

Using ACPI to Report APML P-State Limit Changes to Operating Systems and VMM's

August 7, 2009

AMD

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS" AND AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS PUBLICATION AND RESERVES THE RIGHT TO MAKE CHANGES TO SPECIFICATIONS AND PRODUCT DESCRIPTIONS AT ANY TIME WITHOUT NOTICE. The information contained herein may be of a preliminary or advance nature and is subject to change without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. EXCEPT AS SET FORTH IN AMD'S STANDARD TERMS AND CONDITIONS OF SALE, AMD ASSUMES NO LIABILITY WHATSOEVER, AND DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT.

AMD's products are not designated, intended, authorized or warranted for use as components in systems intended for surgical implant in the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's products could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

Introduction and Context –

This paper applies to server systems that include a management controller (referred to as a “service processor” or “SP” herein).

Starting with Istanbul processors, AMD’s new sideband interface (APML) is designed to allow an external entity to monitor and control processor activity. Using the Remote Management Interface (RMI) mode of APML, data-center management systems can impose P-State Limits on individual servers as necessary to manage overall thermal loads in a data center. Thus, the P-State Limit of an individual server may change from time to time under external control. In response to an APML P-State Limit command, the CPU hardware will change the P-State, if necessary, to conform to the new limit. When this happens, the OS’s state information will be stale, so it is desirable to inform the OS or VMM of the change.

This paper describes a method for using existing ACPI pathways to inform the OS when P-State Limits change under external (APML) control. This method is expected to work with any OS that supports ACPI `_PPC` objects. No changes to OS code should be needed.

(Note: Throughout this paper, the term “OS” will be used generically to refer to the software that invokes ACPI AML routines, whether it be an operating system, a hypervisor, or a virtual machine manager.)

The first part of this paper provides introductory context and an overview of the process. The second part is a BIOS Implementers’ Guide, which includes detailed register setting information and partial reference code.

A Quick Walkthrough of the Overall Process –

The proposed solution is as follows:

- When the Service Processor issues an APML command to change the P-State Limit, the platform should invoke an SCI (System Control Interrupt). Platform implementers can choose one of the two methods described below.
 - The SCI will be handled by the OS’s (pre-existing) ACPI handler.
 - The ACPI handler will interrogate the south bridge (by reading the `GPEX_STS` register), determine which of multiple possible sources caused the SCI, and invoke the appropriate AML (ACPI Machine Language) routine.
 - The AML routine will interrogate hardware to get the new HW P-State Limit value
 - The AML routine will also examine any platform sources of P-State Limits (e.g. AC/DC state).
 - For each `_PPC` object, the AML routine will find the most restrictive value from all the above sources and update the `_PPC` object, if necessary, to reflect it.
 - For each `_PPC` object whose value has changed, the AML routine will then issue a `_PPC Notify (0x80)` message to the OS, indicating that that `_PPC` should be re-evaluated.
 - After re-evaluating all changed `_PPC` objects, the OS will have an accurate understanding of the current P-State Limits. (A single P-State limit governs all the cores in a processor, but different processors may have different limits.)

Note: the above description glosses over details (including some MP aspects) that are discussed in more detail below.

Detailed Description of the Process –

SCI Generation –

On AMD platforms, SCI Interrupts are generated by the south bridge and passed to the x64 processor as special messages on the south-bridge link. Multiple hardware sources may generate an SCI, and each of these sources is connected to its own GPIO pin on the south bridge.

After an APM P-State Limit command the platform should invoke an SCI. Platform implementers may choose one of the following SCI-generation methods.

Method #1 (Software SCI) –

On platforms with AMD south bridges (SB700, A12 and newer), a software-based method of SCI generation is available. One implementation of this method is to configure the system to cause an SMI whenever the hardware P-State Limit changes. The SMI handler can then write to I/O ports in the south bridge to activate a dedicated GP Event. (This can be thought of as a loop-back action, with the GPIO associated with the GP Event being activated by internal software commands rather than an external hardware signal.)

This method is preferred, since it will inform the OS of P-State Limit changes caused by HTC (Hardware Thermal Control) or STC (Software Thermal Control) as well as APM.

Note: Platform implementers may find it convenient during this SMI to read the new P-State Limit from the processor's MSR and store it in ACPI data space for later use by the AML routine.

The reference code below uses this method.

