



Power saving of using USB Selective Suspend Support Whitepaper

(Version 0.6, May 20, 2003, Kris Fleming)

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © 2002 - 2003, Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

**Intel
Mobile
Platforms
Group**

Power saving of using USB Selective Suspend Support Whitepaper

November 2002

Abstract

Extended battery life is an important part of the Intel's mobile platforms and Intel is actively working on several efforts to improve battery life for Intel's mobile platforms. This whitepaper is intended for mobile platform OEMs and IHVs. The whitepaper covers a potential power issues that results in ~10% reduction in battery life for mobile platforms, describes a solution, and implementation details to eliminate these power issues. These power issues effects mobile platforms which contain intergrated USB* devices and mobile platforms while they are using external USB devices.

Introduction

Today mobile platforms contain many internally integrated peripheral components. Many of these are new technology and previously did not exist or were connected externally and sold separately from the mobile platform. Some examples of these integrated peripherals are video cameras, memory card connections, security card readers, wireless communications, and many other peripherals. Increasingly these devices are using USB to provide the internal connections. USB 2.0 supports additional bandwidth and many other additional capabilities. USB will continue to be a popular solution as more internal devices are added to the mobile platform. OEMs and device vendors must understand and resolve the potential platform power issues of using USB.

Background

In order to understand the details of the potential platform power issue and solution some background information is needed. Each background section provides a high level summary of the topic as it relates to this white paper. Four topics are covered:

1. System Power Breakdown
2. ACPI Overview
3. USB Selective Suspend
4. Windows XP support for Selective Suspend

Power Background

To understand the details on the potential platform power issues some basic background information is need. Figure 1 shows an example of the power percentage breakdown for today's typical notebook, this diagram is intended to be used as guideline information. PLS is an

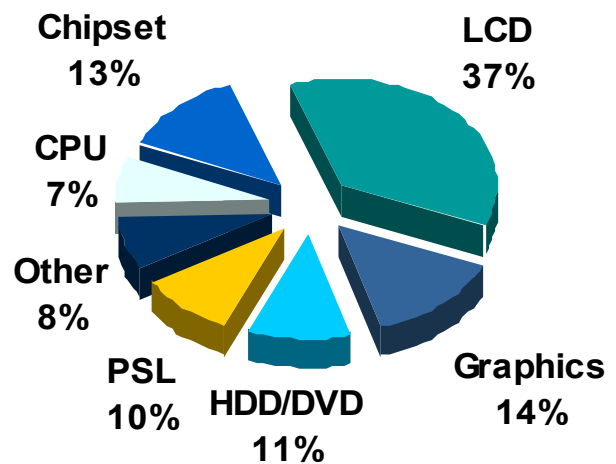


Figure 1: Platform Power Percentage Breakdown

acronym for Power Supply Loss. Note that the CPU and chipsets consume ~20% of the total system power. The proposal in this whitepaper is focused on maximizing power saving for the power used by the CPU and chipset, and does not address power savings for any of the other components.

ACPI Background

ACPI (Advanced Configuration and Power Interface) is an open standard co-developed by Compaq, Intel, Microsoft, Phoenix, and Toshiba.

Power saving of using USB Selective Suspend Support Whitepaper

November 2002

The ACPI standard defines standard interfaces for configuration and power management. Figure 2, is a diagram of the defined ACPI

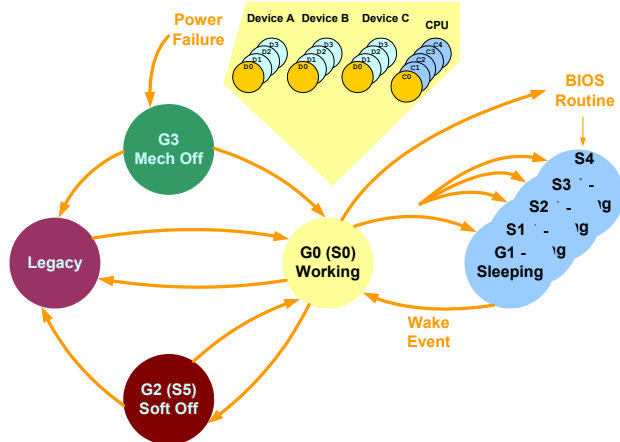


Figure 2: Defined ACPI States

power states and the transitions from one power state to another. G states are system power states. G0 is the working state in which users consider the computer to be on. In the G0 state the Intel Architecture base processors operate in various power states which are referred to as C states, ranging from C0 (Full power) to C1, C2, C3, or C4 (lowest low power states).

