoCopyCurrentIrpStackLocationToNext(Irp);
        IoSetCompletionRoutine(Irp,
               PoSetDevicePowerStateComplete,
               // Always pass FDO to completion routine as its Context;
               // This is because the DriverObject passed by the system to the routine
               // is the Physical Device Object (PDO) not the Functional Device Object (FDO)
               DeviceObject,
               TRUE,            // invoke on success
               TRUE,            // invoke on error
               TRUE);           // invoke on cancellation of the Irp

        PoStartNextPowerIrp(Irp);
        ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);

        KdPrint(("Power() Exit IRP_MN_SET_POWER (device) with ntStatus 0x%x\n", ntStatus));
        return ntStatus;
}
```

Sample Code 12. Handle Device Set Power Irps

```
NTSTATUS
IssueWaitWake(
    IN PDEVICE_OBJECT   DeviceObject
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS ntStatus;
    POWER_STATE powerState;

    KdPrint (("*******************************\n"));
    KdPrint(("IssueWaitWake: Entering\n"));

    if (deviceExtension->WaitWakeIrp != NULL)
    {
        KdPrint(("Wait wake all ready active!\n"));
        return STATUS_INVALID_DEVICE_STATE;
    }

    // Make sure we are capable of waking the machine
    if (deviceExtension->SystemWake <= PowerSystemWorking)
      return STATUS_INVALID_DEVICE_STATE;

    // Send IRP to request wait wake and add a pending IRP flag
    powerState.SystemState = deviceExtension->SystemWake;

    ntStatus = PoRequestPowerIRP(deviceExtension->PhysicalDeviceObject,
                                 IRP_MN_WAIT_WAKE,
                                 powerState,
                                 RequestWaitWakeCompletion,
                                 DeviceObject,
                                 &deviceExtension->WaitWakeIrp);
    if (!deviceExtension->WaitWakeIrp)
    {
        KdPrint(("Wait wake is NULL! (0x%x)\n", ntStatus));
        NtStatus = STATUS_UNSUCCESSFUL;
    }
    KdPrint(("IssueWaitWake: exiting with ntStatus 0x%x\n",
                ntStatus));
    return ntStatus;
}
```

Sample Code 13. Issue Wait/Wake Function

```
NTSTATUS
PoChangeDeviceStateRoutine(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    )
{
    PDEVICE_OBJECT     DeviceObject     = (PDEVICE_OBJECT)Context;
    PDEVICE_EXTENSION  deviceExtension  = DeviceObject->DeviceExtension;
    NTSTATUS           RequestIrpStatus = STATUS_SUCCESS;
    DEVICE_POWER_STATE currDeviceState  =
                        deviceExtension->CurrentDeviceState.DeviceState;
    DEVICE_POWER_STATE nextDeviceState  = deviceExtension->NextDeviceState.DeviceState;

    KdPrint(("Change device state from %s to %s\n",
                DevicePowerStateString[currDeviceState],
                DevicePowerStateString[nextDeviceState]));

    //  If the lower driver returned PENDING, mark our stack location as pending also.
    if (Irp->PendingReturned)
    {
        IoMarkIrpPending(Irp);
    }

    // No, request that we be put into this state
    // by requesting a new Power Irp from the Pnp manager
    deviceExtension->PowerIrp = Irp;
```

```
    if (deviceExtension->NextDeviceState.DeviceState == PowerDeviceD0)
    {
        KdPrint(("Powering up device as a result of system wakeup\n"));
    }

    deviceExtension->SetPowerEventFlag =
            ((deviceExtension->NextDeviceState.DeviceState == PowerDeviceD0)    &&
             (deviceExtension->CurrentDeviceState.DeviceState != PowerDeviceD0) &&
             (gHasRemoteWakeupIssue));

    // simply adjust the device state if necessary
    if (currDeviceState != nextDeviceState)
    {
        deviceExtension->IoctlIrp = NULL;
        RequestIrpStatus = SetDevicePowerState(DeviceObject,
                                               nextDeviceState);
    }
    Irp->IoStatus.Status = STATUS_SUCCESS;

    return STATUS_SUCCESS;
}
```

Sample Code 14. Set Device Power Completion Routine Due To Set System Power

```
NTSTATUS
PoSetDevicePowerStateComplete(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    ) DeviceState
{
    PDEVICE_OBJECT     deviceObject    = (PDEVICE_OBJECT) Context;
    PDEVICE_EXTENSION  deviceExtension = deviceObject->DeviceExtension;
    NTSTATUS           ntStatus        = Irp->IoStatus.Status;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation (Irp);
    DEVICE_POWER_STATE deviceState     = irpStack->Parameters.Power.State.;

    KdPrint(("enter PoSetDevicePowerStateComplete\n"));
```

```
    // If the lower driver returned PENDING, mark our stack location as pending also.
    if (Irp->PendingReturned)
    {
        IoMarkIrpPending(Irp);
    }
    if (NT_SUCCESS(ntStatus))
    {
        KdPrint(("Updating current device state to %s\n",
                 DevicePowerStateString[deviceState]));
        deviceExtension->CurrentDeviceState.DeviceState = deviceState;
    }
    else
    {
        KdPrint(("Error: Updating current device state to %s failed.  NTSTATUS = 0x%x\n",
                 DevicePowerStateString[deviceState],
                 ntStatus));
    }
    KdPrint(("exit PoSetDevicePowerStateComplete\n"));
    return ntStatus;
}
```

Sample Code 15. Completion Routine For Changing Device Power State

```
NTSTATUS
StartThread(
    IN PDEVICE_OBJECT DeviceObject
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS ntStatus;
    HANDLE ThreadHandle;


    ntStatus = PsCreateSystemThread(&ThreadHandle,
                                    (ACCESS_MASK)0,
                                    NULL,
                                    (HANDLE) 0,
                                    NULL,
                                    PowerUpThread,
                                    DeviceObject);

    if (!NT_SUCCESS(ntStatus))
    {
        return ntStatus;
    }
    //
    // Convert the Thread object handle
    // into a pointer to the Thread object
    // itself. Then close the handle.
    //
    ObReferenceObjectByHandle(ThreadHandle,
                              THREAD_ALL_ACCESS,
                              NULL,
                              KernelMode,
                              &deviceExtension->ThreadObject,
                              NULL);

    ZwClose( ThreadHandle );

    return ntStatus;
}
```

Sample Code 16. Start Thread routine

```
VOID
PowerUpThread(
       IN PVOID pContext
       )
{
    PDEVICE_OBJECT DeviceObject = pContext;
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;

    PIO_STACK_LOCATION IRPStack;
    PLIST_ENTRY ListEntry;
    PIRP IRP;
    PVOID ioBuffer;
    NTSTATUS ntStatus;

    KeSetPriorityThread(KeGetCurrentThread(),LOW_REALTIME_PRIORITY );

    KdPrint((
       "PowerUpThread: System Thread Started With DeviceObject: 0x%x, devExt: 0x%x\n",
                   DeviceObject,
                   deviceExtension));

    ntStatus = KeWaitForSingleObject(&deviceExtension->PowerUpEvent,
                                     Suspended,
                                     KernelMode,
                                     FALSE,
                                     NULL);
    if (NT_SUCCESS(ntStatus))
    {
        PVOID descriptorBuffer  = NULL;
        ULONG siz;
        KdPrint(("Power Up Event signalled - Performing get descriptor call\n"));

        // Here the driver would issue a vendor specific command
        // if it was determined that the device did wake up the system
        // then the necessary functions for handling that event would be called

        // Perform a Get device Descriptor in place of the vendor specific command
        siz = sizeof(USB_DEVICE_DESCRIPTOR);
        descriptorBuffer = ExAllocatePool(NonPagedPool, siz);

        if (!descriptorBuffer)
        {
            ntStatus = STATUS_NO_MEMORY;
        }
        else
        {
            ntStatus = GetDescriptor(DeviceObject,
                                     USB_DEVICE_DESCRIPTOR_TYPE,
                                     0,
                                     0,
                                     descriptorBuffer,
                                     siz);
            ExFreePool(descriptorBuffer);
            descriptorBuffer = NULL;
        }
        KdPrint(("PowerUpThread: Get Descriptor Call: 0x%x\n", ntStatus));
    }
    KdPrint(("PowerUpThread - Terminating\n"));
    PsTerminateSystemThread( STATUS_SUCCESS );
}
```

Sample Code 17. System thread for querying a device

An issue exists with the early Intel host controllers in which if a root port is placed in suspend; it may not react correctly to a remote wakeup event (this issue does not exist on currently shipping Intel host controllers). To work around this, the system will pass all set device power requests, however, the device will not physically be put to sleep. However, this

does not mean that devices should not attempt to save power by self-suspending when not in use. The latest Intel host controllers and all other host controllers will have their root ports suspended due to a device requesting to be suspended.

Generating Power IRPs

This section covers the three of the four IRPs that a device driver may generate. Generating an IRP_MN_QUERY_POWER IRP (for a device) is not covered in this section.

IRP_MN_POWER_SEQUENCE

To generate a `IRP_MN_POWER_SEQUENCE` IRP, the driver calls *IoAllocateIRP*() to allocate the IRP, then calls *PoCallDriver*() to pass the IRP down the stack. This paper defers to reader to [12] for more information.

IRP_MN_SET_POWER (Device only)

A device driver can only generate an IRP_MN_SET_POWER IRP for the device, not the system. In other words, a driver may suspend its device but not the whole system. To generate a set device power IRP, the driver calls *PoRequestPowerIRP*() as shown in the sample code below.

```
NTSTATUS
NTSTATUS
SetDevicePowerState(
    IN PDEVICE_OBJECT     DeviceObject,
    IN DEVICE_POWER_STATE DevicePowerState)
{
    PDEVICE_EXTENSION   deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS            ntStatus        = STATUS_SUCCESS;
    POWER_STATE         powerState;

    powerState.DeviceState = DevicePowerState;

    ntStatus = PoRequestPowerIrp(deviceExtension->PhysicalDeviceObject,
                                 IRP_MN_SET_POWER,
                                 powerState,
                                 ChangeDevicePowerStateCompletion,
                                 DeviceObject,
                                 NULL);

    return ntStatus;
}
```

Sample Code 18. Generating a Set Device Power IRP

```
NTSTATUS
ChangeDevicePowerStateCompletion(
    IN PDEVICE_OBJECT       DeviceObject,
    IN UCHAR                MinorFunction,
    IN POWER_STATE          PowerState,
    IN PVOID                Context,
    IN PIO_STATUS_BLOCK     IoStatus
    )
{
    PDEVICE_OBJECT    deviceObject    = (PDEVICE_OBJECT)Context;
    PDEVICE_EXTENSION deviceExtension = deviceObject->DeviceExtension;
    NTSTATUS          ntStatus        = STATUS_SUCCESS;
    PIRP              irp             = deviceExtension->IoctlIrp;
```

```
    if (irp)
    {
        IoCompleteRequest(irp, IO_NO_INCREMENT);
    }
    if (deviceExtension->SetPowerEventFlag)
    {
        // since we are powering up, set the event
        // so that the thread can query the device to see
        // if it generated the remote wakeup
        KdPrint(("Signal Power Up Event for Win98 Gold/SE/ME\n"));
        KeSetEvent(&deviceExtension->PowerUpEvent, 0, FALSE);
        deviceExtension->SetPowerEventFlag = FALSE;
    }

    KdPrint(("Exiting ChangeDevicePowerStateCompletion\n"));

    return ntStatus;
}
```

Sample Code 19. Completion for Driver Generated Set (Device) Power IRP

IRP_MN_WAIT_WAKE

To generate a wait/wake IRP, the driver calls *PoRequestPowerIRP*(). The code below is the same code as shown in **Sample Code 7**.

