Universal Serial Bus
Understanding WDM Power Management, Version 1.0

November 5, 1999

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).

Table Of Contents

Introduction 3
Overview of Power states 4
- System Power States 4
- Simplified System Power State View 5
- Device Power States 5
- Simplified View of Device Power States 5

Power Information To Store In The Device Extension 6
Overview of Device Capabilities Structure 6
Acquiring Device Capabilities 7
The Four Power Irp Minor Functions 9
- IRP_MN_POWER_SEQUENCE 9
- IRP_MN_QUERY_POWER 9
- IRP_MN_WAIT_WAKE 10
- IRP_MN_SET_POWER 13

Generating Power Irps 18
- IRP_MN_POWER_SEQUENCE 18
- IRP_MN_SET_POWER (Device only) 18

Irp Sequences 20
- System Suspend 20
- System Resume 20
- System Resume due to Device Wakeup (Windows® 98 Gold/SE) 20
- System Resume due to Device Wakeup (Post Windows® 98 SE) 20
- Device Suspend 21
- Device Resume 21
- Device Wakeup (Post Windows® 98 SE) 21

Worst Case Scenario 21
Improper Power Management Consequences 22
References 23

Introduction

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 discussion of power states 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 DDK 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. However, the stack may deeper for some irps. This paper will now refer to this stack of device objects as the USB Stack.

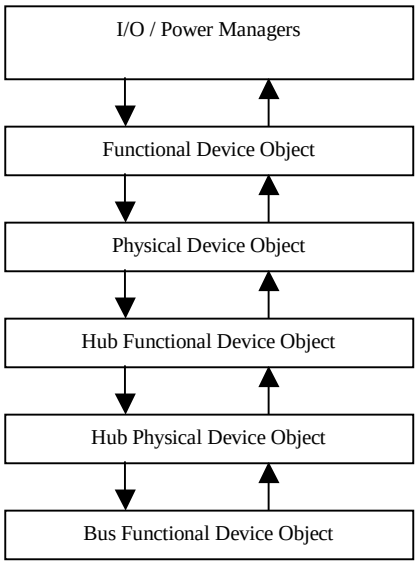


Figure 1. Simplified Device Layering View

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.

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.

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.

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.

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 820 chipset.

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.

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.

For more information on the ACPI system sleep states, see [1].

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. 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 [2].

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.

Power Information To Store In The Device Extension

For simplifying power management implementation, it is necessary to store the device capabilities [3], 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;
BOOL SystemPowerStatePending;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Sample Code 1. Part of Device Extension Parameters for Power Management Implementation

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 system power states to device states is shown in Table 4.

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 4. *SystemState-DeviceState* Mapping

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 *PowerSystemSleeping1*or *PowerSystemSleeping3* depending on the BIOS and host controller capabilities..

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*, otherwise it should be *PowerDeviceD3*.

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 resetting the `SurpriseRemovalOK` flag to `FALSE`. This allows for a device to be hot pluggable in Windows® 2000.

```
// in the PnP dispatch routine
case IRP_MN_QUERY_CAPABILITIES: // 0x09
    PM_KdPrint (("*******************************\n"));
    PM_KdPrint (("IRP_MN_QUERY_CAPABILITIES\n"));

    IoCopyCurrentIrpStackLocationToNext(Irp);
    IoSetCompletionRoutine(Irp,
                           PM_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
PM_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 = FALSE;  // 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
    PM_KdPrint(("************ Device Capabilites ************\n"));
    PM_KdPrint(("SystemWake = %s (0x%x)\n",
                SystemPowerStateString[deviceExtension
                    ->DeviceCapabilities.SystemWake],
                deviceExtension->DeviceCapabilities.SystemWake));

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


    for (ulPowerLevel=PowerSystemUnspecified;
         ulPowerLevel< PowerSystemMaximum;
         ulPowerLevel++)
    {
        PM_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. The device driver is the policy owner of the power sequence irp, set (device) power irp, and the wait/wake irp. The power manager is the policy owner of the set (system) power irp and the query power irp.

Minor Irp Code	Optional/Required
IRP_MN_POWER_SEQUENCE	Optional
IRP_MN_QUERY_POWER	Required
IRP_MN_SET_POWER	Required
IRP_MN_WAIT_WAKE	Optional

Table 5. 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 [4] for more information.

IRP_MN_QUERY_POWER

The power manager is the policy owner of this minor irp code. The power manager uses this irp to request approval of all device drivers 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. To approve the desired system sleep state, the driver needs to implement the following code.

