



# **Enabling Power Event Interception and Control Under Windows\* XP**

**by Chuck DeSylva**

January 1, 2005

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation Intel on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel, the Intel logo, Pentium, Intel Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 2005 Intel Corporation

## Introduction

The purpose this document is to describe a technique for getting Power Event notification from the operating system. This method is primarily aimed at controlling power on Intel® x86-based systems actively running Intel SpeedStep® Technology. And where the use of the Read Time Stamp Counter (RDTSC) instruction<sup>1</sup> varies based on the changing frequencies imposed by Intel SpeedStep Technology.

This document is intended for readers with a simple understanding of Power Management under the Microsoft Windows XP\* Operating System and at least a *simple* understanding of Intel SpeedStep<sup>2</sup>. Also an understanding of Microsoft Windows XP device drivers and Win32\* threading APIs is useful.

## Background

From the very first 4.77 MHz 8088 Intel® Processor to today's latest Pentium® 4 processor an internal system clock running at 14,318,160 Hz has existed. For reasons tied to the original PC system design, this clock is internally divided by 12, yielding a frequency of 1,193,180 Hz. That divided clock is then fed into a 16-bit counter which overflows once every 65,536 counts, effectively dividing the 1,193,180 Hz by 65,536, to result in a frequency of 18.2 Hz. This frequency was available as a hardware interrupt on the original 4.7 MHz 8088 system.

By counting the number of interrupts caused by overflows, 8 bits at a time, from the 16-bit counter/timer, a somewhat accurate hardware counter can be obtained. This is the primary method by which the system clock is read, though it is not without its caveats. One of the existing problems with reading this clock is that you can't read either the interrupt count or the hardware divider at the same time, and when reading the number of interrupts(overflows) you're not sure whether they have been read before the next one comes in or after, relative to the count in the 16-bit divider, it is possible to "synthesize" the hardware state into a coherent counter running at 1.193180 Mhz. Inverting this gives a timer counter interval of 0.838 microseconds, or 838 nanoseconds. Further, calling the system clock with great frequency is prone to error and inaccurate results because of the longer clock interval.

With the Pentium processor, Intel added an additional instruction called RDTSC (Read Time Stamp Counter). This gives software direct access to the number of clock counts the processor has experienced since its last power-on or hardware reset. With contemporary clock rates of, for example, 3.06 Ghz, that results in a timing period of only 0.326 nanoseconds.

The clock rate determined by RDTSC is 100% dependent upon the core frequency of the processor, unlike the older, 1.193180 Mhz clock. Because of this RDTSC varies it's counter as the system speed changes when Intel SpeedStep is enabled making values read by it not consistent. The purpose of this paper is to provide a method not only to force Intel SpeedStep enabled machines to provide consistent RDTSC readings, but to provide an event driven mechanism whereby power changes in the OS can be detected and safely prevented to maintain this consistency.

---

<sup>1</sup> Refer to [Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z](#)

<sup>2</sup> Refer to [Mobile Intel® Pentium® III Processors](#) [Intel SpeedStep® Technology](#)

Issues with Performance monitoring with respect to Intel SpeedStep technology and frequency

As of this writing, the Single Processor HAL (Hardware Abstraction Layer of Windows 2000\*/XP uses the same 1.193180 megahertz counter for its "QueryPerformanceCounter" API as all previous versions of Windows. However, the Multiprocessor HAL of Windows 2000 and XP use the system's own clock rate through the RDTSC processor instruction. This causes problems for software where performance monitoring is underway and Intel Speed Stepping is enabled as the values read from the RDTSC instruction are varying with respect to frequency and hence the timing values read are typically skewed.