```
powerState.SystemState = deviceExtension->SystemWake;
ntStatus = PoRequestPowerIrp(deviceExtension->PhysicalDeviceObject,
                             IRP_MN_WAIT_WAKE,
                             powerState,
                             RequestWaitWakeCompletion,
                             DeviceObject,
                             &deviceExtension->WaitWakeIrp);
```

Sample Code 20. Generating a Wait/Wake IRP

IRP Sequences

This section discusses the sequences of IRP that device's see during system/device suspend/resume events as seen in the provided sample code in this paper.

System Suspend

When the system goes into any sleep state, a query power IRP is broadcasted to all the drivers. Assuming that all drivers pass the query IRP, the system broadcasts set power IRPs (of type system). If the device is capable of waking the system from this power state, the driver issues a wait/wake IRP. Finally, the device driver then generates a set power IRP for the device to place it in the appropriate device sleep state.

System Resume

When a system resumes, a query IRP is not dispatched since all devices support the *S0* state. The system issues a system set power IRP. In response, a driver may cancel its pending wait/wake IRP. The driver then generates a set power IRP to set the device to *D0*.

System Resume due to Device Wakeup (Windows® 98 Gold/SE/ME)

Due to the wait/wake issue as previously discussed, the sequence of events is the same as a system resume. The driver should then query its device if it generated a wakeup event.

System Resume due to Device Wakeup (Windows® 2000)

When a USB device wakes up the system, the wait/wake completion routine is called before the system is powered back to *S0*. The completion routine then generates a set power IRP for the device. The system will then wake up the system with a system set power IRP.

System Event	Code Sequence
System Suspend	1. `IRP_MN_QUERY_POWER` IRP 2. System-generated IRP_MN_SET_POWER IRP 3. Device-generated IRP_MN_SET_POWER IRP
System Resume	1. System-generated IRP_MN_SET_POWER IRP 2. Wait/wake completion (cancelled) 3. Device-generated IRP_MN_SET_POWER IRP
System Resume due to device wakeup (Windows® 98 Gold/SE/ME)	1. System-generated IRP_MN_SET_POWER 2. Wait/wake completion (cancelled) 3. Driver-generated IRP_MN_SET_POWER
System Wakeup due to device wakeup (Windows® 2000)	1. Wait/wake completion (success) 2. Driver-generated IRP_MN_SET_POWER 3. System-generated IRP_MN_SET_POWER

Table 6. Code Sequences for System Power Management Events

Device Suspend

For a device to suspend itself, the driver simply generates a set power IRP to a device sleep state.

Device Resume

For a device to resume itself, it simply issues a set power IRP to *PowerDeviceD0*.

Device Wakeup (Windows® 2000)

When a device wakes itself, the wait/wake completion routine is called which generates a set power IRP to *PowerDeviceD0*.

Device Event	Code Sequence
Device Suspend	Device-generated IRP_MN_SET_POWER IRP
Device Resume	Device-generated IRP_MN_SET_POWER IRP
Device Wakeup (*Windows® 2000*)	1. Wait/wake completion routine 2. Device-generated IRP_MN_SET_POWER IRP

Table 7. Code Sequences for Device Power Management Events

Worst Case Scenario

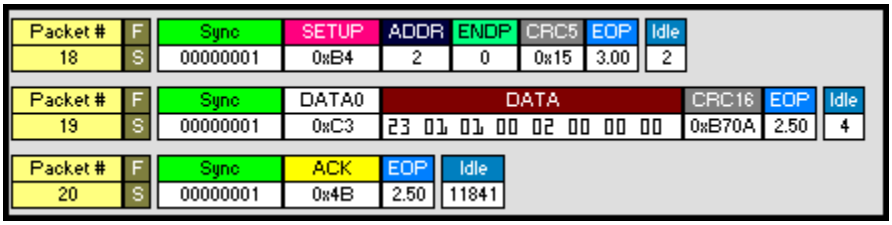
The driver must also accommodate the worst-case scenario for system suspend: the USB host controller is turned off. The driver recognizes this scenario by a set system power IRP to a suspend state, followed by a PnP remove IRP, and finally an unload IRP. The driver must gracefully handle this sequence of IRPs.

Improper Power Management Consequences

When any device receives a set system power IRP, it should set its device power to the appropriate state (as specified in the device capabilities structure saved in the device extension).

If this is not implemented (i.e., the driver simply passes the power IRP down the stack), during system suspend, the device will receive the PnP `IRP_MN_REMOVE_DEVICE` IRP followed by an `IRP_MJ_UNLOAD`. This means that the driver is actually unloaded from the system. On resume, the driver will be reloaded with the `DRIVER_ENTRY` routine being executed which is followed by the normal PnP IRPs that are dispatched during enumeration (i.e. `IRP_MN_START_DEVICE`, `IRP_MN_QUERY_CAPABILITIES`, etc.). If an application has a handle to the driver, that handle will be valid when the system resumes.

Using a CATC™ bus analyzer, one will see the device's parent port being disabled before the system is suspended; hence the driver also receives PnP remove IRP. As stated above, an unload IRP will be dispatched to the driver as well.



Packet #	F	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle
18	S	00000001	0xB4	2	0	0x15	3.00	2

Packet #	F	Sync	DATA0	DATA	CRC16	EOP	Idle
19	S	00000001	0xC3	23 01 01 00 02 00 00 00	0xB70A	2.50	4

Packet #	F	Sync	ACK	EOP	Idle
20	S	00000001	0x4B	2.50	11841

CATC™ Trace 2. Disabling Parent Port Of Device

Appendix – Source Code

```
// ************************************
//
// File: sample.c
//
// ************************************
#define DRIVER

#define INITGUID

#pragma warning(disable:4214) //  bitfield nonstd
#include "wdm.h"
#pragma warning(default:4214)

#include "stdarg.h"
#include "stdio.h"

#pragma warning(disable:4200) //non std struct used
#include "usbdi.h"
#pragma warning(default:4200)

#include <initguid.h>
#include "guid.h"
#include "usbdlib.h"
#include "ioctl.h"
#include "sample.h"
#include "pnp.h"
#include "power.h"

USBD_VERSION_INFORMATION gVersionInformation;
BOOLEAN gHasRemoteWakeupIssue;

#define ULONG_PTR PULONG

#pragma alloc_text(PAGE, PnPAddDevice)

UCHAR *SystemCapString[] = {
   "PowerSystemUnspecified",
   "PowerSystemWorking",
   "PowerSystemSleeping1",
   "PowerSystemSleeping2",
   "PowerSystemSleeping3",
   "PowerSystemHibernate",
   "PowerSystemShutdown",
   "PowerSystemMaximum"
};

UCHAR *DeviceCapString[] = {
   "PowerDeviceUnspecified",
   "PowerDeviceD0",
   "PowerDeviceD1",
   "PowerDeviceD2",
   "PowerDeviceD3",
   "PowerDeviceMaximum"
};

UNICODE_STRING gRegistryPath;
```

```
// ****************************************************************
// Function: DriverEntry
// Purpose:  Initializes dispatch routine for
//           driver object
// ****************************************************************
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
    NTSTATUS ntStatus = STATUS_SUCCESS;

    KdPrint (("entering (PM) DriverEntry (build time/date: %s/%s\n",__TIME__,__DATE__));
    KdPrint (("Registry path is %ws\n", RegistryPath->Buffer));

    // called when Ring 3 app calls CreateFile()
    DriverObject->MajorFunction[IRP_MJ_CREATE] = Create;

    // called when Ring 3 app calls CloseHandle()
    DriverObject->MajorFunction[IRP_MJ_CLOSE]  = Close;
    DriverObject->DriverUnload = Unload;

    // called when Ring 3 app calls DeviceIoCtrl
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ProcessIoctl;

    // handle WMI irps
    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = DispatchWMI;

    DriverObject->MajorFunction[IRP_MJ_PNP]   = DispatchPnP;
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
    // called when device is plugged in
    DriverObject->DriverExtension->AddDevice  = PnPAddDevice;

    // determine the os version and store in a global.
    USBD_GetUSBDIVersion(&gVersionInformation);

    KdPrint (("USBDI Version = 0x%x", gVersionInformation.USBDI_Version));

    // Note: the WDM major, minor version for Win98 ME is 1, 05 respectively
    gHasRemoteWakeupIssue =
                  ((gVersionInformation.USBDI_Version < USBD_WIN2K_WIN98_ME_VERSION) ||
                  ((gVersionInformation.USBDI_Version == USBD_WIN2K_WIN98_ME_VERSION)  &&
                  (IoIsWdmVersionAvailable((UCHAR)1, (UCHAR)05))));

    gRegistryPath.Buffer = (PWSTR)ExAllocatePool(NonPagedPool, RegistryPath->Length);

    ntStatus =  gRegistryPath.Buffer != NULL ? STATUS_SUCCESS : STATUS_NO_MEMORY;

    if (NT_SUCCESS(ntStatus))
    {
        RtlCopyMemory(gRegistryPath.Buffer,
                      RegistryPath->Buffer,
                      RegistryPath->Length);

        gRegistryPath.Length        = RegistryPath->Length;
        gRegistryPath.MaximumLength = RegistryPath->MaximumLength;
    }

    KdPrint (("exiting (PM) DriverEntry (%x)\n", ntStatus));
    return ntStatus;
}
```

```
// ********************************************************************
// Function: GenericCompletion
// Purpose:  Simply sets an event set by a thread that attached this
//           routine to an irp
// ********************************************************************
NTSTATUS
GenericCompletion(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PKEVENT Event)
{
   KeSetEvent(Event, IO_NO_INCREMENT, FALSE);
   return STATUS_MORE_PROCESSING_REQUIRED;
}

// ********************************************
// Function: DispatchWMI
// Purpose:  Handle any WMI irps
// ********************************************
NTSTATUS
DispatchWMI(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{
    PDEVICE_EXTENSION   deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS            ntStatus;

    KdPrint (("WMI: Entering\n"));

    IoSkipCurrentIrpStackLocation(Irp);
    ntStatus = IoCallDriver(deviceExtension->StackDeviceObject, Irp);

    return ntStatus;

}

// ********************************************
// Function: Unload
// Purpose:  Called when driver is unloaded.  Also
//           free any resources allocated in
//           DriverEntry()
// ********************************************
VOID
Unload(
    IN PDRIVER_OBJECT DriverObject
    )
{
    KdPrint (("Unload\n"));

    if (gRegistryPath.Buffer != NULL)
    {
        ExFreePool(gRegistryPath.Buffer);
        gRegistryPath.Buffer = NULL;
    }
}
```

```
// ********************************************
// Function: RemoveDevice
// Purpose:  Remove instance of a device
// ********************************************
NTSTATUS
RemoveDevice(
    IN  PDEVICE_OBJECT DeviceObject
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS          ntStatus        = STATUS_SUCCESS;
    UNICODE_STRING    deviceLinkUnicodeString;

    KdPrint (("****************************\n"));
    KdPrint (("RemoveDevice: Entering\n"));

    if (deviceExtension->WaitWakeIrp)
    {
        KdPrint(("Cleanup: Cancelling WaitWake irp\n"));
        IoCancelIrp(deviceExtension->WaitWakeIrp);
    }

    RtlInitUnicodeString (&deviceLinkUnicodeString,
                          deviceExtension->DeviceLinkNameBuffer);

    IoDeleteSymbolicLink(&deviceLinkUnicodeString);

    if (deviceExtension->ConfigurationDescriptors)
       {
        int i = 0;

        for (i = 0; i < deviceExtension->DeviceDescriptor->bNumConfigurations; i++)
        {
            if (deviceExtension->ConfigurationDescriptors[i])
            {
                ExFreePool(deviceExtension->ConfigurationDescriptors[i]);
                deviceExtension->ConfigurationDescriptors[i] = NULL;
            }
        }
        ExFreePool(deviceExtension->ConfigurationDescriptors);
        deviceExtension->ConfigurationDescriptors = NULL;
    }

    if (deviceExtension->DeviceDescriptor)
    {
        ExFreePool(deviceExtension->DeviceDescriptor);
        deviceExtension->DeviceDescriptor = NULL;
    }

    // Delete the link to the Stack Device Object, and delete the
    // Functional Device Object we created
    IoDetachDevice(deviceExtension->StackDeviceObject);
    IoDeleteDevice (DeviceObject);


    KdPrint(("RemoveDevice Exiting With ntStatus 0x%x\n", ntStatus));
    return ntStatus;
}
```

```
// *********************************************
// Function: FreeInterfaceList
// Purpose:  Free the memory allocated for the
//           interface list
// *********************************************
VOID
FreeInterfaceList(
        IN PDEVICE_OBJECT DeviceObject
        )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    ULONG ulCurrentConfigurationIndex = deviceExtension->ulCurrentConfigurationIndex;
    PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor =
                  deviceExtension->ConfigurationDescriptors[ulCurrentConfigurationIndex];
    ULONG interfaceNumber = 0;

    if (deviceExtension->InterfaceList)
    {
        for (interfaceNumber = 0;
             interfaceNumber < ConfigurationDescriptor->bNumInterfaces;
             interfaceNumber++)
        {
            ExFreePool(deviceExtension->InterfaceList[interfaceNumber]);
            deviceExtension->InterfaceList[interfaceNumber] = NULL;
        }
        ExFreePool(deviceExtension->InterfaceList);
        deviceExtension->InterfaceList = NULL;
    }

    return;
}

// *********************************************
// Function: StopDevice
// Purpose:  Unconfigure the device
// *********************************************
NTSTATUS
StopDevice(
    IN  PDEVICE_OBJECT DeviceObject
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS ntStatus = STATUS_SUCCESS;
    PURB urb;
    ULONG siz;
    ULONG length;

    KdPrint (("StopDevice: Entering\n"));

    // Send the select configuration urb with a NULL pointer for the configuration
    // handle, this closes the configuration and puts the device in the 'unconfigured'
    // state.