```
case IRP_MN_QUERY_POWER:
    // Set a flag used to queue up incoming irps
    deviceExtension->PowerTransitionPending = TRUE;
    // Finish any outstanding transactions,
    // wait for all transaction to finish here
    KeWaitForSingleObject(deviceExtension->NoPendingTransactions,
                          Executive,
                          KernelMode,
                          FALSE,
                          NULL);
    // Notify that driver is ready for next power irp
    PoStartNextPowerIrp(Irp);
    // set the irp stack location for pdo
    IoSkipCurrentIrpStackLocation(Irp);
    // pass the driver down to the pdo
    ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);
    break;
```

Sample Code 4. IRP_MN_QUERY_POWER

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 [5], 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, generating 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: lose modem connection). The application will typically handle this situation.

However, failing a query irp may not prevent the system from following up with a set power irp to the designated sleep state (for example, in a notebook such as shutdown due to loss of power). Therefore, it is the author's recommendation to 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 [5] for more information.

To fail this irp (i.e., reject the system's request to enter the specified system state), the driver has the following minimal code implemented. However, if the driver is to fail / reject this irp, the driver, as stated in the previous paragraph, must be prepared for the scenario in which the system power is about to be lost. Note that the irp is not passed down the USB stack.

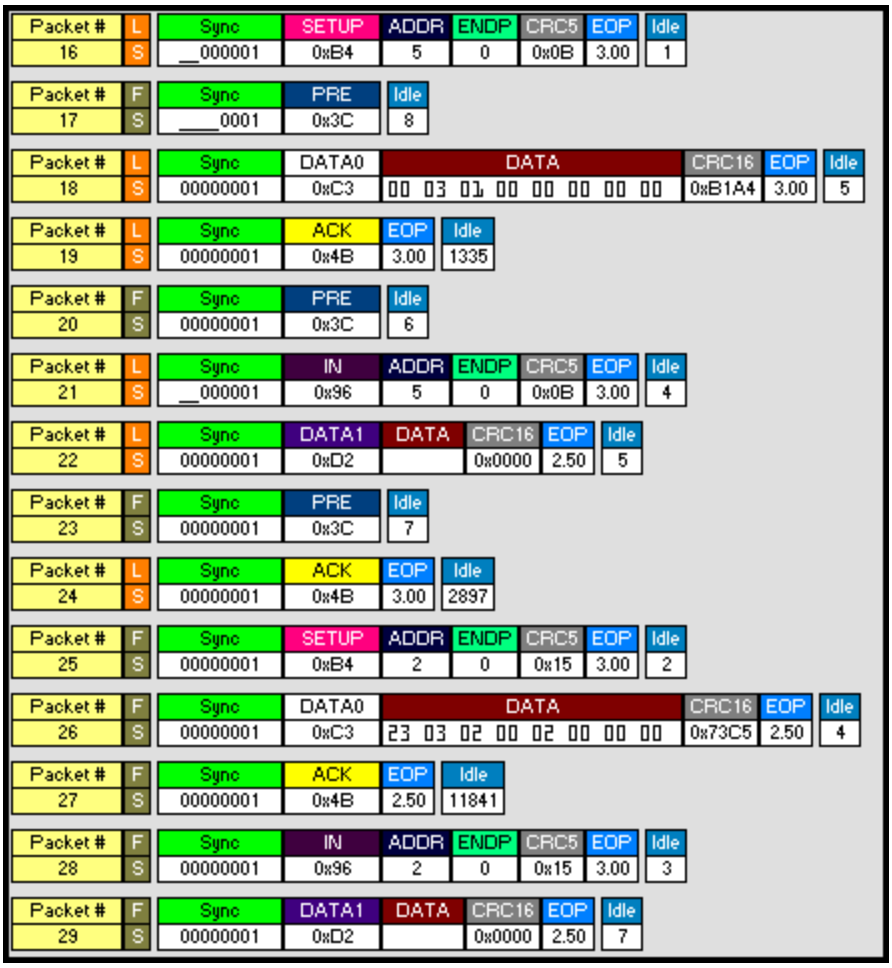
```
case IRP_MN_QUERY_POWER:
    ntStatus = Irp->IoStatus.Status = STATUS_UNSUCCESSFUL;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp);
    break;
```

Sample Code 5. Rejecting a Query Power Request

IRP_MN_WAIT_WAKE

Device drivers of remote wakeup capable devices generate this optional irp 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 a "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	0xB4	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	0xB4	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*(),as shown below. For more information, see [6,7]. 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,
                             PM_RequestWaitWakeCompletion,
                             DeviceObject,
                             &deviceExtension->WaitWakeIrp);
```