By changing the power usage on a system running Intel Speed Stepping, such that Speed Stepping is effectively pushed to where no frequency changes occur, the RDTSC instruction becomes reliable from one read to the next and is hence useful for the purposes of code profiling. This can be done simply enough by indirect calls to the Processor Power Scheme API calls ReadProcessorPwrScheme and WriteProcessorPwrScheme. However, the Windows XP operating system does not allow for a method of being interrupted when the Power scheme changes. The purpose of this mechanism is to be provided notification in the event some process, or user intervention changes the power scheme to other than the non-frequency changing state set state while the profiling session is underway.

The method described here utilizes a power-managed driver to detect these subtleties and notify an event handler in user mode process space. In doing this, the user mode process can defer any power scheme changes to after the profiling session is completed thus retaining consistency in the profiling metrics.

## **Performance Event Interception**

The method involves first registering a power managed event callback (DeviceDispatchPower) in a driver running in Kernel Mode. The Driver source for this routine is listed in section 2.2.2.3. This routine responds to all Power Managed events in the operating system. In a standard Power-managed scenario this routine is provided to the driver so that it can change the power states on the hardware that it is controlling. Since the Driver is filtering Power events for the entire system it provides an easy scheme for filtering these events and preventing deleterious event changes by deferring them off to a time after the profiling session has completed.

Figure 1 displays this architecture from a high-level.

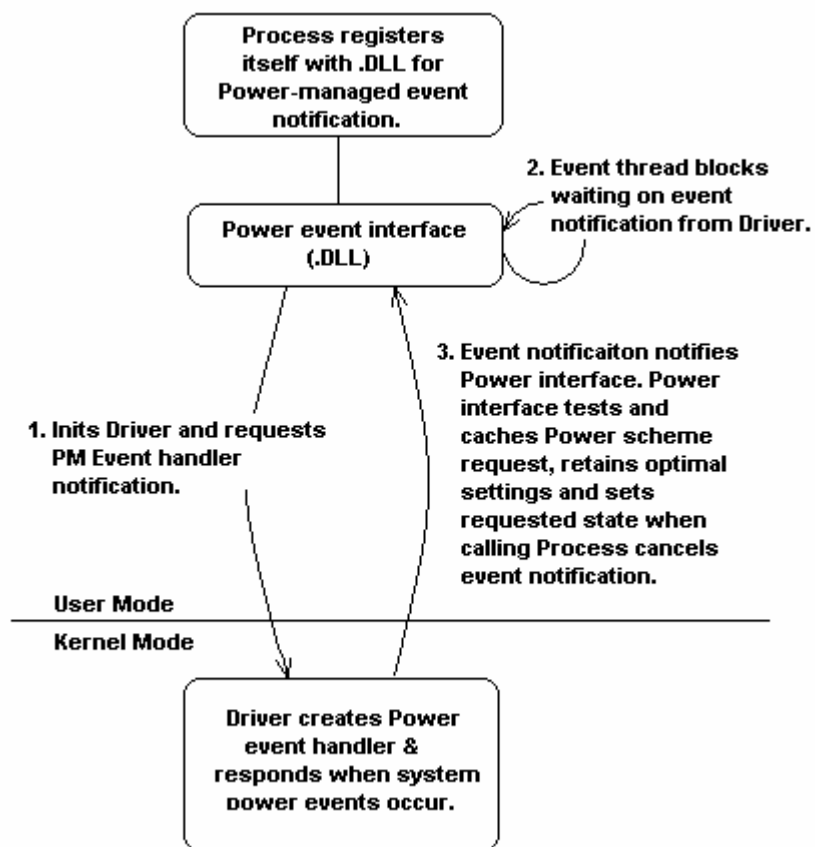


Figure 1. Power Managed Event Notification Architecture.

## Conclusion

A simple technique has been presented for getting Power Event notification from the Microsoft Windows XP operating system. With a simple driver and a few calls User mode applications can get notified when power changes occur without having to poll the Operating System to obtain it's power state. This method is particularly useful when power on Intel® x86-based systems actively running Intel SpeedStep Technology need to be properly performance monitored. And where the use of the Read Time Stamp Counter (RDTS) instruction<sup>3</sup> varies based on the changing frequencies imposed by Intel SpeedStep Technology.