    siz = sizeof(struct _URB_SELECT_CONFIGURATION);

    urb = ExAllocatePool(NonPagedPool, siz);

    if (urb)
    {
        NTSTATUS status;

        UsbBuildSelectConfigurationRequest(urb,
                                           (USHORT) siz,
                                           NULL);

        status = CallUSBD(DeviceObject, urb, &length);

        KdPrint (("Device Configuration Closed status = 0x%x usb status = 0x%x.\n",
                  status,
                  urb->UrbHeader.Status));
        ExFreePool(urb);
```

```
    }
    else
    {
        ntStatus = STATUS_NO_MEMORY;
    }

    KdPrint (("StopDevice Exiting With ntStatus 0x%x\n", ntStatus));
    return ntStatus;
}

// ********************************************
// Function: PnPAddDevice
// Purpose:  Create new instance of the device
// ********************************************
NTSTATUS
PnPAddDevice(
    IN PDRIVER_OBJECT DriverObject,
    IN PDEVICE_OBJECT PhysicalDeviceObject
    )
{
    NTSTATUS                ntStatus = STATUS_SUCCESS;
    PDEVICE_OBJECT          deviceObject = NULL;
    PDEVICE_EXTENSION       deviceExtension;

    KdPrint(("PnPAddDevice: Entering\n"));

    // create our funtional device object (FDO)
    ntStatus = CreateDeviceObject(DriverObject,
                                        PhysicalDeviceObject,
                                        &deviceObject);

    if (NT_SUCCESS(ntStatus))
    {
        deviceExtension = deviceObject->DeviceExtension;
        deviceObject->Flags |= DO_POWER_PAGABLE;

        deviceObject->Flags &= ~DO_DEVICE_INITIALIZING;

        deviceExtension->PhysicalDeviceObject = PhysicalDeviceObject;

        // Attach to the StackDeviceObject.  This is the device object that what we
        // use to send Irps and Urbs down the USB software stack
        deviceExtension->StackDeviceObject =
                        IoAttachDeviceToDeviceStack(deviceObject, PhysicalDeviceObject);

        ASSERT (deviceExtension->StackDeviceObject != NULL);

        // initialize original power level as fully on
        deviceExtension->CurrentDeviceState.DeviceState = PowerDeviceD0;
        deviceExtension->CurrentSystemState.SystemState = PowerSystemWorking;

        deviceExtension->WaitWakeIrp = NULL;

        deviceExtension->PnPstate = eRemoved;
        deviceExtension->InterfaceList = NULL;

        if (gHasRemoteWakeupIssue)
        {
            // no need to initialize unless we are running on Win98 Gold/SE/ME
            KdPrint(("Initializing PowerUpEvent\n"));
            KeInitializeEvent(&deviceExtension->PowerUpEvent,
                              SynchronizationEvent,
                              FALSE);
        }
    }
    KdPrint(("PnPAddDevice Exiting With ntStatus 0x%x\n", ntStatus));
    return ntStatus;
}
```

```
// ********************************************
// Function: CreateDeviceObject
// Purpose:  Create new instance of the device
// ********************************************
NTSTATUS
CreateDeviceObject(
    IN PDRIVER_OBJECT DriverObject,
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN PDEVICE_OBJECT *DeviceObject
    )
{
    NTSTATUS ntStatus;
    UNICODE_STRING deviceLinkUnicodeString;
    PDEVICE_EXTENSION deviceExtension;

    KdPrint(("CreateDeviceObject: Enter\n"));

    ntStatus = IoRegisterDeviceInterface(PhysicalDeviceObject,
                                         (LPGUID)&GUID_CLASS_PM,
                                         NULL,
                                         &deviceLinkUnicodeString);

    if (NT_SUCCESS(ntStatus))
    {
       // IoSetDeviceInterfaceState enables or disables a previously
       // registered device interface. Applications and other system components
       // can open only interfaces that are enabled.

        ntStatus = IoSetDeviceInterfaceState(&deviceLinkUnicodeString, TRUE);
    }

    if (NT_SUCCESS(ntStatus))
    {
        ntStatus = IoCreateDevice(DriverObject,
                                  sizeof (DEVICE_EXTENSION),
                                  NULL,
                                  FILE_DEVICE_UNKNOWN,
                                  FILE_AUTOGENERATED_DEVICE_NAME,
                                  FALSE,
                                  DeviceObject);
    }
    if (NT_SUCCESS(ntStatus))
    {
        // Initialize our device extension
        deviceExtension = (PDEVICE_EXTENSION) ((*DeviceObject)->DeviceExtension);

        RtlCopyMemory(deviceExtension->DeviceLinkNameBuffer,
                      &deviceLinkUnicodeString,
                      sizeof(deviceLinkUnicodeString));

        deviceExtension->ConfigurationHandle    = NULL;
        deviceExtension->DeviceDescriptor       = NULL;

        deviceExtension->InterfaceList = NULL;
    }

    RtlFreeUnicodeString(&deviceLinkUnicodeString);

    KdPrint(("CreateDeviceObject: Exiting With ntStatus 0x%x\n", ntStatus));

    return ntStatus;
}
```

```
// ********************************************
// Function: CallUSBD
// Purpose:  Submit an USB urb
// ********************************************
NTSTATUS
CallUSBD(
    IN PDEVICE_OBJECT DeviceObject,
    IN PURB Urb,
    OUT PULONG Length
    )
{
    NTSTATUS ntStatus, status = STATUS_SUCCESS;
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    PIRP irp;
    IO_STATUS_BLOCK ioStatus;
    PIO_STACK_LOCATION nextStack;
    PKEVENT pSyncEvent = &(deviceExtension->SyncEvent);

    // issue a synchronous request (see notes above)
    KeInitializeEvent(pSyncEvent, NotificationEvent, FALSE);

    irp = IoBuildDeviceIoControlRequest(
                IOCTL_INTERNAL_USB_SUBMIT_URB,
                deviceExtension->StackDeviceObject,
                NULL,
                0,
                NULL,
                0,
                TRUE, /* INTERNAL */
                pSyncEvent,
                &ioStatus);

    // Prepare for calling the USB driver stack
    nextStack = IoGetNextIrpStackLocation(irp);
    ASSERT(nextStack != NULL);

    // Set up the URB ptr to pass to the USB driver stack
    nextStack->Parameters.Others.Argument1 = Urb;

    // Call the USB class driver to perform the operation.  If the returned status
    // is PENDING, wait for the request to complete.
    ntStatus = IoCallDriver(deviceExtension->StackDeviceObject, irp);

    if (ntStatus == STATUS_PENDING)
    {
        status = KeWaitForSingleObject(pSyncEvent,
                                       Suspended,
                                       KernelMode,
                                       FALSE,
                                       NULL);
    }
    else
    {
        ioStatus.Status = ntStatus;
    }
    ntStatus = ioStatus.Status;

    if (Length)
    {
        if (NT_SUCCESS(ntStatus))
            *Length = Urb->UrbBulkOrInterruptTransfer.TransferBufferLength;
        else
            *Length = 0;
    }

    return ntStatus;
}
```

```
// ********************************************
// Function: Create
// Purpose:  Called when user app calls CreateFile
// ********************************************
NTSTATUS
Create(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    NTSTATUS ntStatus;
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

    KdPrint (("In Create\n"));
    ntStatus = Irp->IoStatus.Status;

    IoCompleteRequest (Irp, IO_NO_INCREMENT);

    KdPrint (("Exit Create (%x)\n", ntStatus));

    return ntStatus;

}

// ********************************************
// Function: Close
// Purpose:  Called when user app calls CloseHandle
// ********************************************
NTSTATUS
Close(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS ntStatus = STATUS_SUCCESS;

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

    KdPrint(("In Close\n"));

    IoCompleteRequest (Irp, IO_NO_INCREMENT);

    KdPrint (("Exit Close (%x)\n", ntStatus));

    return ntStatus;
}
```

```
// ********************************************
// Function: GetRegistryDword
// Purpose:  Read a DWORD from the registry
// ********************************************
NTSTATUS
GetRegistryDword(
    IN      PUNICODE_STRING RegPath,
    IN      PWCHAR          ValueName,
    IN OUT  PULONG          Value
    )
{
    RTL_QUERY_REGISTRY_TABLE paramTable[2];  //zero'd second table terminates parms
    ULONG lDef = *Value;                     // default
    NTSTATUS ntStatus;

    RtlZeroMemory(paramTable, sizeof(paramTable));

    paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
    paramTable[0].Name = ValueName;
    paramTable[0].EntryContext = Value;
    paramTable[0].DefaultType = REG_DWORD;
    paramTable[0].DefaultData = &lDef;
    paramTable[0].DefaultLength = sizeof(ULONG);

    ntStatus = RtlQueryRegistryValues(RTL_REGISTRY_ABSOLUTE | RTL_REGISTRY_OPTIONAL,
                                      RegPath->Buffer,
                                      paramTable,
                                      NULL,
                                      NULL);

    KdPrint(("GetRegistryDword exiting with ntStatus 0x%x\n", ntStatus));

    return ntStatus;
}


// ********************************************
// Function: SelectInterface
// Purpose:  Change a specified interface's
//           alternate setting
// ********************************************
NTSTATUS
SelectInterface(
    PDEVICE_OBJECT DeviceObject,
    UCHAR ucInterfaceNumber,
    UCHAR ucAltSetting
    )
{
    PDEVICE_EXTENSION             deviceExtension    = DeviceObject->DeviceExtension;
    PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor =
      deviceExtension->ConfigurationDescriptors
                                     [deviceExtension->ulCurrentConfigurationIndex];
    PUSB_INTERFACE_DESCRIPTOR     interfaceDescriptor     = NULL;
    PUSB_ENDPOINT_DESCRIPTOR      endpointDescriptor      = NULL;
    PUSBD_INTERFACE_INFORMATION   interfaceObject         = NULL;

    NTSTATUS    ntStatus    = STATUS_SUCCESS;
    PURB        urb         = NULL;
    int         i           = 0;
    ULONG       siz         = 0;
    ULONG       length      = 0;

    interfaceDescriptor =
            USBD_ParseConfigurationDescriptorEx(ConfigurationDescriptor,
                                                (PVOID)ConfigurationDescriptor,
                                                ucInterfaceNumber, // 0
                                                ucAltSetting,
                                                -1,
                                                -1,
                                                -1);
```

```
if (!interfaceDescriptor)
    return STATUS_UNSUCCESSFUL;

siz = GET_SELECT_INTERFACE_REQUEST_SIZE(interfaceDescriptor->bNumEndpoints);

urb = ExAllocatePool(NonPagedPool, siz);

if (!urb)
    return STATUS_NO_MEMORY;

urb->UrbHeader.Function =  URB_FUNCTION_SELECT_INTERFACE;
urb->UrbHeader.Length = (USHORT)siz;
urb->UrbSelectInterface.ConfigurationHandle =
                                        deviceExtension->ConfigurationHandle;
urb->UrbSelectInterface.Interface.Length =sizeof(struct _USBD_INTERFACE_INFORMATION);

if (interfaceDescriptor->bNumEndpoints > 1)
{
    urb->UrbSelectInterface.Interface.Length +=(interfaceDescriptor->bNumEndpoints-1)
                                           * sizeof(struct _USBD_PIPE_INFORMATION);
}

urb->UrbSelectInterface.Interface.InterfaceNumber  = ucInterfaceNumber;
urb->UrbSelectInterface.Interface.AlternateSetting = ucAltSetting;
urb->UrbSelectInterface.Interface.Class     = interfaceDescriptor->bInterfaceClass;
urb->UrbSelectInterface.Interface.SubClass  =
                                       interfaceDescriptor->bInterfaceSubClass;
urb->UrbSelectInterface.Interface.Protocol  =
                                       interfaceDescriptor->bInterfaceProtocol;
urb->UrbSelectInterface.Interface.NumberOfPipes    =
                                       interfaceDescriptor->bNumEndpoints;
// Interface Handle is considered opaque
//urb->UrbSelectInterface.Interface.InterfaceHandle