Sample Code 6. 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. 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:
    PM_KdPrint(("=======================================\n"));
    PM_KdPrint(("PM_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 7. Handling a Wait/Wake Irp

An issue exists in *Windows® 98* and *Windows 98® Second Edition* 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. In later operating system releases, such as *Windows® 2000* or *Millennium*, this issue does not exist. This issue only affects remote wakeup devices. Non-wakeup devices should suspend themselves when not in use for the benefit of potential power savings.

Remote wakeup devices need to determine the operating system version by acquiring the version of the USB Driver Interface (`USBDI`). This is shown in the *DriverEntry*() routine below. Note that the `USBDI` version information is stored in a global variable.

If the operating system is *Windows 98® Second Edition* or earlier, the device must not suspend itself, except in response of a system suspend. 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, then issue a vendor or class specific command to determine if the device caused the system resume.

```
USBD_VERSION_INFORMATION gVersionInformation;

NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    )
{
    NTSTATUS ntStatus = STATUS_SUCCESS;
    PDEVICE_OBJECT deviceObject = NULL;
    PWSTR path;


    DriverObject->MajorFunction[IRP_MJ_CREATE] = PM_Create;
                                            // called when Ring 3 app calls CreateFile()
    DriverObject->MajorFunction[IRP_MJ_CLOSE]  = PM_Close;
                                            // called when Ring 3 app calls CloseHandle()
    DriverObject->DriverUnload = PM_Unload;

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

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

    DriverObject->MajorFunction[IRP_MJ_PNP]   = PM_PnP;
    DriverObject->MajorFunction[IRP_MJ_POWER] = PM_Power;
    DriverObject->DriverExtension->AddDevice  = PM_PnPAddDevice; // called when device
                                                                 // is plugged in
    // determine the os version and store in a global.
    USBD_GetUSBDIVersion(&gVersionInformation);

    return ntStatus;
}
```

Sample Code 8. Driver Entry Routine

The `USBDIVersion` has the following values for each of the following operating system values. The sample code uses the value corresponding to *Windows 98™ Second Edition* to distinguish between operating systems that have this issue and those that provide complete wait/wake functionality.