Additionally in the G0 state, each of the devices attached to that platform are in various power states which are referred to as Device (D) states, ranging from D0 (Full power) to D1, D2, D4. G1 is called the sleeping state and defines 4 sleep states, S0 (Full on), S1, S2, S3, S4. S1, S2, S3 are known as the “Suspend” states in which the processor is stopped and the system state and context is stored in memory. S4 state is known as the “Hibernate” state in which the entire system is stopped and all of the system states and contexts are stored to disk. Each of these different power states allow for different level of power saving and usage.

For this paper the C3/C4 states are of most interest. C3 and C4 are low power states for the

processor which the processor saves power and allows for battery life to be extended by ~10% under typical use. The C3/C4 states are entered when the system is idle (including IO) and the processor only takes a few milliseconds to enter this state. Any bus mastering activities will prevent the processor from entering C3/C4.

USB Background

The USB specification [i] defines support for low power modes. One of these low power mode features is called USB “Selective Suspend”. This feature allows an USB device driver which supports selective suspend to turn off the USB device it controls when the device is idle. When the device is no longer idle and is to be used again, the system wakes the device and resumes normal operation. The USB specification [i] also defines an additional low power feature call Remote Wakeup. This feature, if supported by the USB device, allows the USB device to wake itself from Selective Suspend. This allows a device specific event to resume the system instead of the system resuming the device.

Power saving of using USB Selective Suspend Support Whitepaper

November 2002

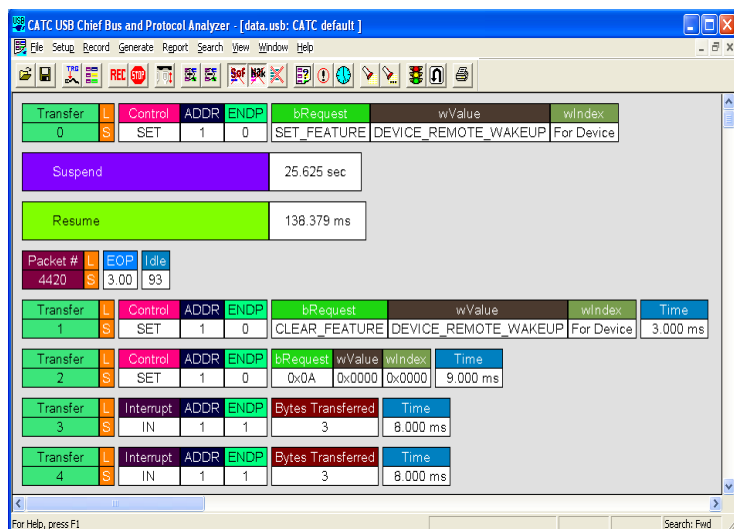


Figure 3: USB Trace for Selective Suspend and Remote Wakeup

Figure 3 shows a USB trace for the USB commands that occur for Selective Suspend and Remote Wakeup. This trace was done using a CATC* USB Chief Bus and Protocol Analyzer. The first transfer of interest is the “Set Feature”, in which the USB device’s remote wakeup feature is enabled. This only occurs if the device supports remote wakeup. Next, the USB device was suspended for ~26 seconds. After the device wakes the host, the USB device is resumed and the remote wakeup feature is disabled by the “Clear Feature” USB command. Note that in this example, the resume process consumes ~138 milliseconds which will vary from device to device. The resume delay for each device should be considered when determining an appropriate idle condition with part of the idle detection algorithms is using time.

USB devices have two possible methods to be powered: bus-power or self-powered. Bus-powered devices receive power from the USB bus itself, while self-powered devices are powered by an alternative power source. The USB specification [i] defines power restrictions

while operating in low power mode. When a bus powered device is in “Selective Suspend” it is current limited to 2.5 mA if Remote wakeup is enabled, and 500 uA otherwise. This current limitation is often the major reason why internal devices in a mobile platform are self-powered.

Windows XP support for USB Selective Suspend

Microsoft’s Windows XP has built in support for both USB 1.1 and USB 2.0 and incorporates support for USB “Selective Suspend”. This new feature will stop the USB host controller (HC) from polling if all ports are suspended. This will allow processor to go to C3/C4 state.