Method #2 (Hardware Signal) –

Many platforms have a hardware signal (trace) between the SP and a General Purpose Event (GPIO) pin on the south bridge. By activating this signal, the SP can cause the south bridge to create an SCI.

Any platform that is fully OPMA compliant will have such a signal trace, since Section 8.17 of the OPMA Specification states:

“The motherboard must route the MCARD_SCI_INT_L signal from the OPMA connector to an input on the Southbridge that is capable of generating an SCI (System Control Interrupt) to the ACPI subsystem.”

Many non-OPMA systems have a similar signal.

Since this method requires the SP to control the signal pin, platform developers who wish to use this method will need to have control over the code running on the SP (i.e. source code). The reference code below does not demonstrate this method. Again, method #1 (Software SCI generation) is preferred over this method.

SCI Source Determination –

When the OS's ACPI handler responds to the SCI, it will interact with the south bridge to determine the source of the SCI, by finding the active bit in the GPEX_STS register. This register reflects all of the possible General Purpose Event sources. (General Purpose Events correspond to GPIO pins that have been configured to cause an SCI when activated by a platform event. Even if the SCI was triggered by software, it will have an associated GPIO pin and a corresponding bit in GPEX_STS.)

Invoking the Correct AML Routine –

Once the General Purpose Event has been determined, the OS's ACPI handler invokes the appropriate AML routine. This invocation is done with the assistance of the system ROM, which contains a platform-specific mapping between General Purpose Events and AML routines.

AML Processing Following a P-State Limit Command –

In response to an APM P-State Limit command (and subsequent SCI), the appropriate AML routine has now been invoked. It will need to do the following:

1. Determine the value of the newly imposed P-State Limit
2. Update the _PPC objects, if necessary. (A _PPC object is a per-processor ACPI construct in which the platform indicates the maximum-performance P-State currently available to the OS.)
3. Notify the OS to re-evaluate any _PPC objects that have changed.

1. Determine the value of the newly imposed P-State Limit, more detail –

After APM sets a new P-State Limit, the limit will be reflected in a processor MSR (MSRC001_0061 P-State Current Limit Register). AML code needs to know this new value to update the _PPC object(s). However, AML code is not able to directly access MSR's. Therefore, AML code needs to pursue an alternate method to determine the new limit. If SCI method #1 is used, the MSR should be read directly while servicing the SMI that creates the software SCI. The value should be stored in ACPI data space for later use by the AML routine. This method is preferred and is shown in the reference code below.

If SCI method #2 is used, there will be no SMI phase, so the MSR cannot be read directly. On Hydra-M die (Istanbul and Magny-Cours) a selection of MSR values will be mirrored in PCI configuration space. The processor's P-state Limit register is not directly available, but its three component inputs are available to AML code through PCI Config space. They can be read and compared to determine the value of the processor's current P-State Limit. The three component P-State values are:

- 1) The HTC (Hardware Thermal Control) P-State Limit register
- 2) The STC (Software Thermal Control) P-state Limit register
- 3) The APM P-State Limit register

AML code should use a PCI Config-Space Operation Region to read these three registers and compare them to find the most restrictive value. That value will be the processor's overall Current P-State Limit.

Using one of these methods, the AML code now knows the value of the processor's current P-State Limit.

2. Update the `_PPC` objects and Notify the OS, more detail –

The `_PPC` object is used to communicate a platform-imposed P-State Limit to the OS. The processor's current P-State Limit (discussed above) is not the only source of platform-imposed P-state limits. Other platform items, such as "docked vs. undocked" status "AC vs. DC power" and others can also affect the platform's current P-State limit. The `_PPC` object (within each processor object) should change as necessary to reflect the most restrictive P-state limit currently being requested by all the various sources. For this reason, `_PPC` objects should not be counted on to act as variables storing the value of platform-imposed P-State Limits. Each platform-imposed limit should be stored in an ACPI variable separate from the `_PPC` objects.

Whenever there is an event causing a change in any of the P-State Limit sources, AML code should examine and compare all the sources to determine the most restrictive value. For each `_PPC` object in the system, if the new value matches the existing value, nothing needs to be done. If the new value is different from the existing `_PPC` value, the `_PPC` should be updated to reflect the new value and a Notify 80h should be sent so that the OS will re-evaluate that `_PPC` object. After re-evaluating all the `_PPC` objects that changed, the OS will have an up-to-date understanding, allowing it to make accurate scheduling decisions.