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

Table 6. USBDIVersion Values

IRP_MN_SET_POWER

The Power Manager uses this irp 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. However, 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 *Windows® 98 Second Edition* or earlier, a system thread is created. The purpose of this system thread is to communicate with the device after resume system resume to determine whether the device caused a remote wakeup event. 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. In the sample code below, the system thread simply performs a get device descriptor call.

```
case IRP_MN_SET_POWER:

    // The system power policy manager sends this IRP to set the system power state.
    // The device is the policy owner for setting the device power.

     PM_KdPrint(("=======================================\n"));
     PM_KdPrint(("PM_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:
         {
             // Get input system power state
             sysPowerState.SystemState = irpStack->Parameters.Power.State.SystemState;
```

```
PM_KdPrint(("PM_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)
{
    desiredDevicePowerState.DeviceState = PowerDeviceD0;
    PM_KdPrint(("Powering up, will set D0 instead of 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 = PM_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 PM_PnPAddDevice() )
        desiredDevicePowerState.DeviceState =
         deviceExtension->DeviceCapabilities.DeviceState[
                                                    sysPowerState.SystemState];

        PM_KdPrint(("PM_Power() IRP_MN_WAIT_WAKE issued, will use state map\n"));

        // if we are using an early version of Win98, allocate the
        // event memory so that we check
        if (gVersionInformation.USBDI_Version <= USBD_WIN98_SE_VERSION)
        {
            PM_StartThread(DeviceObject);
        }
    }
    else
    {
        // if no wait/wake and the system's not in working state, just turn off
        desiredDevicePowerState.DeviceState = PowerDeviceD3;

        PM_KdPrint(("PM_Power() Settting PowerDeviceD3 (off)\n"));
    }
} // powering down

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

// We've determined the desired device state; are we already in this state?
if (desiredDevicePowerState.DeviceState !=
                            deviceExtension->CurrentDeviceState.DeviceState)
{
    BOOLEAN bSignal = FALSE;
```

```
            PM_KdPrint(("Change device state from %s to %s\n",
                DevicePowerStateString[deviceExtension->CurrentDeviceState.DeviceState],
                DevicePowerStateString[desiredDevicePowerState.DeviceState]));

            // Request that we be put into this state by requesting a new Power Irp
            // from the Pnp manager
            deviceExtension->PowerIrp = Irp;

            if (desiredDevicePowerState.DeviceState == PowerDeviceD0)
            {
                PM_KdPrint(("Powering up device as a result of system wakeup\n"));
                bSignal = deviceExtension->SignalDone = TRUE;
            }

            ntStatus = PoRequestPowerIrp(deviceExtension->PhysicalDeviceObject,
                                         IRP_MN_SET_POWER,
                                         desiredDevicePowerState,
                                         // completion routine will pass Irp down to PDO
                                         PM_PoRequestCompletion,
                                         DeviceObject,
                                         NULL);

            if (bSignal)
            {
                KeWaitForSingleObject(&deviceExtension->PowerUpDone,
                                      Executive,
                                      KernelMode,
                                      FALSE,
                                      NULL);
                // since we are powering up, set the event
                // so that the thread can query the device to see
                // if it generated the remote wakeup
                PM_KdPrint(("Signal Power Up Event for Win98 Gold/SE\n"));
                KeSetEvent(&deviceExtension->PowerUpEvent, 0, FALSE);
            }
        }
        else
        {
            // Yes, just pass it on to PDO (Physical Device Object)
            IoCopyCurrentIrpStackLocationToNext(Irp);
            PoStartNextPowerIrp(Irp);
            ntStatus = PoCallDriver(deviceExtension->StackDeviceObject, Irp);

        }

        PM_KdPrint(("PM_Power() Exit IRP_MN_SET_POWER (system)\n"));
    }
    break;

case DevicePowerState:

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

    // For requests to D1, D2, or D3 ( sleep or off states ),
     // sets deviceExtension->CurrentDeviceState to DeviceState immediately.
     // This enables any code checking state to consider us as sleeping or off
     // already, as this will imminently become our state.

    // For requests to DeviceState D0 ( fully on ), sets fGoingToD0 flag TRUE
    // to flag that we must set a completion routine and update
     // deviceExtension->CurrentDeviceState there.
     // In the case of powering up to fully on, we really want to make sure
     // the process is completed before updating our CurrentDeviceState,
     // so no IO will be attempted or accepted before we're really ready.

    fGoingToD0 = (BOOLEAN)
              (irpStack->Parameters.Power.State.DeviceState == PowerDeviceD0);

    IoCopyCurrentIrpStackLocationToNext(Irp);
```

```
    if (fGoingToD0)
    {
        PM_KdPrint(("PM_Power() Set PowerIrp Completion Routine, fGoingToD0 =%d\n",
                    fGoingToD0));
        IoSetCompletionRoutine(Irp,
               PM_PowerIrp_Complete,
               // 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);

    PM_KdPrint(("PM_Power() Exit IRP_MN_SET_POWER (device)\n"));
    break;
} /* case irpStack->Parameters.Power.Type */
break; /* IRP_MN_SET_POWER */
```