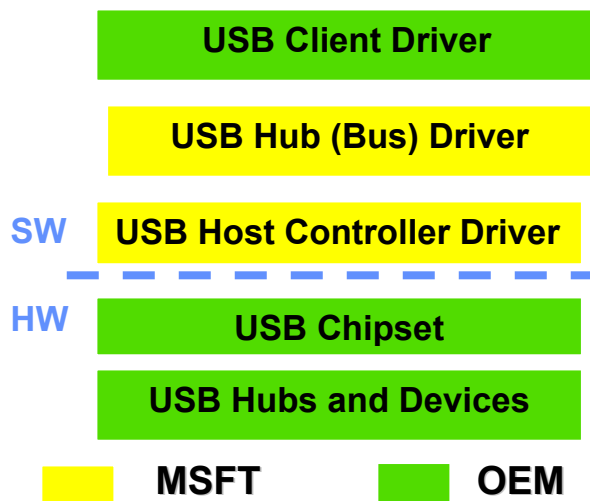


Figure 4: Windows XP USB Stack

As shown in Figure 4, Microsoft provides two key drivers to support USB, USB Hub (Bus) Driver, and USB Host Controller Driver [ii]. OEMs with USB devices develop a USB client driver to expose their USB device features. Microsoft has provided an API to allow USB client drives to have their USB device placed in USB Selective Suspend and to support USB Remote Wakeup (if supported by the USB device).

Power saving of using USB Selective Suspend Support Whitepaper

November 2002

For most USB devices, their USB client driver must provide support for USB Selective Suspend. One exception to this case is for USB HID (Human Interface Device), Windows XP has supports for USB HID and has a built-in support of USB Selective Suspend for some USB HID devices. Currently only a few USB HID devices have selective suspend support and they are listed in the input.inf file, this is often referred to as the “Good List” for USB Selective Suspend, but it is only in reference to USB HID devices. Here is the current list of USB HID devices which have USB Selective Suspend support.

- Aspire USB Mouse
- Cypress USB Mouse
- Genius USB Net Mouse Pro
- Evolution USB Mouse-Trak by ITAC
- Qtronix USB Mouse
- Other OEM products based on one of these devices

Recommendations

When using USB for internal device interconnection, OEMs should take additional steps to insure the maximum battery life of their platforms. One essential step is to ensure that all internally connected USB devices and the software for the devices support USB “Selective Suspend”. USB “Selective Suspend” is very important for the mobile platforms because the processor can only enter C3/C4 when **all** of the attached USB devices are in USB “Selective Suspend”. This is due to the IO reads that occurs from the USB host control which reads the USB polling list in system memory every few milliseconds when any of the hub’s USB devices are not in the Selective Suspend state. Without this, then the lowest C state the processor can

enter is C2, which results in a ~10% reduction in battery life.

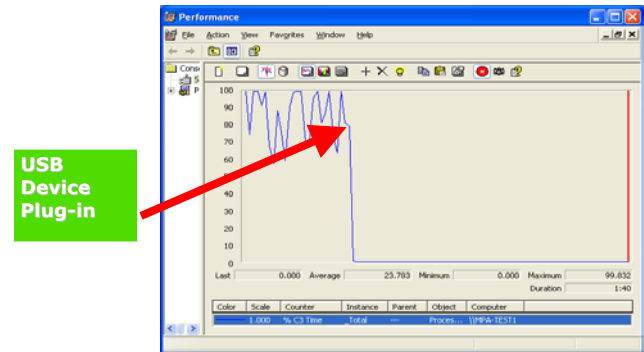


Figure 5: C3 Percentage for a USB Device without USB Selective Suspend Support

Figure 5 shows Microsoft Windows XP Performance Monitor application used to monitor various processor and operating system parameters. In this figure, the Performance Monitor application is graphing the processor’s time percentage in the C3 low power state. The C3 time percentage was monitored before and after a USB device, which does not support USB Selective Suspend, was plugged in to one of the system’s USB ports. As shown in Figure 5, the processor’s time percentage in the C3 power is 0% after the USB device is plugged in and never changes until the USB device is unplugged (not shown). This is due to the USB device not entering USB Selective Suspend when the device is idle and therefore prevents the processor from entering C3. Note the variation at the beginning of the monitoring cycle is due to other system activity (in this case it is networking activity) and reduces the C3 time percentage to be less than 100%.