// Fill in the Interface pipe information

interfaceObject = &urb->UrbSelectInterface.Interface;
endpointDescriptor = (PUSB_ENDPOINT_DESCRIPTOR)((ULONG)interfaceDescriptor +
                                            (ULONG)interfaceDescriptor->bLength);

for (i = 0; i < interfaceDescriptor->bNumEndpoints; i++)
{
    interfaceObject->Pipes[i].MaximumPacketSize   =
                                          endpointDescriptor->wMaxPacketSize;
    interfaceObject->Pipes[i].EndpointAddress    =
                                          endpointDescriptor->bEndpointAddress;
    interfaceObject->Pipes[i].Interval          = endpointDescriptor->bInterval;
    switch (endpointDescriptor->bmAttributes)
    {
    case 0x00:
        interfaceObject->Pipes[i].PipeType = UsbdPipeTypeControl;
        break;
    case 0x01:
        interfaceObject->Pipes[i].PipeType = UsbdPipeTypeIsochronous;
        break;
    case 0x02:
        interfaceObject->Pipes[i].PipeType = UsbdPipeTypeBulk;
        break;
    case 0x03:
        interfaceObject->Pipes[i].PipeType = UsbdPipeTypeInterrupt;
        break;
    }
    endpointDescriptor++;
    // PipeHandle  is opaque and is not to be filled in
    interfaceObject->Pipes[i].MaximumTransferSize = 64*1023;
    interfaceObject->Pipes[i].PipeFlags = 0;
}

ntStatus = CallUSBD(DeviceObject, urb, &length);

if (NT_SUCCESS(ntStatus))
```

```
    {
        UpdatePipeInfo(DeviceObject, ucInterfaceNumber, interfaceObject);
    }
    return ntStatus;
}

// ***********************************************
// Function: UpdatePipeInfo
// Purpose:  Store the pipe information
// ***********************************************
NTSTATUS
UpdatePipeInfo(
    IN PDEVICE_OBJECT DeviceObject,
    IN UCHAR ucInterfaceNumber,
    IN PUSBD_INTERFACE_INFORMATION interfaceObject
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;

    if (deviceExtension->InterfaceList[ucInterfaceNumber])
        ExFreePool(deviceExtension->InterfaceList[ucInterfaceNumber]);

    deviceExtension->InterfaceList[ucInterfaceNumber] = ExAllocatePool(NonPagedPool,
                                                            interfaceObject->Length);

    if (!deviceExtension->InterfaceList[ucInterfaceNumber])
        return STATUS_NO_MEMORY;

    // save a copy of the interfaceObject information returned
    RtlCopyMemory(deviceExtension->InterfaceList[ucInterfaceNumber],
                  interfaceObject,
                  interfaceObject->Length);

    return STATUS_SUCCESS;
}
```

```
// ************************************
//
// File: sample.h
//
// ************************************
#ifndef _sample_h_
#define _sample_h_

#ifdef DRIVER

#define REGISTRY_PARAMETERS_PATH  \
        L"\\REGISTRY\\Machine\\System\\CurrentControlSet\\SERVICES\\Sample\\Parameters"

#define RECIPIENT_DEVICE      0x01
#define FEATURE_REMOTE_WAKEUP 0x01
#define FEATURE_TEST_MODE     0x02

#define USBD_WIN98_GOLD_VERSION 0x0101
#define USBD_WIN98_SE_VERSION   0x0200
#define USBD_WIN2K_VERSION      0x0300

#define NAME_MAX 64

enum {
    eRemoved,         // Started->IRP_MN_REMOVE_DEVICE
                      // SurprisedRemoved->IRP_MN_REMOVE_DEVICE
                      // Initial State set in PnpAddDevice
    eStarted,         // Removed->IRP_MN_START_DEVICE
                      // RemovePending->IRP_MN_CANCEL_REMOVE_DEVICE
                      // StopPending->IRP_MN_CANCEL_STOP_DEVICE
                      // Stopped->IRP_MN_START_DEVICE
    eRemovePending,   // Started->IRP_MN_QUERY_REMOVE_DEVICE
    eSurprisedRemoved,// Started->IRP_MN_SURPRISE_REMOVAL
    eStopPending,     // Started->IRP_MN_QUERY_STOP_DEVICE
    eStopped          // StopPendingState->IRP_MN_STOP_DEVICE    INITIALIZING,
} PNP_STATE;

typedef struct _DEVICE_EXTENSION {

    // ************************************************************
    // Saved Device Objects
    // Device object we call when submitting Urbs/IRPs to the USB stack
    PDEVICE_OBJECT StackDeviceObject;
    // physical device object - submit power IRPs to
    PDEVICE_OBJECT PhysicalDeviceObject;

    DEVICE_CAPABILITIES DeviceCapabilities;

    POWER_STATE CurrentSystemState;
    POWER_STATE CurrentDeviceState;
    POWER_STATE NextDeviceState;

    // configuration handle for the configuration the
    // device is currently in
    USBD_CONFIGURATION_HANDLE ConfigurationHandle;

    // ptr to the USB device descriptor
    // for this device
    PUSB_DEVICE_DESCRIPTOR DeviceDescriptor;

    // keep an array of pointers to the configuration descriptor(s)
    PUSB_CONFIGURATION_DESCRIPTOR * ConfigurationDescriptors;
    ULONG ulCurrentConfigurationIndex;

    // Pointers to interface info structs
    PUSBD_INTERFACE_INFORMATION * InterfaceList;

    PIRP PowerIrp;

    PIRP WaitWakeIrp;
```

```
    enum PNP_STATE PnPstate;
    //
    // DeviceWake from capabilities
    //
    DEVICE_POWER_STATE DeviceWake;

    KEVENT PowerUpEvent;
    KEVENT PowerUpDone;
    PETHREAD   ThreadObject;

    KEVENT NoPendingTransactionsEvent;
    ULONG   ulOutStandingIrps;

    PUNICODE_STRING RegistryPath;
    UNICODE_STRING MofResourceName;

    // Kernel Event for sync calls for this D.O.
    KEVENT SyncEvent;

    // Name buffer for our named Functional device object link
    WCHAR DeviceLinkNameBuffer[NAME_MAX];

    BOOLEAN SetPowerEventFlag;
    PIRP IoctlIrp;

    BOOLEAN PowerTransitionPending;

} DEVICE_EXTENSION, *PDEVICE_EXTENSION;


NTSTATUS
DispatchWMI(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    );

VOID
Unload(
    IN PDRIVER_OBJECT DriverObject
    );

NTSTATUS
StopDevice(
    IN  PDEVICE_OBJECT DeviceObject
    );

NTSTATUS
RemoveDevice(
    IN  PDEVICE_OBJECT DeviceObject
    );

NTSTATUS
CallUSBD(
    IN PDEVICE_OBJECT DeviceObject,
    IN PURB Urb,
    OUT PULONG Length
    );

NTSTATUS
PnPAddDevice(
    IN PDRIVER_OBJECT DriverObject,
    IN PDEVICE_OBJECT PhysicalDeviceObject
    );

NTSTATUS
CreateDeviceObject(
    IN PDRIVER_OBJECT DriverObject,
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN PDEVICE_OBJECT *DeviceObject
    );
```

```
NTSTATUS
Create(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );

NTSTATUS
Close(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );

NTSTATUS
SelectInterface(
    PDEVICE_OBJECT DeviceObject,
    UCHAR ucInterfaceNumber,
    UCHAR ucAltSetting
    );

NTSTATUS
UpdatePipeInfo(
    IN PDEVICE_OBJECT DeviceObject,
    IN UCHAR ucInterfaceNumber,
    IN PUSBD_INTERFACE_INFORMATION interfaceObject
    );

NTSTATUS
ProcessIoctl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );

NTSTATUS
GenericCompletion(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PKEVENT Event
    );

VOID
FreeInterfaceList(
    IN PDEVICE_OBJECT DeviceObject
    );

NTSTATUS
ResetParentPort(
    IN IN PDEVICE_OBJECT DeviceObject
    );

NTSTATUS
GetRegistryDword(
    IN      PUNICODE_STRING RegPath,
    IN      PWCHAR    ValueName,
    IN OUT  PULONG    Value
    );

#endif

#endif
```

```
// ************************************
//
// File: pnp.c
//
// ************************************
#define DRIVER

#define INITGUID

#pragma warning(disable:4214) //  bitfield nonstd
#include "wdm.h"
#pragma warning(default:4214)

#include "stdarg.h"
#include "stdio.h"

#pragma warning(disable:4200) //non std struct used
#include "usbdi.h"
#pragma warning(default:4200)

#include "usbdlib.h"
#include "ioctl.h"
#include "sample.h"
#include "pnp.h"

extern UCHAR *SystemPowerStateString[];

extern UCHAR *DevicePowerStateString[];

UCHAR *PnPString[] = {
    "IRP_MN_START_DEVICE",                  // 0x00
    "IRP_MN_QUERY_REMOVE_DEVICE",           // 0x01
    "IRP_MN_REMOVE_DEVICE",                 // 0x02
    "IRP_MN_CANCEL_REMOVE_DEVICE",          // 0x03
    "IRP_MN_STOP_DEVICE",                   // 0x04
    "IRP_MN_QUERY_STOP_DEVICE",             // 0x05
    "IRP_MN_CANCEL_STOP_DEVICE",            // 0x06
    "IRP_MN_QUERY_DEVICE_RELATIONS",        // 0x07
    "IRP_MN_QUERY_INTERFACE",               // 0x08
    "IRP_MN_QUERY_CAPABILITIES",            // 0x09
    "IRP_MN_QUERY_RESOURCES",               // 0x0A
    "IRP_MN_QUERY_RESOURCE_REQUIREMENTS",   // 0x0B
    "IRP_MN_QUERY_DEVICE_TEXT",             // 0x0C
    "IRP_MN_FILTER_RESOURCE_REQUIREMENTS", // 0x0D
    "IRP_MN_READ_CONFIG",                   // 0x0F
    "IRP_MN_WRITE_CONFIG",                  // 0x10
    "IRP_MN_EJECT",                         // 0x11
    "IRP_MN_SET_LOCK",                      // 0x12
    "IRP_MN_QUERY_ID",                      // 0x13
    "IRP_MN_QUERY_PNP_DEVICE_STATE",        // 0x14
    "IRP_MN_QUERY_BUS_INFORMATION",         // 0x15
    "IRP_MN_DEVICE_USAGE_NOTIFICATION",     // 0x16
    "IRP_MN_SURPRISE_REMOVAL"               // 0x17
};
```

```
// ********************************************
// Function: DispatchPnP
// Purpose:  PnP Dispatch Routine
// ********************************************
NTSTATUS
DispatchPnP(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{
    PDEVICE_EXTENSION   deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION  IrpStack        = IoGetCurrentIrpStackLocation(Irp);
    BOOLEAN             passRequest;
    NTSTATUS            ntStatus;

    // Default: Pass the request down to the next lower driver
    // without a completion routine
    passRequest = TRUE;

    KdPrint (("DispatchPnP: Entering with IRP: %s\n",
                              PnPString[IrpStack->MinorFunction]));

    switch (IrpStack->MinorFunction)
    {
    case IRP_MN_START_DEVICE: // 0x00

        // In the completion routine, we complete the configuration
        // process
        IoCopyCurrentIrpStackLocationToNext(Irp);
        IoSetCompletionRoutine(Irp,
                               StartCompletionRoutine,
                               DeviceObject,
                               TRUE,
                               TRUE,
                               TRUE);

        ntStatus = IoCallDriver(deviceExtension->StackDeviceObject, Irp);
        passRequest = FALSE;

        break;

    case IRP_MN_REMOVE_DEVICE:  // 0x02

        deviceExtension->PnPstate = eRemoved;
        KdPrint(("Remove device\n"));
        ntStatus = RemoveDevice(DeviceObject);

        break; //IRP_MN_REMOVE_DEVICE

    case IRP_MN_STOP_DEVICE:    // 0x04

        ntStatus = StopDevice(DeviceObject);

        break; //IRP_MN_STOP_DEVICE

    case IRP_MN_QUERY_CAPABILITIES: // 0x09

        KdPrint (("IRP_MN_QUERY_CAPABILITIES\n"));

        IoCopyCurrentIrpStackLocationToNext(Irp);
        IoSetCompletionRoutine(Irp,
                               QueryCapabilitiesCompletionRoutine,
                               DeviceObject,
                               TRUE,
                               TRUE,
                               TRUE);

        ntStatus = IoCallDriver(deviceExtension->StackDeviceObject, Irp);

        passRequest = FALSE;
        break;
```

```
    default:
        // A PnP Minor Function was not handled
        KdPrint(("PnP IOCTL not handled 0x%x\n",
                 IrpStack->MinorFunction));
        break;

    } /* switch MinorFunction*/

    if (passRequest)
    {
        // just pass it down

        IoSkipCurrentIrpStackLocation(Irp);

        ntStatus = IoCallDriver(deviceExtension->StackDeviceObject, Irp);
    }

    KdPrint (("Exit PnP 0x%x\n", ntStatus));

    return ntStatus;

}//DispatchPnP

// ***********************************************
// Function: StartCompletionRoutine
// Purpose:  Start the configuration process
// ***********************************************
NTSTATUS
StartCompletionRoutine(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    )
{
    PDEVICE_OBJECT deviceObject       = (PDEVICE_OBJECT) Context;
    PDEVICE_EXTENSION deviceExtension = deviceObject->DeviceExtension;
    NTSTATUS ntStatus = STATUS_SUCCESS;
    KIRQL irql = KeGetCurrentIrql();