Sample Code 9. Handling IRP_MN_SET_POWER Irps

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

    PM_KdPrint (("*********************************\n"));
    PM_KdPrint(("PM_IssueWaitWake: Entering\n"));

    if (deviceExtension->WaitWakeIrp != NULL)
    {
        PM_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,
                                 PM_RequestWaitWakeCompletion,
                                 DeviceObject,
                                 &deviceExtension->WaitWakeIrp);
    if (!deviceExtension->WaitWakeIrp)
    {
        PM_KdPrint(("Wait wake is NULL! (0x%x)\n", ntStatus));
        NtStatus = STATUS_UNSUCCESSFUL;
    }
    PM_KdPrint(("PM_IssueWaitWake: exiting with ntStatus 0x%x\n",
                ntStatus));
    return ntStatus;
}
```

Sample Code 10. Issue Wait/Wake Function

```
NTSTATUS
PM_PoRequestCompletion(
    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;
    PIRP              irp             = deviceExtension->PowerIrp;
    NTSTATUS ntStatus;

    // We will return the status set by the PDO for the power request we're completing
    ntStatus = IoStatus->Status;

    PM_KdPrint(("Enter PM_PoRequestCompletion()\n"));

    PM_KdPrint(("Setting device state to %s\n",
                                      DevicePowerStateString[PowerState.DeviceState]));
    deviceExtension->CurrentDeviceState.DeviceState = PowerState.DeviceState;

    // we must pass down to the next driver in the stack
    IoCopyCurrentIrpStackLocationToNext(irp);

    PoStartNextPowerIrp(irp);
    PoCallDriver(deviceExtension->StackDeviceObject, irp);

    deviceExtension->PowerIrp = NULL;

    if (deviceExtension->SignalDone)
    {
        deviceExtension->SignalDone = FALSE;
        KeSetEvent(&deviceExtension->PowerUpDone, 0, FALSE);
    }
    PM_KdPrint(("PM_PoRequestCompletion() Exit IRP_MN_SET_POWER\n"));
    return ntStatus;
}
```

Sample Code 11. Set Device Power Completion Routine Due To Set System Power

```
NTSTATUS
PM_PowerIrp_Complete(
    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;

    PM_KdPrint(("enter PM_PowerIrp_Complete\n"));

    if (Irp->PendingReturned)   // if lower driver returned pending,
        IoMarkIrpPending(Irp);  // mark stack location as pending also
    irpStack = IoGetCurrentIrpStackLocation (Irp);
    PM_KdPrint(("Setting device state to D0\n"));
    deviceExtension->CurrentDeviceState.DeviceState = PowerDeviceD0;

    Irp->IoStatus.Status = ntStatus;

    return ntStatus;
}
```

Sample Code 12. Completion Routine For When Powering Up To *PowerDeviceD0* in Response to Set Device Power Irps

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 irps that a device driver may generate. The only minor irp code that a device cannot generate is an `IRP_MN_QUERY_POWER` irp.

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 [4] 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
PM_SetDevicePowerState(
    IN PDEVICE_OBJECT   DeviceObject,
    IN ULONG            ulPowerState
    )
{
    PDEVICE_EXTENSION   deviceExtension = DeviceObject->DeviceExtension;
    NTSTATUS            ntStatus        = STATUS_SUCCESS;
    POWER_STATE         PowerState;

    PowerState.DeviceState = (DEVICE_POWER_STATE)ulPowerState;

    deviceExtension->SelfRequestedPowerIrpEvent = ExAllocatePool(NonPagedPool,
                                                                 sizeof(KEVENT));

    if (!deviceExtension->SelfRequestedPowerIrpEvent)
        return STATUS_INSUFFICIENT_RESOURCES;

    KeInitializeEvent(deviceExtension->SelfRequestedPowerIrpEvent,
                      NotificationEvent,
                      FALSE);

    ntStatus = PoRequestPowerIrp(deviceExtension->PhysicalDeviceObject,
                                 IRP_MN_SET_POWER,
                                 PowerState,
                                 PM_PoSelfRequestCompletion,
                                 DeviceObject,
                                 NULL);
```

```
    if  (ntStatus == STATUS_PENDING)
    {
        // We only need to wait for completion if we're powering up
        if ( deviceExtension->CurrentDeviceState.DeviceState >
                                                PowerState.DeviceState)
        {
            NTSTATUS waitStatus;

            waitStatus = KeWaitForSingleObject(
                          deviceExtension->SelfRequestedPowerIrpEvent,
                          Suspended,
                          KernelMode,
                          FALSE,
                          NULL);
            ExFreePool(deviceExtension->SelfRequestedPowerIrpEvent);
            deviceExtension->SelfRequestedPowerIrpEvent = NULL;
            //KeResetEvent(&deviceExtension->SelfRequestedPowerIrpEvent);
        }
    }

    deviceExtension->CurrentDeviceState.DeviceState = PowerState.DeviceState;
    ntStatus = STATUS_SUCCESS;

    return ntStatus;
}
```