## About the Author

Mr. Desylva is a Software Applications Engineering Manager in the Intel Software and Solutions Group.. He and his team are responsible for the performance optimization of cutting edge consumer software titles running on Intel Desktop systems. Prior to working on application software optimization, Chuck worked as a driver developer for Intel Corporation. He was involved in developing/deploying the first device drivers for USB, AGP (GART) and Intel's first graphics devices (i740/810(e)).

<sup>3</sup> Refer to <http://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf>.

## Additional Resources

[Download code sample](#)

### Articles

- [Enabling Power Event Interception Notification Code Sample](#)
- [Intel® Mobile Application Architecture Guide](#)
- [Windows\\* WM\\_POWERBROADCAST Messages in a Mobilized Environment](#)
- [The Big Sleep: The Art of Graceful Application Suspension](#)
- [Power Management: Designing Applications to Conserve Battery Life](#)
- [Flex Processor Muscle Where it Counts, Part 1 \(on the need for and characteristics of a good test suite\)](#)

### Intel Developer Centers

- [Mobilized Software](#)
- [Intel® Centrino Mobile Technology](#)
- [Intel® PCA Processors](#)
- [Windows\\*](#)

### Community

- [Handheld & Wireless Application Development](#)
- [Intel® VTune™ Performance Analyzer for Windows](#)
- [Intel® VTune™ Performance Analyzer 7.1 Beta](#)

### Other Resources

- [Intel® Early Access Program for Mobility](#)
- [CMP Mobilized Software Site\\*](#)

## Sources

The architecture for this project was divided in to three parts. The first being a test executable, the second a .DLL (Dynamic Link Library) which abstracts the device driver calls and lastly the device driver which is responsible for responding to power-managed events. This code was compiled with Microsoft Visual Studio\* 6.0 and Microsoft Visual Studio .Net\* 2003. Note that standard Microsoft headers have been (stdafx.cpp, stdafx.h) omitted from this listing.

## Executable

PWRCONTROLTEST.CPP:

```
//...  
  
#include <windows.h>  
  
HINSTANCE hPWRDLL;  
  
typedef bool (CALLBACK* t_fnPWRDLL_CB)(void);  
t_fnPWRDLL_CB fnPWRDLL_ESS;  
t_fnPWRDLL_CB fnPWRDLL_DSS;  
  
void main( void ) {  
    bool bIReVal = false;  
  
    hPWRDLL = LoadLibrary("PWRDLL");  
    if (hPWRDLL != NULL) {  
        fnPWRDLL_ESS = (t_fnPWRDLL_CB)GetProcAddress( hPWRDLL,  
            "ReturnOriginalPowerScheme" );  
        if (!fnPWRDLL_ESS)  
        {  
            printf("Could not get EnableSpeedStepping Proc Handle\n");  
            return;  
        }  
  
        fnPWRDLL_DSS = (t_fnPWRDLL_CB)GetProcAddress( hPWRDLL,  
            "SetAlwaysOnPowerScheme" );  
        if (!fnPWRDLL_DSS)  
        {  
            printf("Could not get SetAlwaysOnPowerScheme Proc Handle\n");  
            return;  
        }  
    }  
}
```

```

    if ( !fnPWRDLL_DSS() ) {
        MessageBox(NULL, "Unable to set Always on Power Scheme",
            "PwrControlTest.EXE", MB_OK );
        return;
    }

    // Sleep for 20 seconds, time enough to play with power scheme and
    // test the power event interception handling.
    Sleep( 20000 );

    if ( !fnPWRDLL_ESS() ) {
        MessageBox(NULL, "Unable to return original Power Scheme",
            "PwrControlTest.EXE", MB_OK );
        return;
    }

    FreeLibrary( hPWRDLL );
    return;
}