    KdPrint(("enter StartCompletionRoutine at IRQL %s with ntStatus 0x%x\n",
             irql == PASSIVE_LEVEL ? "PASSIVE_LEVEL": "DISPATCH_LEVEL",
             Irp->IoStatus.Status));

    if (NT_SUCCESS(Irp->IoStatus.Status))
    {
        // If the lower driver returned PENDING, mark our stack location as pending also.
        if (Irp->PendingReturned)
        {
            IoMarkIrpPending(Irp);
        }

        ntStatus = StartDevice(deviceObject);

        Irp->IoStatus.Status = ntStatus;
    }
    return ntStatus;
}
```

```
// ***********************************************
// Function: GetDescriptor
// Purpose:  Generic routine for acquiring descriptors
// ***********************************************
NTSTATUS
GetDescriptor(
    IN PDEVICE_OBJECT DeviceObject,
    IN UCHAR ucDescriptorType,
    IN UCHAR ucDescriptorIndex,
    IN USHORT usLanguageID,
    IN PVOID descriptorBuffer,
    IN ULONG ulBufferSize
    )
{
    NTSTATUS    ntStatus    = STATUS_SUCCESS;
    PURB        urb         = NULL;
    ULONG       length      = 0;

    urb = ExAllocatePool(NonPagedPool, sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST));

    if (!urb)
    {
        ntStatus = STATUS_NO_MEMORY;
        goto GetDescriptorEnd;
    }

    UsbBuildGetDescriptorRequest(urb,
                    (USHORT) sizeof (struct _URB_CONTROL_DESCRIPTOR_REQUEST),
                    ucDescriptorType,
                    ucDescriptorIndex,
                    usLanguageID,
                    descriptorBuffer,
                    NULL,
                    ulBufferSize,
                    NULL);

    ntStatus = CallUSBD(DeviceObject, urb, &length);

GetDescriptorEnd:
    return ntStatus;
}

// ***********************************************
// Function: StartDevice
// Purpose:  Acquire descriptors, configure device
// ***********************************************
NTSTATUS
StartDevice(
    IN  PDEVICE_OBJECT DeviceObject
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS ntStatus;
    PVOID descriptorBuffer  = NULL;
    ULONG siz;
    ULONG i = 0;

    KdPrint (("StartDevice: Entering\n"));

    // Get the device Descriptor
    siz = sizeof(USB_DEVICE_DESCRIPTOR);
    descriptorBuffer = ExAllocatePool(NonPagedPool, siz);

    if (!descriptorBuffer)
    {
        ntStatus = STATUS_NO_MEMORY;
        goto StartDeviceEnd;
    }
```

```
ntStatus = GetDescriptor(DeviceObject,
                         USB_DEVICE_DESCRIPTOR_TYPE,
                         0,
                         0,
                         descriptorBuffer,
                         siz);

if (!NT_SUCCESS(ntStatus))
{
    goto StartDeviceEnd;
}

deviceExtension->DeviceDescriptor = (PUSB_DEVICE_DESCRIPTOR)descriptorBuffer;

descriptorBuffer = NULL;

deviceExtension->ConfigurationDescriptors = ExAllocatePool(NonPagedPool,
                         deviceExtension->DeviceDescriptor->bNumConfigurations *
                         sizeof(PUSB_CONFIGURATION_DESCRIPTOR));

if (!deviceExtension->ConfigurationDescriptors)
{
    ntStatus = STATUS_NO_MEMORY;
    goto StartDeviceEnd;
}

// Acquire all configuration descriptors
for (i = 0; i < deviceExtension->DeviceDescriptor->bNumConfigurations; i++)
{
    UCHAR tempBuffer[9];
    // Get the configuration descriptor (first 9 bytes)
    siz = sizeof(USB_CONFIGURATION_DESCRIPTOR);
    descriptorBuffer = &tempBuffer; //ExAllocatePool(NonPagedPool, siz);

    if (!descriptorBuffer)
    {
        ntStatus = STATUS_NO_MEMORY;
        goto StartDeviceEnd;
    }

    ntStatus = GetDescriptor(DeviceObject,
                             USB_CONFIGURATION_DESCRIPTOR_TYPE,
                             (UCHAR)i,
                             0,
                             descriptorBuffer,
                             siz);

    if (!NT_SUCCESS(ntStatus))
    {
        goto StartDeviceEnd;
    }

    // Now get the rest of the configuration descriptor & save
    siz = ((PUSB_CONFIGURATION_DESCRIPTOR)descriptorBuffer)->wTotalLength;
    //ExFreePool(descriptorBuffer);
    //descriptorBuffer = NULL;

    descriptorBuffer = ExAllocatePool(NonPagedPool, siz);

    if (!descriptorBuffer)
    {
        ntStatus = STATUS_NO_MEMORY;
        goto StartDeviceEnd;
    }

    ntStatus = GetDescriptor(DeviceObject,
                             USB_CONFIGURATION_DESCRIPTOR_TYPE,
                             (UCHAR)i,
                             0,
                             descriptorBuffer,
                             siz);
```

```
        if (!NT_SUCCESS(ntStatus))
        {
            goto StartDeviceEnd;
        }

        deviceExtension->ConfigurationDescriptors[i] =
                                (PUSB_CONFIGURATION_DESCRIPTOR)descriptorBuffer;

        descriptorBuffer = NULL;
    } // get all configuration descriptors


    // If the GetDescriptor calls were successful, then configure the device with
    // the first configuration descriptor
    if (NT_SUCCESS(ntStatus))
    {
        ntStatus = ConfigureDevice(DeviceObject, 0);
    }

StartDeviceEnd:
    KdPrint (("StartDevice Exiting With ntStatus: 0x%x\n", ntStatus));

    return ntStatus;
}

// ***********************************************
// Function: ConfigureDevice
// Purpose:  Configure device with configuration index
// ***********************************************
NTSTATUS
ConfigureDevice(
    IN PDEVICE_OBJECT DeviceObject,
    IN ULONG ConfigIndex
    )
{
    PDEVICE_EXTENSION deviceExtension              = DeviceObject->DeviceExtension;
    PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor =
                            deviceExtension->ConfigurationDescriptors[ConfigIndex];
    PUSBD_INTERFACE_INFORMATION interfaceObject     = NULL;
    PUSB_INTERFACE_DESCRIPTOR interfaceDescriptor   = NULL;
    PUSBD_INTERFACE_LIST_ENTRY pIfcListEntry        = NULL;
    PUSB_INTERFACE_DESCRIPTOR InterfaceDescriptor   = NULL;

    NTSTATUS ntStatus       = STATUS_SUCCESS;
    PURB urb                = NULL;
    LONG interfaceNumber    = 0;
    LONG InterfaceClass     = -1;
    LONG InterfaceSubClass  = -1;
    LONG InterfaceProtocol  = -1;
    ULONG nInterfaceNumber  = 0;
    ULONG nPipeNumber       = 0;
    USHORT siz              = 0;
    ULONG  ulAltSettings    = 0;
    int i = 0;
    ULONG length            = 0;

    KdPrint(("enter ConfigureDevice\n"));

    // Build up the interface list entry information
    pIfcListEntry = ExAllocatePool(NonPagedPool,
              (ConfigurationDescriptor->bNumInterfaces + 1)*
              sizeof(USBD_INTERFACE_LIST_ENTRY));

    if (!pIfcListEntry)
    {
        ntStatus = STATUS_NO_MEMORY;
        goto ConfigureDeviceEnd;
    }
    for (interfaceNumber = 0;
         interfaceNumber < ConfigurationDescriptor->bNumInterfaces;
```

```
     interfaceNumber++)
{
    // Acquire each default interface descriptor
    InterfaceDescriptor =
           USBD_ParseConfigurationDescriptorEx(ConfigurationDescriptor,
                                               ConfigurationDescriptor,
                                               interfaceNumber,
                                               0,  // should be zero on initialization
                                               -1, // interface class not a criteria
                                               -1, // interface subclass not a criteria
                                               -1);// interface protocol not a criteria

    if (InterfaceDescriptor)
    {
        pIfcListEntry[interfaceNumber].InterfaceDescriptor = InterfaceDescriptor;
    }
    else
    {
        KdPrint(("ConfigureDevice() ParseConfigurationDescriptorEx() failed\n”));
        KdPrint(("returning STATUS_INSUFFICIENT_RESOURCES\n"));
        ntStatus = STATUS_UNSUCCESSFUL;
        goto ConfigureDeviceEnd;
    }
}
// set last entry in interface list to NULL
pIfcListEntry[ConfigurationDescriptor->bNumInterfaces].InterfaceDescriptor = NULL;

urb = USBD_CreateConfigurationRequestEx(ConfigurationDescriptor, pIfcListEntry);

if (!urb)
{
    ntStatus = STATUS_NO_MEMORY;
    goto ConfigureDeviceEnd;
}

ntStatus = CallUSBD(DeviceObject, urb, &length);

if (!NT_SUCCESS(ntStatus))
{
     KdPrint(("Select Config Failed. ntStatus = 0x%x; urb status = 0x%x\n",
              ntStatus,
              urb->UrbHeader.Status));
    goto ConfigureDeviceEnd;
}

// Save the configuration handle for this device
deviceExtension->ConfigurationHandle =
                               urb->UrbSelectConfiguration.ConfigurationHandle;

deviceExtension->InterfaceList = ExAllocatePool(NonPagedPool,
                                            ConfigurationDescriptor->bNumInterfaces *
                                            sizeof(PUSBD_INTERFACE_INFORMATION));

if (!deviceExtension->InterfaceList)
{
    ntStatus = STATUS_NO_MEMORY;
    goto ConfigureDeviceEnd;
}

// Build up the interface descriptor info
for (interfaceNumber = 0;
     interfaceNumber < ConfigurationDescriptor->bNumInterfaces;
     interfaceNumber++)
{
    interfaceObject = pIfcListEntry[interfaceNumber].Interface;

    ASSERT(interfaceObject);

    for (nPipeNumber=0; nPipeNumber < interfaceObject->NumberOfPipes; nPipeNumber++)
    {
        // fill out the interfaceobject info
```

```
            // perform any pipe initialization here
            interfaceObject->Pipes[nPipeNumber].MaximumTransferSize = 64*1023;
        }

        deviceExtension->InterfaceList[interfaceNumber] = NULL;
        UpdatePipeInfo(DeviceObject,
                       (UCHAR)interfaceNumber,
                       pIfcListEntry[interfaceNumber].Interface);
    }

ConfigureDeviceEnd:

    if (urb)
    {
        ExFreePool(urb);
        urb = NULL;
    }

    // if the interface list was not initialized, then
    // free up all the memory allocated by USBD_CreateConfigurationRequestEx
    if (!NT_SUCCESS(ntStatus))
    {
        for (interfaceNumber = 0;
             interfaceNumber < ConfigurationDescriptor->bNumInterfaces;
             interfaceNumber++)
        {
            ExFreePool(pIfcListEntry[interfaceNumber].Interface);
            pIfcListEntry[interfaceNumber].Interface = NULL;
        }
    }

    if (pIfcListEntry)
    {
        ExFreePool(pIfcListEntry);
        pIfcListEntry = NULL;
    }

    KdPrint (("exit ConfigureDevice (%x)\n", ntStatus));

    return ntStatus;
}
```

```
// ********************************************
// Function: QueryCapabilitiesCompletionRoutine
// Purpose:  Save the capabilities information
// ********************************************
NTSTATUS
QueryCapabilitiesCompletionRoutine(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    )
{
    PDEVICE_OBJECT deviceObject       = (PDEVICE_OBJECT) Context;
    PDEVICE_EXTENSION deviceExtension = deviceObject->DeviceExtension;
    NTSTATUS ntStatus = STATUS_SUCCESS;
    PIO_STACK_LOCATION IrpStack = IoGetCurrentIrpStackLocation (Irp);
    ULONG ulPowerLevel;

    KIRQL irql = KeGetCurrentIrql();

    KdPrint(("enter QueryCapabilitiesCompletionRoutine at IRQL %s with ntStatus 0x%x\n",
                irql == PASSIVE_LEVEL ? "PASSIVE_LEVEL": "DISPATCH_LEVEL",
                Irp->IoStatus.Status));

    //  If the lower driver returned PENDING, mark our stack location as pending also.
    if (Irp->PendingReturned)
    {
        IoMarkIRPPending(Irp);
    }
    if (NT_SUCCESS(Irp->IoStatus.Status))
    {
        ASSERT(IrpStack->MajorFunction == IRP_MJ_PNP);
        ASSERT(IrpStack->MinorFunction == IRP_MN_QUERY_CAPABILITIES);

        IrpStack = IoGetCurrentIRPStackLocation (Irp);