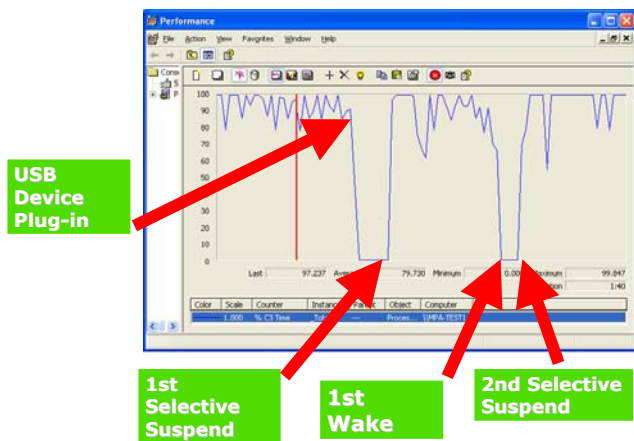


Figure 6: C3 Percentage for a USB Device with USB Selective Suspend Support

Figure 6, also shows Microsoft Windows XP Performance Monitor application graphing the processor's time percentage in the C3 low power state. The C3 time percentage was monitored before and after a USB device, which does support USB Selective Suspend, was plugged in to one of the system's USB ports. As shown in Figure 6, the processor's time percentage in the C3 power state time percentage is 0% after the USB device is plugged in. After the device's USB client driver has determine that the device is idle (in this case ~5 seconds) then the device's USB client driver causes the device to enter USB Selective Suspend and this allows the C3 power state time percentage to be ~100%. This continues until the device wakes the system due to any device active. After the device's USB client driver has determine that the device is idle again (again in this case it is ~5 seconds) then the device's USB client driver places the device into USB Selective Suspend and this allows the C3 power state time percentage to be ~100%.

Selective Suspend Support Code Walk through

To gain a better understanding of the amount of effort that it takes to support USB Selective Suspend, this section of the paper provides a

high level code walk through to explain the details. Each of the steps are described in the following sub-sections. The sample code in this document is based on sample code in the Windows XP DDK.

Idle Detection

The first step in the USB Selective Suspend process is for the USB client driver to determine that their device is idle. Idle detection is device dependent in which each device has its own definition of when the device is idle. For example a USB mouse is considered idle when there is no mouse movement for some amount of time and a USB camera is considered idle when the lens cover is close. Enhanced idle detection can be a product differentiator and lead to increase battery savings.

Submitting the Idle Notification Request

Once the USB Client driver has determined that it is idle, then it will submit an Idle Notification I/O Request Packet (IRP) to USB HUB driver. This consists of three intermediate steps. First the driver must allocate some non-page pool memory used to pass parameters for the Idle Notification, which consists of a pointer to the Idle Notification callback routine and the context passed to the callback routine. Figure 7 provides an example code in which the `FDODeviceExt` is passed as context.

```
idleCallbackInfo = ExAllocatePool(NonPagedPool,
    sizeof(struct _USB_IDLE_CALLBACK_INFO));
idleCallbackInfo->IdleCallback = IdleNotificationCallback;
idleCallbackInfo->IdleContext = (PVOID)FDODeviceExt;
```

Figure 7: Idle Notification Parameter passing Sample Code

Next the step is to allocate and initialize the IRP that is used to submit the Idle Notification parameters as shown in Figure 8. The Idle Notification is done via an Internal Device Control with IOCTL Control Code define for "Submit Idle Notification". Also the pointer to

Power saving of using USB Selective Suspend Support Whitepaper

November 2002

the Idle Notification Callback parameters is passed in the one Device IOCTL parameters and the IdleNotificationRequestComplete routine is set as the completion routine.

```
irp = IoAllocateIrp(FDODeviceExt->topOfStackDeviceObject->StackSize, FALSE);
nextStack = IoGetNextIrpStackLocation(irp);
nextStack->MajorFunction =
    IRP_MJ_INTERNAL_DEVICE_CONTROL;
nextStack->Parameters.DeviceIoControl.IoControlCode =
    IOCTL_INTERNAL_USB_SUBMIT_IDLE_NOTIFICATION;
nextStack->Parameters.DeviceIoControl.Type3InputBuffer =
    idleCallbackInfo;
nextStack->Parameters.DeviceIoControl.InputBufferLength =
    sizeof(struct _USB_IDLE_CALLBACK_INFO);
IoSetCompletionRoutine(irp, IdleNotificationRequestComplete,
    FDODeviceExt, TRUE, TRUE, TRUE);
```

Figure 8: Idle Notification IRP Allocation and Initialization Sample Code

The third step is to submit the Idle Notification IRP to the USB HUB driver. This is done by the IoCallDriver function as shown in Figure 9.

```
IoCallDriver(FDODeviceExt->topOfStackDeviceObject, irp);
```

Figure 9: Submitting Idle Notification IRP to USB HUB driver Sample Code

Idle Notification Request IRP Completion Routing

Once the USB HUB driver has processed the parameters for the Idle Notification Request, the USB HUB driver will complete the IRP and then its completion routine will be called. The completion routine should free all allocated memory which was allocated for the Idle Notification parameters and also free that allocated IRP. The completion routine must return

STATUS_MORE_PROCESSING_REQUIRED since it feed the IRP.