```

## **Dynamic Link Library**

```

PWRDLL.CPP: extern "C" {
#include <windows.h>
#include <winnt.h>
#include <powerprof.h>
};
//...
#define ALWAYS_ON 0x0

```

```

MACHINE_PROCESSOR_POWER_POLICY mppp;
UINT uiDynThrottleAC;
UINT uiDynThrottleDC;
bool bHasBeenDisabledFirst = FALSE;
bool bStopEvent = false;
bool bDriverEnabled = false;

static HANDLE h_DeviceDriver;
HANDLE SharedEvent;

DWORD WINAPI SetupDriverEvent(LPVOID sParms)
{
    DWORD bytesReturned;
    DWORD WaitStatus;
    UINT psIdx = 0;
    bool bRet = FALSE;

    // Create named event shared between the driver & calling process.
    SharedEvent = CreateEvent(NULL, false, false, "SharedEvent");
    if ( SharedEvent == NULL ) {
        return 0;
    }

    // The named event has been created and can now open a handle to it
    if (!DeviceIoControl( h_DeviceDriver, IOCTL_PWRNTFY_EVENT, NULL, 0,
        NULL, 0, &bytesReturned, NULL )) {
        return 0;
    }

    if (!DeviceIoControl( h_DeviceDriver, IOCTL_PWRNTFY_EVENT_START, NULL, 0,

```

```

        NULL, 0, &bytesReturned, NULL )) {
        return 0;
    }

while ( !bStopEvent )
{
    WaitStatus = WaitForSingleObject( SharedEvent, INFINITE );
    if ( WaitStatus == WAIT_OBJECT_0 )
    {
        GetActivePwrScheme(&psIdx);
        if ( ReadProcessorPwrScheme(psIdx, &mppp) )
        {
            uiDynThrottleAC = mppp.ProcessorPolicyAc.DynamicThrottle;
            uiDynThrottleDC = mppp.ProcessorPolicyDc.DynamicThrottle;
        }

        mppp.ProcessorPolicyAc.DynamicThrottle = ALWAYS_ON;
        mppp.ProcessorPolicyDc.DynamicThrottle = ALWAYS_ON;

        WriteProcessorPwrScheme(psIdx, &mppp);
        SetActivePwrScheme(psIdx, NULL, NULL);
    }
}
return 1;
}

SPEEDSTEPDLL_API bool ReturnOriginalPowerScheme( void ) {
    UINT psIdx = 0;
    bool bRet = FALSE;
    bStopEvent = TRUE;

```

```

if ( bHasBeenDisabledFirst == true ) {
    if ( GetActivePwrScheme(&psIdx) ) {
        if ( ReadProcessorPwrScheme(psIdx, &mppp) ) {
            mppp.ProcessorPolicyAc.DynamicThrottle = uiDynThrottleAC;
            mppp.ProcessorPolicyDc.DynamicThrottle = uiDynThrottleDC;

            if ( WriteProcessorPwrScheme(psIdx, &mppp) ) {
                if ( SetActivePwrScheme(psIdx, NULL, NULL) ) {
                    bRet = TRUE;
                }
            }
        }
    }
}
return bRet;
}

```

```

SPEEDSTEPDLL_API bool SetAlwaysOnPowerScheme( void ) {
    UINT psIdx = 0;
    bool bRet = false;

    if ( bHasBeenDisabledFirst == false ) bHasBeenDisabledFirst = true;

    if ( GetActivePwrScheme(&psIdx) ) {
        if ( ReadProcessorPwrScheme(psIdx, &mppp) ) {
            uiDynThrottleAC = mppp.ProcessorPolicyAc.DynamicThrottle;
            uiDynThrottleDC = mppp.ProcessorPolicyDc.DynamicThrottle;
            bRet = true;
        }
    }
}