        RtlCopyMemory(&deviceExtension->DeviceCapabilities,
                      IrpStack->Parameters.DeviceCapabilities.Capabilities,
                      sizeof(DEVICE_CAPABILITIES));

        // print out capabilities info
        KdPrint(("************ Device Capabilites ************\n"));
        KdPrint(("SystemWake = 0x%x\n", deviceExtension->DeviceCapabilities.SystemWake));
        KdPrint(("DeviceWake = 0x%x\n", deviceExtension->DeviceCapabilities.DeviceWake));

        KdPrint(("SystemWake = %s\n",
                SystemPowerStateString[deviceExtension->DeviceCapabilities.SystemWake]));
        KdPrint(("DeviceWake = %s\n",
                DevicePowerStateString[deviceExtension->DeviceCapabilities.DeviceWake]));

        KdPrint(("Device Address: 0x%x\n", deviceExtension->DeviceCapabilities.Address));

        for (ulPowerLevel=PowerSystemUnspecified;
             ulPowerLevel< PowerSystemMaximum;
             ulPowerLevel++)
        {
            KdPrint(("Dev State Map: sys st %s = dev st %s\n",
                     SystemPowerStateString[ulPowerLevel],
                     DevicePowerStateString[
                        deviceExtension->DeviceCapabilities.DeviceState[ulPowerLevel]]));
        }
        Irp->IoStatus.Status = STATUS_SUCCESS;
    }

    return ntStatus;
}
```

```
// ************************************
//
// File: pnp.h
//
// ************************************
#ifndef _pmpnp_h_
#define _pmpnp_h_

NTSTATUS
DispatchPnP(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    );

NTSTATUS
StartCompletionRoutine(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    );

NTSTATUS
StartDevice(
    IN  PDEVICE_OBJECT DeviceObject
    );

NTSTATUS
GetDescriptor(
    IN PDEVICE_OBJECT DeviceObject,
    IN UCHAR ucDescriptorType,
    IN UCHAR ucDescriptorIndex,
    IN USHORT usLanguageID,
    IN PVOID descriptorBuffer,
    IN ULONG ulBufferSize
    );

NTSTATUS
ConfigureDevice(
    IN PDEVICE_OBJECT DeviceObject,
    IN ULONG ConfigIndex
    );

NTSTATUS
QueryCapabilitiesCompletionRoutine(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    );

#endif
```

```
// ************************************
//
// File: power.c
//
// ************************************
#define DRIVER

#pragma warning(disable:4214) //  bitfield nonstd
#include "wdm.h"
#pragma warning(default:4214)

#include "stdarg.h"
#include "stdio.h"

#pragma warning(disable:4200) //non std struct used
#include "usbdi.h"
#pragma warning(default:4200)

#include "usbdlib.h"
#include "ioctl.h"
#include "sample.h"
#include "power.h"
#include "pnp.h"

extern USBD_VERSION_INFORMATION gVersionInformation;
extern BOOLEAN gHasRemoteWakeupIssue;

UCHAR *SystemPowerStateString[] = {
   "PowerSystemUnspecified",
   "PowerSystemWorking",
   "PowerSystemSleeping1",
   "PowerSystemSleeping2",
   "PowerSystemSleeping3",
   "PowerSystemHibernate",
   "PowerSystemShutdown",
   "PowerSystemMaximum"
};

UCHAR *DevicePowerStateString[] = {
   "PowerDeviceUnspecified",
   "PowerDeviceD0",
   "PowerDeviceD1",
   "PowerDeviceD2",
   "PowerDeviceD3",
   "PowerDeviceMaximum"
};
extern UNICODE_STRING gRegistryPath;

// ***********************************************
// Function: PoSetDevicePowerStateComplete
// Purpose:  Completion routine for changing
//           the device power state
// ***********************************************
NTSTATUS
PoSetDevicePowerStateComplete(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    ) DeviceState
{
    PDEVICE_OBJECT     deviceObject    = (PDEVICE_OBJECT) Context;
    PDEVICE_EXTENSION  deviceExtension = deviceObject->DeviceExtension;
    NTSTATUS           ntStatus        = Irp->IoStatus.Status;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation (Irp);
    DEVICE_POWER_STATE deviceState     = irpStack->Parameters.Power.State.;

    KdPrint(("enter PoSetDevicePowerStateComplete\n"));
```

```
    // If the lower driver returned PENDING, mark our stack location as pending also.
    if (Irp->PendingReturned)
    {
        IoMarkIrpPending(Irp);
    }
    if (NT_SUCCESS(ntStatus))
    {
        KdPrint(("Updating current device state to %s\n",
                 DevicePowerStateString[deviceState]));
        deviceExtension->CurrentDeviceState.DeviceState = deviceState;
    }
    else
    {
        KdPrint(("Error: Updating current device state to %s failed.  NTSTATUS = 0x%x\n",
                 DevicePowerStateString[deviceState],
                 ntStatus));
    }
    KdPrint(("exit PoSetDevicePowerStateComplete\n"));
    return ntStatus;
}


// ***********************************************
// Function: HandleSetSytemPowerState
// Purpose:  Handle Set Power State (System)
// ***********************************************
NTSTATUS
HandleSetSytemPowerState(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp)
{
    PDEVICE_EXTENSION  deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS           ntStatus        = STATUS_SUCCESS;
    POWER_STATE        sysPowerState;

    // Get input system power state
    sysPowerState.SystemState = irpStack->Parameters.Power.State.SystemState;

    KdPrint(("Power() Set Power, type SystemPowerState = %s\n",
                   SystemPowerStateString[sysPowerState.SystemState] ));

    // If system is in working state always set our device to D0
    //  regardless of the wait state or system-to-device state power map
    if (sysPowerState.SystemState ==  PowerSystemWorking)
    {
        deviceExtension->NextDeviceState.DeviceState = PowerDeviceD0;
        KdPrint(("Power() PowerSystemWorking, will set D0, not use state map\n"));

        // cancel the pending wait/wake irp
        if (deviceExtension->WaitWakeIrp)
        {
            BOOLEAN bCancel = IoCancelIrp(deviceExtension->WaitWakeIrp);

            ASSERT(bCancel);
        }
    }
    else // powering down
    {
        NTSTATUS ntStatusIssueWW = STATUS_INVALID_PARAMETER; // assume that we won't be
                                                             // able to wake the system

        // issue a wait/wake irp if we can wake the system up from this system state
        // for devices that do not support wakeup, the system wake value will be
        // PowerSystemUnspecified == 0
        if (sysPowerState.SystemState <= deviceExtension->DeviceCapabilities.SystemWake)
        {
            ntStatusIssueWW = IssueWaitWake(DeviceObject);
        }
```

```
            if (NT_SUCCESS(ntStatusIssueWW) || ntStatusIssueWW == STATUS_PENDING)
            {
                // Find the device power state equivalent to the given system state.
                // We get this info from the DEVICE_CAPABILITIES struct in our device
                // extension (initialized in PnPAddDevice() )
                deviceExtension->NextDeviceState.DeviceState =
                  deviceExtension->DeviceCapabilities.DeviceState[sysPowerState.SystemState];

                KdPrint(("Power() IRP_MN_WAIT_WAKE issued, will use state map\n"));

                if (gHasRemoteWakeupIssue)
                {
                    StartThread(DeviceObject);
                }
            }
            else
            {
                // if no wait pending and the system's not in working state, just turn off
                deviceExtension->NextDeviceState.DeviceState = PowerDeviceD3;

                KdPrint(("Power() Setting PowerDeviceD3 (off)\n"));
            }
        } // powering down

        KdPrint(("Current Device State: %s\n",
            DevicePowerStateString[deviceExtension->CurrentDeviceState.DeviceState]));
        KdPrint(("Next Device State:    %s\n",
               DevicePowerStateString[deviceExtension->NextDeviceState.DeviceState]));

        // We've determined the desired device state; are we already in this state?
        if (deviceExtension->NextDeviceState.DeviceState !=
                            deviceExtension->CurrentDeviceState.DeviceState)
        {
            // attach a completion routine to change the device power state
            IoCopyCurrentIrpStackLocationToNext(Irp);
            IoSetCompletionRoutine(Irp,
                                   PoChangeDeviceStateRoutine,
                                   DeviceObject,
                                   TRUE,
                                   TRUE,
                                   TRUE);
        }
        else
        {
            // Yes, just pass it on to PDO (Physical Device Object)
            IoSkipCurrentIrpStackLocation(Irp);
        }
        PoStartNextPowerIrp(Irp);
        ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);

        KdPrint(("Power() Exit IRP_MN_SET_POWER (system) with ntStatus 0x%x\n", ntStatus));

        return ntStatus;
    }
```

```
// ********************************************
// Function: HandleSetDevicePowerState
// Purpose:  Handle Set Power State (Device)
// ********************************************
NTSTATUS
HandleSetDevicePowerState(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{
    PDEVICE_EXTENSION  deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS           ntStatus        = STATUS_SUCCESS;

    KdPrint(("Power() Set Power, type DevicePowerState = %s\n",
        DevicePowerStateString[irpStack->Parameters.Power.State.DeviceState]));

    IoCopyCurrentIrpStackLocationToNext(Irp);
    IoSetCompletionRoutine(Irp,
           PoSetDevicePowerStateComplete,
           // Always pass FDO to completion routine as its Context;
           // This is because the DriverObject passed by the system to the routine
           // is the Physical Device Object (PDO) not the Functional Device Object (FDO)
           DeviceObject,
           TRUE,            // invoke on success
           TRUE,            // invoke on error
           TRUE);           // invoke on cancellation of the Irp

    PoStartNextPowerIrp(Irp);
    ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);

    KdPrint(("Power() Exit IRP_MN_SET_POWER (device) with ntStatus 0x%x\n", ntStatus));
    return ntStatus;
}

// ********************************************
// Function: DispatchPower
// Purpose:  Dispatch routine for power irps
// ********************************************
NTSTATUS
DispatchPower(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    )
{
    PDEVICE_EXTENSION  deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS           ntStatus        = STATUS_SUCCESS;

    KdPrint(("DispatchPower() IRP_MJ_POWER\n"));

    switch (irpStack->MinorFunction)
    {
    case IRP_MN_WAIT_WAKE:

        KdPrint(("======================================\n"));
        KdPrint(("Power() Enter IRP_MN_WAIT_WAKE --\n"));
        // The only way this comes through us is if we send it via PoRequestPowerIrp
       // not attaching a completion routine
        IoSkipCurrentIrpStackLocation(Irp);
        PoStartNextPowerIrp(Irp);
        ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);
        break;

    case IRP_MN_SET_POWER:

        KdPrint(("Power() Enter IRP_MN_SET_POWER\n"));

        // Set Irp->IoStatus.Status to STATUS_SUCCESS to indicate that the device
        // has entered the requested state. Drivers cannot fail this IRP.
```

```
        switch (irpStack->Parameters.Power.Type)
        {
        case SystemPowerState:
            ntStatus = HandleSetSytemPowerState(DeviceObject, Irp);
            break;

        case DevicePowerState:

            ntStatus = HandleSetDevicePowerState(DeviceObject, Irp);
            break;

        }
        break;

    case IRP_MN_QUERY_POWER:
        //
        // A power policy manager sends this IRP to determine whether it can change
        // the system or device power state, typically to go to sleep.
        //
        if (irpStack->Parameters.Power.Type == SystemPowerState)
        {
            HandleSystemQueryIrp(DeviceObject, Irp);
        }
        else if (irpStack->Parameters.Power.Type == DevicePowerState)
        {
            HandleDeviceQueryIrp(DeviceObject, Irp);
        }
        break;

    default:

        KdPrint(("Power() UNKNOWN POWER MESSAGE (%x)\n", irpStack->MinorFunction));

        // All unhandled power messages are passed on to the PDO
        IoCopyCurrentIrpStackLocationToNext(Irp);
        PoStartNextPowerIrp(Irp);
        ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);
    }
    KdPrint(("Exit DispatchPower()  ntStatus = 0x%x\n", ntStatus ) );

    return ntStatus;
}

// ***********************************************
// Function: ChangeDevicePowerStateCompletion
// Purpose:  Handle completion of set device power
//           irps
// ***********************************************
NTSTATUS
ChangeDevicePowerStateCompletion(
    IN PDEVICE_OBJECT       DeviceObject,
    IN UCHAR                MinorFunction,
    IN POWER_STATE          PowerState,
    IN PVOID                Context,
    IN PIO_STATUS_BLOCK     IoStatus
    )
{
    PDEVICE_OBJECT    deviceObject    = (PDEVICE_OBJECT)Context;
    PDEVICE_EXTENSION deviceExtension = deviceObject->DeviceExtension;
    NTSTATUS          ntStatus        = STATUS_SUCCESS;
    PIRP              irp             = deviceExtension->IoctlIrp;

    if (irp)
    {
        IoCompleteRequest(irp, IO_NO_INCREMENT);
    }
    if (deviceExtension->SetPowerEventFlag)
    {
        // since we are powering up, set the event
        // so that the thread can query the device to see
        // if it generated the remote wakeup
```