Sample Code 13. Generating a Set Device Power Irp

```
NTSTATUS
PM_PoRequestCompletion(
    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;
    PIRP              irp             = deviceExtension->PowerIrp;
    NTSTATUS ntStatus;

    // We will return the status set by the PDO for the power request we're completing
    ntStatus = IoStatus->Status;

    PM_KdPrint(("Enter PM_PoRequestCompletion()\n"));

    PM_KdPrint(("Setting device state to %s\n",
                                    DevicePowerStateString[PowerState.DeviceState]));
    deviceExtension->CurrentDeviceState.DeviceState = PowerState.DeviceState;

    // we must pass down to the next driver in the stack
    IoCopyCurrentIrpStackLocationToNext(irp);

    PoStartNextPowerIrp(irp);
    PoCallDriver(deviceExtension->StackDeviceObject, irp);

    deviceExtension->PowerIrp = NULL;

    if (deviceExtension->SignalDone)
    {
        deviceExtension->SignalDone = FALSE;
        KeSetEvent(&deviceExtension->PowerUpDone, 0, FALSE);
    }
    PM_KdPrint(("PM_PoRequestCompletion() Exit IRP_MN_SET_POWER\n"));
    return ntStatus;
}
```

Sample Code 14. 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 6.

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

Sample Code 15. 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)

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 (Post Windows® 98 SE)

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 (Win98 Gold & SE)	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 (Post Win98 SE)	1. Wait/wake completion (success) 2. Driver-generated IRP_MN_SET_POWER 3. System-generated IRP_MN_SET_POWER

Table 7. 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 (Post Windows® 98 SE)

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 (Post Windows® 98 SE)	1. Wait/wake completion routine 2. Device-generated IRP_MN_SET_POWER irp

Table 8. Code Sequences for Device Power Management Events

Worst Case Scenario

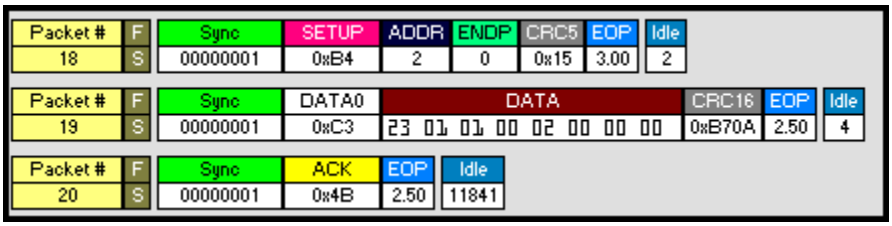
The driver must also accommodate the worst-case scenario for system suspend: the USB host controller is turned off. The driver recognizes this scenario by a set system power irp to a suspend state, followed by a PnP remove irp, and finally an unload irp. The driver must gracefully handle this sequence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



CATC Trace 2. Disabling Parent Port Of Device

References

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

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

[3] *Preliminary Windows® 2000/WDM DDK; Setup, Plug & Play, Power Management; Reference; 3.0 Plug and Play Structures; DEVICE_CAPABILITIES*

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

[5] *Preliminary Windows® 2000/WDM 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*

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

[7] *Preliminary Windows® 2000/WDM DDK; Setup, Plug & Play, Power Management; Reference; Part 2: Power Management; 1.0 Power Management Support Routines; PoRequestPowerIrp*