```

```

}

mppp.ProcessorPolicyAc.DynamicThrottle = ALWAYS_ON;
mppp.ProcessorPolicyDc.DynamicThrottle = ALWAYS_ON;

if (WriteProcessorPwrScheme(psIdx, &mppp)) {
    if (SetActivePwrScheme(psIdx, NULL, NULL)) {
        bRet = true;
    }
}

HANDLE h_SDEThread = CreateThread(NULL, 0, SetupDriverEvent, NULL, 0, NULL);
return bRet;
}

BOOL APIENTRY DIIMain( HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved )
{
    //...
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            if ( !bDriverEnabled )
            {
                //...
                h_DeviceDriver = CreateFile( DRIVER_NAME_DYN_LOAD,
                                             GENERIC_READ | GENERIC_WRITE,
                                             0, NULL, OPEN_EXISTING,
                                             FILE_ATTRIBUTE_NORMAL, NULL );
            }
        }
    }
}

```

```

        if ( h_DeviceDriver == INVALID_HANDLE_VALUE )
            bDriverEnabled = false;
        else bDriverEnabled = true;
    }
    return bDriverEnabled;

case DLL_PROCESS_DETACH:
    if ( bDriverEnabled )
    {
        bStopEvent = true;
        //...
        CloseHandle( SharedEvent );
        CloseHandle( h_DeviceDriver );
        bDriverEnabled = false;
    }
    return bDriverEnabled;

case DLL_THREAD_ATTACH:
case DLL_THREAD_DETACH:
default:
    return true;
}
}

```

## **Driver**

PWRNTFY.C:

```

VOID DriverUnload( PDRIVER_OBJECT pdrvo ) {
    PDEVICE_OBJECT pdevo = pdrvo->DeviceObject;
    UNICODE_STRING uniDosDeviceName;

```

```

    if (SharedEventHandle != NULL) ZwClose( SharedEventHandle );
    KeClearEvent( SharedEvent );
    RemovePowerCallback();

    RtlInitUnicodeString( &uniDosDeviceName, UNI_DOS_DEVICE_NAME );
    IoDeleteSymbolicLink( &uniDosDeviceName );
    IoDeleteDevice( pdevo );
    return;
}

NTSTATUS DriverCreateClose( PDEVICE_OBJECT pdevo, PIRP pirp ) {
    PDEVICE_EXTENSION pde = pdevo->DeviceExtension;
    pirpstk = IoGetCurrentIrpStackLocation( pirp );
    pirp->IoStatus.Status = STATUS_SUCCESS;
    pirp->IoStatus.Information = 0;
    IoCompleteRequest(pirp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

NTSTATUS DriverDeviceControl( PDEVICE_OBJECT pdevo, PIRP pirp ) {
    NTSTATUS nts = STATUS_SUCCESS;
    PDEVICE_EXTENSION pde = pdevo->DeviceExtension;
    UNICODE_STRING EventName;

    pirpstk = IoGetCurrentIrpStackLocation( pirp );
    pirp->IoStatus.Information = 0;

    switch (pirpstk->Parameters.DeviceIoControl.IoControlCode) {
    case IOCTL_PWRNTFY_EVENT:
        RtlInitUnicodeString(&EventName, L"\\BaseNamedObjects\\SharedEvent");

```

```

        SharedEvent = IoCreateNotificationEvent(&EventName, &SharedEventHandle);
        if (SharedEvent != NULL) nts = STATUS_SUCCESS;
        else nts = STATUS_UNSUCCESSFUL;
        break;
case IOCTL_PWRNTFY_EVENT_START:
    SetupPowerCallback();
    nts = STATUS_SUCCESS;
    break;
default:
    nts = STATUS_INVALID_PARAMETER;
    break;
}
pirp->IoStatus.Status = nts;
IoCompleteRequest( pirp, IO_NO_INCREMENT);
return nts;
}

VOID DeviceDispatchPower(PVOID CallbackContext, PVOID Argument1, PVOID Argument2)
{
    KeSetEvent(SharedEvent, 0, FALSE);
    return;
}

```