```
        KdPrint(("Signal Power Up Event for Win98 Gold/SE/ME\n"));
        KeSetEvent(&deviceExtension->PowerUpEvent, 0, FALSE);
        deviceExtension->SetPowerEventFlag = FALSE;
    }

    KdPrint(("Exiting ChangeDevicePowerStateCompletion\n"));

    return ntStatus;
}

// ***********************************************
// Function: SetDevicePowerState
// Purpose:  Change the device state
// ***********************************************
NTSTATUS
SetDevicePowerState(
    IN PDEVICE_OBJECT     DeviceObject,
    IN DEVICE_POWER_STATE DevicePowerState)
{
    PDEVICE_EXTENSION   deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS            ntStatus        = STATUS_SUCCESS;
    POWER_STATE         powerState;

    powerState.DeviceState = DevicePowerState;

    ntStatus = PoRequestPowerIrp(deviceExtension->PhysicalDeviceObject,
                                 IRP_MN_SET_POWER,
                                 powerState,
                                 ChangeDevicePowerStateCompletion,
                                 DeviceObject,
                                 NULL);

    return ntStatus;
}

// ***********************************************
// Function: IssueWaitWake
// Purpose:  Create/issue a wait/wake irp
// ***********************************************
NTSTATUS
IssueWaitWake(
    IN PDEVICE_OBJECT   DeviceObject
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS ntStatus;
    POWER_STATE powerState;

    KdPrint (("*********************************\n"));
    KdPrint(("IssueWaitWake: Entering\n"));

    //
    // Make sure one isn't pending already -- serial will only handle one at
    // a time.
    //
    if (deviceExtension->WaitWakeIrp != NULL)
    {
        KdPrint(("Wait wake all ready active!\n"));
        return STATUS_INVALID_DEVICE_STATE;
    }

    powerState.SystemState = deviceExtension->DeviceCapabilities.SystemWake;

    ntStatus = PoRequestPowerIrp(deviceExtension->PhysicalDeviceObject,
                                 IRP_MN_WAIT_WAKE,
                                 powerState,
                                 RequestWaitWakeCompletion,
                                 DeviceObject,
                                 &deviceExtension->WaitWakeIrp);
```

```
    if (!deviceExtension->WaitWakeIrp)
    {
        KdPrint(("Wait wake is NULL!\n"));
        return STATUS_UNSUCCESSFUL;
    }

    KdPrint(("IssueWaitWake: exiting with ntStatus 0x%x (wait wake is 0x%x)\n",
                    ntStatus,
                    deviceExtension->WaitWakeIrp));
    return ntStatus;
}

// ********************************************
// Function: RequestWaitWakeCompletion
// Purpose:  Completion routine for irp generated
//           by PoRequestPowerIrp in IssueWaitWake
// ********************************************
NTSTATUS
RequestWaitWakeCompletion(
    IN PDEVICE_OBJECT       DeviceObject,
    IN UCHAR                MinorFunction,
    IN POWER_STATE          PowerState,
    IN PVOID                Context,
    IN PIO_STATUS_BLOCK     IoStatus
    )
{
    PDEVICE_OBJECT      deviceObject    = Context;
    PDEVICE_EXTENSION   deviceExtension = deviceObject->DeviceExtension;
    NTSTATUS            ntStatus        = IoStatus->Status;

    KdPrint(("########################################\n"));
    KdPrint(("###    RequestWaitWakeCompletion     ###\n"));
    KdPrint(("########################################\n"));

    deviceExtension->WaitWakeIrp = NULL;

    KdPrint(("RequestWaitWakeCompletion: Wake irp completed status 0x%x\n", ntStatus));

    //IoCompleteRequest(deviceExtension->WaitWakeIrp, IO_NO_INCREMENT);

    switch (ntStatus)
    {
    case STATUS_SUCCESS:
        KdPrint(("RequestWaitWakeCompletion: Wake irp completed succefully.\n"));

        deviceExtension->IoctlIrp = NULL;

        // We need to request a set power to power up the device.
        ntStatus = SetDevicePowerState(DeviceObject, PowerDeviceD0);

        break;
    case STATUS_CANCELLED:
        KdPrint(("RequestWaitWakeCompletion: Wake irp cancelled\n"));
        break;
    default:
        break;
    }
    return ntStatus;
}
```

```
// ******************************************
// Function: PoSystemQueryCompletionRoutine
// Purpose:  Finish handling system suspend
//           notification
// ******************************************
NTSTATUS
PoSystemQueryCompletionRoutine(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    )
{
    PDEVICE_OBJECT     DeviceObject     = (PDEVICE_OBJECT)Context;
    PDEVICE_EXTENSION  deviceExtension  = DeviceObject->DeviceExtension;

    KIRQL irql = KeGetCurrentIrql();

    KdPrint(("enter PoSystemQueryCompletionRoutine at IRQL %s with ntStatus 0x%x\n",
                irql == PASSIVE_LEVEL ? "PASSIVE_LEVEL": "DISPATCH_LEVEL",
                Irp->IoStatus.Status));

    //  If the lower driver returned PENDING, mark our stack location as pending also.
    if (Irp->PendingReturned)
    {
        IoMarkIrpPending(Irp);
    }

    // Finish any outstanding transactions,
    // wait for all transaction to finish here
    KeWaitForSingleObject(&deviceExtension->NoPendingTransactionsEvent,
                          Executive,
                          KernelMode,
                          FALSE,
                          NULL);

    Irp->IoStatus.Status = STATUS_SUCCESS;

    return STATUS_SUCCESS;
}

// ******************************************
// Function: PoChangeDeviceStateRoutine
// Purpose:  Completion routine for changing the
//           device state due to system set power irps
// ******************************************
NTSTATUS
PoChangeDeviceStateRoutine(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    )
{
    PDEVICE_OBJECT     DeviceObject     = (PDEVICE_OBJECT)Context;
    PDEVICE_EXTENSION  deviceExtension  = DeviceObject->DeviceExtension;
    NTSTATUS           RequestIrpStatus = STATUS_SUCCESS;
    DEVICE_POWER_STATE currDeviceState  =
                        deviceExtension->CurrentDeviceState.DeviceState;
    DEVICE_POWER_STATE nextDeviceState  = deviceExtension->NextDeviceState.DeviceState;

    KdPrint(("Change device state from %s to %s\n",
                DevicePowerStateString[currDeviceState],
                DevicePowerStateString[nextDeviceState]));

    //  If the lower driver returned PENDING, mark our stack location as pending also.
    if (Irp->PendingReturned)
    {
        IoMarkIrpPending(Irp);
    }

    // No, request that we be put into this state
    // by requesting a new Power Irp from the Pnp manager
```

```
    deviceExtension->PowerIrp = Irp;

    if (deviceExtension->NextDeviceState.DeviceState == PowerDeviceD0)
    {
        KdPrint(("Powering up device as a result of system wakeup\n"));
    }

    deviceExtension->SetPowerEventFlag =
            ((deviceExtension->NextDeviceState.DeviceState == PowerDeviceD0)    &&
             (deviceExtension->CurrentDeviceState.DeviceState != PowerDeviceD0) &&
             (gHasRemoteWakeupIssue));

    // simply adjust the device state if necessary
    if (currDeviceState != nextDeviceState)
    {
        deviceExtension->IoctlIrp = NULL;
        RequestIrpStatus = SetDevicePowerState(DeviceObject,
                                               nextDeviceState);
    }
    Irp->IoStatus.Status = STATUS_SUCCESS;

    return STATUS_SUCCESS;
}

// *****************************************
// Function: StartThread
// Purpose:  Generic routine for starting a thread
// *****************************************
NTSTATUS
StartThread(
    IN PDEVICE_OBJECT DeviceObject
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS ntStatus;
    HANDLE ThreadHandle;

    ntStatus = PsCreateSystemThread(&ThreadHandle,
                                    (ACCESS_MASK)0,
                                    NULL,
                                    (HANDLE) 0,
                                    NULL,
                                    PowerUpThread,
                                    DeviceObject);

    if (!NT_SUCCESS(ntStatus))
    {
       return ntStatus;
    }

    //
    // Convert the Thread object handle
    // into a pointer to the Thread object
    // itself. Then close the handle.
    //
    ObReferenceObjectByHandle(
               ThreadHandle,
               THREAD_ALL_ACCESS,
               NULL,
               KernelMode,
               &deviceExtension->ThreadObject,
               NULL );

    ZwClose( ThreadHandle );

    return ntStatus;
}
```

```
// ********************************************************************
// Function: PowerUpThread
// Purpose:  Performs a Get Device Descriptor call after system returns
//           to S0 to be replaced by vendor specific command for
//           querying the device if it generated a remote wakeup
// ********************************************************************
VOID
PowerUpThread(
        IN PVOID pContext
        )
{
    PDEVICE_OBJECT DeviceObject = pContext;
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;

    NTSTATUS ntStatus;

    KeSetPriorityThread(
              KeGetCurrentThread(),
              LOW_REALTIME_PRIORITY );

    ntStatus = KeWaitForSingleObject(&deviceExtension->PowerUpEvent,
                                     Suspended,
                                     KernelMode,
                                     FALSE,
                                     NULL);

    if (NT_SUCCESS(ntStatus))
    {
        PVOID descriptorBuffer  = NULL;
        ULONG siz;
        KdPrint(("Power Up Event signalled - Performing get descriptor call\n"));

        // Get the device Descriptor
        siz = sizeof(USB_DEVICE_DESCRIPTOR);
        descriptorBuffer = ExAllocatePool(NonPagedPool, siz);

        if (!descriptorBuffer)
        {
            ntStatus = STATUS_NO_MEMORY;
        }
        else
        {
            ntStatus = GetDescriptor(DeviceObject,
                                     USB_DEVICE_DESCRIPTOR_TYPE,
                                     0,
                                     0,
                                     descriptorBuffer,
                                     siz);
            ExFreePool(descriptorBuffer);
            descriptorBuffer = NULL;
        }
        KdPrint(("PowerUpThread: Get Descriptor Call: 0x%x\n", ntStatus));
    }
    KdPrint(("PowerUpThread - Terminating\n"));
    PsTerminateSystemThread( STATUS_SUCCESS );
}
```

```
// ********************************************************************
// Function: HandleSystemQueryIrp
// Purpose:  Determine whether or not we should prevent the system from
//           going to sleep
// ********************************************************************
NTSTATUS
HandleSystemQueryIrp(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PDEVICE_EXTENSION  deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS           ntStatus        = STATUS_SUCCESS;

    BOOLEAN fNoCompletionRoutine = FALSE;
    BOOLEAN fPassDownIrp         = TRUE;

    KdPrint(("=====================================\n"));
    KdPrint(("Power() IRP_MN_QUERY_POWER to %s\n",
             SystemPowerStateString[irpStack->Parameters.Power.State.SystemState]));

    // first determine if we are transmitting data
    if (deviceExtension->ulOutStandingIrps > 0)
    {
        BOOLEAN fOptionDetermined = FALSE;
        ULONG   ulStopDataTransmissionOnSuspend = 1;
        BOOLEAN fTransmittingCritialData = FALSE;

        // determine if driver should stop transmitting data
        fOptionDetermined = (BOOLEAN)GetRegistryDword(&gRegistryPath,
                                            L"StopDataTransmissionOnSuspend",
                                            &ulStopDataTransmissionOnSuspend);
        // Note COMPANY_X_PRODUCT_Y_REGISTRY_PATH is the absolute registry path
        // which would be defined as:       L"\\REGISTRY\\Machine\\System…
        // …\\CurrentControlSet\\Services\\COMPANY_X\\PRODUCT_Y and corresponds
        // to the following section in the registry (created by an .inf file or
        // installation routine): HKLM\\System\\CurrentControlSet\\Services…
        // …\\COMPANY_X\\PRODUCT_Y which would contain the DWORD entry
        // StopDataTransmissionOnSuspend

        if (ulStopDataTransmissionOnSuspend ||
            !fOptionDetermined ||
            irpStack->Parameters.Power.State.SystemState == PowerSystemShutdown)
            // stop data transmission if the option was set to do so or if the
            // option could not be read or if the system is entering S5
        {
            // Set a flag used to queue up incoming irps
            deviceExtension->PowerTransitionPending = TRUE;

