Universal Serial Bus
Handling The IRP_MN_START_DEVICE PnP IRP

A Short Primer for Configuring a USB Device, Version 0.1

Kosta Koeman
Intel Corporation
kosta.koeman@intel.com

Abstract

This white paper summarizes the proper steps required in configuring an USB device and provides generic code that accommodates any device. This paper assumes that the reader has some experience with WDM drivers.

Contributors

John Keys, Intel Corporation

Revision History

Version 1.0 Original Release

Table Of Contents

Introduction

The purpose of this document is to clarify to developers how drivers to configure a USB device. This brief paper will provide generic code that will accommodate any number of configuration descriptors and interfaces.

Configuration of a USB device is a simple to step process: descriptor acquisition, and device registration. The first step involves acquiring the device and configuration descriptors (the latter including interface, endpoint, and possibly class descriptors). These descriptors are used for the second step: registering interface and pipe information to the operating system. This process occurs when receiving the PnP IRP, IRP_MN_START_DEVICE. For the *Windows™ 2000* DDK perspective on handling this irp, please see [1].

The code provided in this document was derived from *Windows® 2000* DDK (March 9, 2000 build) sample code, and the author's experimentation.

Handling IRP_MN_START_DEVICE

The following section provides the necessary code for handling the IRP_MN_START_DEVICE PnP IRP. The first step is to pass down the IRP so that the underlying drivers can complete their configuration. This is shown below by attaching a completion routine (in this case *StartCompletionRoutine*()), which commences with the first stage of the configurations process: acquiring the descriptors.

It should be noted that is some sample code, the driver passes the PnP start IRP down the stack, then blocks on an event that is set in the attached completion routine. This is acceptible at PASSIVE_LEVEL (the IRQL in which PnP IRPs are dispatched), but generally not at DISPATCH_LEVEL [2]. As a matter of practice, the author advises utilizing completion routines for continuing work after passing an IRP down the stack.

```
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;
        .
        .
        .

    } /* 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
```

Sample Code 1. Portion of PnP Dispatch Routine

```
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;
}
```

Sample Code 2. Completion Routine for the PnP IRP, IRP_MN_START_DEVICE

Once the IRP_MN_START_DEVICE has been passed down the stack and control has returned to the driver in the completion routine, the driver may begin communicating with the device and acquire the descriptors: first the device descriptor and second with the configuration descriptor (including the interface, endpoint, and other class descriptors). Note that a device has only one device descriptor, but can also have several configuration descriptors. Therefore, rather than storing a pointer directly in the device extension, an array of pointers to configuration descriptors is created. For the *Windows™ 2000* DDK perspective on configuring a USB device, please see [3].

```
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++)
    {
        // Get the configuration descriptor (first 9 bytes)
        UCHAR tempBuffer[9];
        descriptorBuffer = &tempBuffer;

        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;

            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;
    }
```

Sample Code 3. Start Device Routine

```
    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;
}
```

Sample Code 4. Generic Get Descriptor Routine

Once all the device and configuration descriptors are acquired, the driver commences the second stage of device configuration: registration of interface and pipe information to the operating system. The driver designates this information in the *ConfigureDevice()* routine shown below by building up an interface list.

```
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++)
{
    InterfaceDescriptor =
        USBD_ParseConfigurationDescriptorEx(ConfigurationDescriptor,
                                ConfigurationDescriptor,
                                interfaceNumber,
                                0,    // set default interface
                                -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
    {
        ntStatus = STATUS_INVALID_PARAMETER;
        goto ConfigureDeviceEnd;
    }
}
pIfcListEntry[ConfigurationDescriptor->bNumInterfaces].InterfaceDescriptor
                                          = NULL; // set last entry to NULL

urb = USBD_CreateConfigurationRequestEx(ConfigurationDescriptor,
                                       pIfcListEntry);
if (!urb)
{
    ntStatus = STATUS_NO_MEMORY;
    goto ConfigureDeviceEnd;
}

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

if (!NT_SUCCESS(ntStatus))
{
    goto ConfigureDeviceEnd;
}
deviceExtension->InterfaceList =
    ExAllocatePool(NonPagedPool,
                   ConfigurationDescriptor->bNumInterfaces *
                   sizeof(PUSBD_INTERFACE_INFORMATION));

if (!deviceExtension->InterfaceList)
{
    ntStatus = STATUS_NO_MEMORY;
    goto ConfigureDeviceEnd;
}
// Save the configuration handle for this device
deviceExtension->ConfigurationHandle =
              urb->UrbSelectConfiguration.ConfigurationHandle;

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

    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] =
```

```
                                    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;
}
```

Sample Code 5. Register/Setup Interface/Pipe Information

Selecting Alternate Interfaces

When configuring a device, the default settings should be set. Once an alternate setting is desired, first terminate all transactions on endpoints that correspond to that changing interface alternate setting. Then implement the following code.

```
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;
}
```

Sample Code 6. Select (Alternate) Interface

```
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;
}
```

Sample Code 7. Saving Pipe Information Code

References

[1] *Windows™ 2000 DDK; Setup, Plug & Play, Power Management, Design Guide, Part 2: Plug and Play; 3.0 Starting, Stopping, and Removing Devices; 3.1 Starting a Device; 3.1.1 Starting a Device in a Function Driver*

[2] *Windows 2000™ DDK; Setup, Plug & Play, Power Management, Design Guide, Part 2: Plug and Play; 2.0 Rules for Handling PnP IRPS; 2.3 Postponing PnP IRP Processing Until Lower Drivers Finish*

[3] *Windows 2000™ DDK; Kernel-Mode Drivers; Design Guide; Part 5: USB Drivers; 2.0 Configuring USB Devices*

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
```