Idle Notification Callback

When the USB HUB driver is ready to put the USB Client driver's device in Selective Suspend, it will call the idle notification callback function.

The idle notification callback function has three tasks to do before it returns to the USB HUB driver from the callback function.

The first task for the idle notification callback is the USB Client driver must prepare its USB device for low power mode. This is device dependent and may involve sending one or more commands to the USB device to prepare the device to enter low power mode. Some devices may not support wake and therefore it is unnecessary to prepare the device. A driver may intend on having the device wake up if an event happens and therefore it would enable the device to monitor one or more events while in low power mode. For example, a USB HID mouse may choose to wake the host if any mouse movement exists.

The second task for the idle notification callback, is the USB Client driver may post a WaitWake IRP to allow the device to wake it self if it support USB remote wake. Posting a WaitWake IRP is done by calling PoRequestPowerIRP with Minor Function parameter set to IRP_MN_WAITWAKE. An example of submitting a WaitWake IRP is shown in Figure 10, with WaitWakeCallback passed as the callback routine to be called when wake occurs.

```
POWER_STATE poState;
poState.SystemState = PowerDeviceD0;

PoRequestPowerIrp(FDODeviceExt->lowerPhysicalDeviceObject,
    IRP_MN_WAIT_WAKE,
    poState,
    (PREQUEST_POWER_COMPLETE)WaitWakeCallback,
    FDODeviceExtension,
    NULL);
```

Figure 10: Submitting WaitWake IRP to USB HUB driver Sample Code

The third task for the idle notification callback, is the USB Client driver requests the PNP manager to be placed in lower power device state.

Power saving of using USB Selective Suspend Support Whitepaper

November 2002

Requesting a different power state is done by calling `PoRequestPowerIRP` with `MinorFunction` parameter set to `IRP_MN_SET_POWER`. A very important step in the Idle Notification Callback is not to return until all these tasks are complete, and therefore the USB Client driver must wait until the device is in the new requested low power state as shown in Figure 11

```
powerState.DeviceState = PowerDeviceD1;
KeInitializeEvent(&irpCompletionEvent, NotificationEvent,
    FALSE);
irpContext->FDODeviceExtension = FDODeviceExt;
irpContext->Event = &irpCompletionEvent;

PoRequestPowerIrp(FDODeviceExt->lowerPhysicalDeviceObject,
    IRP_MN_SET_POWER,
    powerState,
    (PREQUEST_POWER_COMPLETE)
    PoIrpCompletionFunc,
    irpContext,
    NULL);

if (STATUS_PENDING == ntStatus) {
    KeWaitForSingleObject(&irpCompletionEvent,
        Executive, KernelMode, FALSE, NULL);
}
```

Figure 11: Power Down Device Sample Code

Remote Wake

When the USB device is ready to wakeup, the USB HUB driver completes the posted `WaitWake` IRP. USB Client driver's `WaitWake` callback is called and will request to be in fully power mode (D0) as shown in Figure 12.

```
powerState.DeviceState = PowerDeviceD0;

PoRequestPowerIrp(deviceExt->PhysicalDeviceObject,
    IRP_MN_SET_POWER,
    powerState,
    (PREQUEST_POWER_COMPLETE)
    WWIrpCompletionFunc,
    deviceExt,
    NULL);
```

Figure 12: Power UP Request Device Sample Code

Summary

The white paper reviewed that main power issues for internally attached USB devices which can result in a ~10% battery loss if USB Selective Suspend is not supported by all of the USB Client drivers for each internally attached USB

devices. Additionally the paper provided a code walk through to allow the reader details understanding of what is necessary to support USB Selective Suspend and how a small amount of code it actually takes to support it. Finally, Intel encourages OEMs to use USB devices which support USB Selective Suspend. Additionally, Intel encourages IHVs to implement USB Selective Suspend support in their USB client drivers.

ⁱ Universal Serial Bus Specification, Version 1.1 (www.usb.org)

ⁱⁱ Microsoft Windows XP DDK