            // attach a completion routine to wait for
            IoCopyCurrentIrpStackLocationToNext(Irp);
            IoSetCompletionRoutine(Irp,
                                   PoSystemQueryCompletionRoutine,
                                   DeviceObject,
                                   TRUE,
                                   TRUE,
                                   TRUE);

            fNoCompletionRoutine = FALSE;
        }
        else
            fPassDownIrp = FALSE;
    }

    ntStatus = PassDownPowerIrp(DeviceObject, Irp, fPassDownIrp, fNoCompletionRoutine);

    return ntStatus;
}
```

```
// ********************************************************************
// Function: PassDownPowerIrp
// Purpose:  Passes down a power irp.  If no completion routine is
//           necessary, then IoSkipCurrentIrpStackLocation() is called.
// ********************************************************************
NTSTATUS
PassDownPowerIrp(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN BOOLEAN bPassDownIrp,
    IN BOOLEAN bNoCompletionRoutine
    )
{
    PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS          ntStatus        = STATUS_SUCCESS;

    // Notify that driver is ready for next power irp
    PoStartNextPowerIrp(Irp);

    if (bPassDownIrp)
    {
        if (bNoCompletionRoutine)
        {
            // set the irp stack location for pdo
            IoSkipCurrentIrpStackLocation(Irp);
        }
        // pass the driver down to the pdo
        ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);
    }
    else
    {
        ntStatus = Irp->IoStatus.Status = STATUS_UNSUCCESSFUL;
        Irp->IoStatus.Information = 0;
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
    }
    return ntStatus;
}

// ********************************************************************
// Function: HandleDeviceQueryIrp
// Purpose:  Accept a request to power down if no data is being
//           transmitted
// ********************************************************************
NTSTATUS
HandleDeviceQueryIrp(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PDEVICE_EXTENSION  deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION irpStack        = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS           ntStatus        = STATUS_SUCCESS;

    KdPrint(("========================================\n"));
    KdPrint(("Power() IRP_MN_QUERY_POWER to %s\n",
             DevicePowerStateString[irpStack->Parameters.Power.State.DeviceState]));

    ntStatus = PassDownPowerIrp(DeviceObject,
                                Irp,
                                (BOOLEAN)(deviceExtension->ulOutStandingIrps == 0),
                                TRUE); // No completion routine

    return ntStatus;
}
```

```
// ************************************
//
// File: power.h
//
// ************************************
#ifndef _power_h_
#define _power_h_

NTSTATUS
PoSetDevicePowerStateComplete(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    );

NTSTATUS
HandleSetSytemPowerState(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );

NTSTATUS
HandleSetDevicePowerState(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    );

NTSTATUS
DispatchPower(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp
    );

NTSTATUS
ChangeDevicePowerStateCompletion(
    IN PDEVICE_OBJECT       DeviceObject,
    IN UCHAR                MinorFunction,
    IN POWER_STATE          PowerState,
    IN PVOID                Context,
    IN PIO_STATUS_BLOCK     IoStatus
    );

NTSTATUS
SetDevicePowerState(
    IN PDEVICE_OBJECT     DeviceObject,
    IN DEVICE_POWER_STATE PowerState
    );

NTSTATUS
IssueWaitWake(
    IN PDEVICE_OBJECT   DeviceObject
    );

NTSTATUS
RequestWaitWakeCompletion(
    IN PDEVICE_OBJECT       DeviceObject,
    IN UCHAR                MinorFunction,
    IN POWER_STATE          PowerState,
    IN PVOID                Context,
    IN PIO_STATUS_BLOCK     IoStatus
    );

NTSTATUS
PoSystemQueryCompletionRoutine(
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    );

NTSTATUS
PoChangeDeviceStateRoutine(
```

```
    IN PDEVICE_OBJECT NullDeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    );

NTSTATUS
StartThread(
    IN PDEVICE_OBJECT DeviceObject
    );

VOID
PowerUpThread(
        IN PVOID pContext
        );

NTSTATUS
HandleSystemQueryIrp(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );

NTSTATUS
HandleDeviceQueryIrp(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );

NTSTATUS
PassDownPowerIrp(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN BOOLEAN bPassDownIrp,
    IN BOOLEAN bNoCompletionRoutine
    );

#endif

// ************************************
//
// File: ioctl.c
//
// ************************************
#define DRIVER

#pragma warning(disable:4214) //  bitfield nonstd
#include "wdm.h"
#pragma warning(default:4214)

#include "stdarg.h"
#include "stdio.h"
#include "devioctl.h"

#pragma warning(disable:4200) //non std struct used
#include "usbdi.h"
#pragma warning(default:4200)

#include "usbdlib.h"

#include "ioctl.h"
#include "sample.h"
#include "power.h"

extern USBD_VERSION_INFORMATION gVersionInformation;
extern BOOLEAN gHasRemoteWakeupIssue;
```

```
// ********************************************************************
// Function: ProcessIoctl
// Purpose:  Handle DeviceIoCtrl requests
// ********************************************************************
NTSTATUS
ProcessIoctl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    NTSTATUS            ntStatus        = STATUS_SUCCESS;
    PDEVICE_EXTENSION   deviceExtension = DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION  irpStack        = IoGetCurrentIrpStackLocation (Irp);
    PVOID ioBuffer;
    ULONG inputBufferLength;
    ULONG outputBufferLength;
    ULONG ioControlCode;
    ULONG length = 0;


    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;

    ASSERT (deviceExtension);

    ioBuffer            = Irp->AssociatedIrp.SystemBuffer;
    inputBufferLength   = irpStack->Parameters.DeviceIoControl.InputBufferLength;
    outputBufferLength  = irpStack->Parameters.DeviceIoControl.OutputBufferLength;
    ioControlCode       = irpStack->Parameters.DeviceIoControl.IoControlCode;

    switch (ioControlCode)
    {
    case IOCTL_GET_DEVICE_POWER_STATE:
        {
            PULONG pulPowerState = (PULONG)ioBuffer;

            KdPrint(("IOCTL_GET_DEVICE_POWER_STATE: 0x%x\n",
                            (ULONG)deviceExtension->CurrentDeviceState.DeviceState));

            *pulPowerState = (ULONG)deviceExtension->CurrentDeviceState.DeviceState;
            Irp->IoStatus.Information = length = sizeof(ULONG);
        }
        break;

    case IOCTL_RESET_PARENT_PORT:

        ntStatus = ResetParentPort(DeviceObject);
        Irp->IoStatus.Information = 0;

        break;

    case IOCTL_GET_USBDI_VERSION:
        {
            PULONG pulVersion = (PULONG)ioBuffer;

            *pulVersion = gVersionInformation.USBDI_Version;
            Irp->IoStatus.Status = ntStatus = STATUS_SUCCESS;
            Irp->IoStatus.Status = sizeof(ULONG);
        }
        break;

       default:
           ntStatus = STATUS_INVALID_PARAMETER;
    }
    Irp->IoStatus.Status = ntStatus;
```

```
    if (ntStatus == STATUS_PENDING)
    {
        IoMarkIrpPending(Irp);
    }
    else
    {
        IoCompleteRequest (Irp, IO_NO_INCREMENT);
    }
    return ntStatus;
}

// ********************************************************************
// Function: ResetParentPort
// Purpose:  Rest the device
// ********************************************************************
NTSTATUS
ResetParentPort(
    IN PDEVICE_OBJECT DeviceObject
    )
{
    PDEVICE_EXTENSION deviceExtension  = DeviceObject->DeviceExtension;
    NTSTATUS ntStatus, status = STATUS_SUCCESS;
    PIRP irp;
    KEVENT event;
    IO_STATUS_BLOCK ioStatus;
    PIO_STACK_LOCATION nextStack;

    KeInitializeEvent(&event, NotificationEvent, FALSE);

    irp = IoBuildDeviceIoControlRequest(
                IOCTL_INTERNAL_USB_RESET_PORT,
                deviceExtension->StackDeviceObject,
                NULL,
                0,
                NULL,
                0,
                TRUE, // internal ( use IRP_MJ_INTERNAL_DEVICE_CONTROL )
                &event,
                &ioStatus);

    nextStack = IoGetNextIrpStackLocation(irp);

    ntStatus = IoCallDriver(deviceExtension->StackDeviceObject, irp);

    if (ntStatus == STATUS_PENDING)
    {
        status = KeWaitForSingleObject(&event,
                        Suspended,
                        KernelMode,
                        FALSE,
                        NULL);
    }
    else
    {
        ioStatus.Status = ntStatus;
    }

    //
    // USBD maps the error code for us
    //
    ntStatus = ioStatus.Status;

    KdPrint(("Exit ResetPort (%x)\n", ntStatus));

    return ntStatus;
}
```

```
// ************************************
//
// File: ioctl.h
//
// ************************************
#ifndef   __IOCTL_H__
#define   __IOCTL_H__

#define IOCTL_INDEX  0x0000


#define IOCTL_GET_DEVICE_POWER_STATE    CTL_CODE(FILE_DEVICE_UNKNOWN,  \
                                                 IOCTL_INDEX,\
                                                 METHOD_BUFFERED,  \
                                                 FILE_ANY_ACCESS)

#define IOCTL_RESET_PARENT_PORT          CTL_CODE(FILE_DEVICE_UNKNOWN,  \
                                                 IOCTL_INDEX+1, \
                                                 METHOD_BUFFERED,  \
                                                 FILE_ANY_ACCESS)

#define IOCTL_GET_USBDI_VERSION          CTL_CODE(FILE_DEVICE_UNKNOWN,  \
                                                 IOCTL_INDEX+2, \
                                                 METHOD_BUFFERED,  \
                                                 FILE_ANY_ACCESS)


#endif   // __IOCTL_H__

// ************************************
//
// File: guid.h
//
// ************************************
#ifndef _guid_h_
#define _guid_h_

// {41D40828-3DEB-11d3-BCFB-00A0C956C0B7}
DEFINE_GUID(GUID_CLASS_PM, 0x41d40828, 0x3deb, 0x11d3, 0xbc, 0xfb, 0x0, 0xa0, 0xc9, 0x56,
0xc0, 0xb7);

#endif
```

References

[1] *Advanced Configuration and Power Interface Specification, Revision 1.0b; Section 2.4.*

[2] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; Part 3: Power Management; 3.1 Sytem Power State; 3.1.1 System Working State S0*

[3] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; Part 3: Power Management; 3.1 Sytem Power State; 3.1.2 System Sleeping States S1, S2, S3, S4*

[4] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; Part 3: Power Management; 3.1 Sytem Power State; 3.1.3 System Shutdown State S5*

[5] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; Part 3: Power Management; 3.1 Sytem Power State; 3.1.4 System Power Actions*

[6] *Advanced Configuration and Power Interface Specification, Revision 1.0b; Section 2.3.*

[7] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; Part 3: Power Management; 1.0 Supporting Power Management In Drivers; 1.1 Kernel-Mode Power Management Components; 1.1.3 Power Manager*

[8] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; Part 3: Power Management; 1.0 Supporting Power Management In Drivers; 1.1 Kernel-Mode Power Management Components; 1.1.4 Driver Role In Power Management*

[9] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Reference; Part 1: Plug and Play; 3.0 Plug and Play Structures; DEVICE_CAPABILITIES*

[10] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; Part 3: Power Mangement; 1.0 Supporting Power Management in Drivers; 1.2 Power Management Responsibilities for Drivers; 1.2.3 Handling Power IRPs; 1.2.3.1 Power IRPS for the System.*

[11] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; Part 3: Power Mangement; 1.0 Supporting Power Management in Drivers; 1.2 Power Management Responsibilities for Drivers; 1.2.3 Handling Power IRPs; 1.2.3.2 Power IRPS for Individual Devices.*

[12] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Reference; Part 2: Power Management; 2.0 I/O Request for Power Management; IRP_MN_POWER_SEQUENCE*

[13] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Reference; Part 2: Power Management; 2.0 I/O Request for Power Management; IRP_MN_WAIT_WAKE*

[14] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Reference; Part 2: Power Mangement; 1.0 Power Management Support Routines; PoRegisterDeviceForIdleDetection.*

[15] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Reference; Part 2: Power Mangement; 1.0 Power Management Support Routines; PoSetDeviceBusy.*

[16] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Reference; Part 2: Power Management; 2.0 I/O Request for Power Management; IRP_MN_QUERY_POWER*

[17] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; 3.0 Handling System Power State Requests; 3.4 Handling IRP_MN_QUERY_POWER for System Power States; 3.4.1Failing a System Query-Power IRP*

[18] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Design Guide; Part 2: Plug and Play; 3.0 Starting, Stopping, and Removing Devices; 3.2 Stopping A Device For Resource Rebalancing; 3.2.5 Holding Incoming IRPs When A Device Is Paused*

[19] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Reference; Part 2: Power Management; 1.0 Power Management Support Routines; PoRequestPowerIRP*

[20] *Windows® 2000 DDK; Setup, Plug & Play, Power Management; Reference; Part 2: Power Management; 2.0 I/O Request for Power Management; IRP_MN_SET_POWER*