Part Two - BIOS Implementation Guide and Reference Code

Introductory Notes

- Most of the sample code pieces are platform and codebase specific, therefore they may not be able to build directly on certain code bases without necessary modifications
- All sample code here is BIOS code base independent and is provided for the purpose of demonstrating the BIOS logic only.

1. Configure processor registers to generate SW SMI on P-state limit change

Time to execute: Late BIOS POST (after PCI enumeration)

```
// a) Configure Thermal Local Vector Table Entry (APIC330) to allow delivering of SMI type message
Address64 = 0xf00000330; // Local APIC register 0x330
Value32 = 0x200; // Set [10:8] = 2 for SMI type
PciRootBridgeIo->Mem.Write (This, PciWidthUint32, Address64, 1, &Value32);

// b) Configure HTC Register (F3x64)
Address64 = PCI_ADDRESS (0, 0x18, 0x3, 0x64); // F3x64
PciRootBridgeIo->Pci.Read (This, PciWidthUint32, Address64, 1, &value32);
Value32 |= 0xc0; // [7:6] – Set PSIApicLoEn & PSIApicHiEn: enables local interrupt when P-
state limit // has changed
PciRootBridgeIo->Pci.Write (This, PciWidthUint32, Address64, 1, &Value32);

// c) Configure SMI Trigger IO Cycle Register (MSRC001_0056) to enable the local-event-triggered SMI message
// to generate software SMI (by issuing I/O cycle to south bridge) (**Assuming CPU IoTrap is NOT enabled)
MsrValue64 = ReadMsr (0xc0010056);
If (!(MsrValue64 & 0xffff)) { // Has [IoPortAddress] not been programmed yet?
    MsrValue64 = 0x02cc00b0; // Send OEM_APML_SW_SMI (0xcc) to 0xb0 port
    WriteMsr (0xc0010056, MsrValue64);
}
```

2. Enable SB700 GPIO66 to generate GPE event (for A12 and newer)

Time to execute: ACPI_ENABLE SMI (Note: this can also be put in late BIOS POST)

```
// Set PMIO_10h[7] = 1: ACPI_EVENT[30] is routed to use GPIO66 as input
SmiIoWrite8 (0xcd6, 0x10);
Value8 = SmiIoRead8 (0xcd7);
SmiIoWrite8 (0xcd7, Value8 | BIT7);

// Set SMBUS:0x7e[5] = 0: Set GPIO66 as GPIO_OUT
// Set SMBUS 0x7e[1] = 1: Initial GPIO66 = high level
Value32 = MmPci32 (0, 0, 0x14, 0, 0x7e);
Value32 &= ~BIT5;
Value32 |= BIT1;
MmPci32 (0, 0, 0x14, 0, 0x7e) = Value32;
```

3. Implement software SMI handler - oemApmIcallback

Time to execute: SMI initialization (Generally BIOS codebase specific)

```
typedef struct {
    UINT32 OldNodePs; // ONPS: [3:0]-P0; [7:4]-P1; and so on
    UINT32 NewNodePs; // NNPS: [3:0]-P0; [7:4]-P1; and so on
} NV_STORE_STRUCT;

// Global variable to be shared by SMI handler and ASL
NV_STORE_STRUCT NvStore;

// This SW SMI handler reads the P-state limit register value from every processor node and save them into a shared
// memory area which can be later accessed by ASL code, then triggers a 'software SCI'.
VOID
oemApmIcallback (
    VOID
)
{
    UINT32 Value32;
```

```

UINT64 msrValue64;

// Read P-state limit value for every processor node
tmpNewPsMap = readCurrentPstateLimitForAllNodes();
tmpOldPsMap = NvStore->OldNodePs;
NvStore->NewNodePs = tmpNewPsMap;

// SB700 A12 and newer: Toggle SMBUS:0x7e[1] (0 then back to 1): Setting GPIO66 to low level triggers GPE event
Value32 = MmPci32 (0, 0, 0x14, 0, 0x7e);
Value32 &= ~BIT1;
MmPci32 (0, 0, 0x14, 0, 0x7e) = Value32;           // This causes SB700 to generate a SCI (on GPE[30] event)

StallInternalFunction (1000);                       // Delay 1ms
Value32 = MmPci32 (0, 0, 0x14, 0, 0x7e);
Value32 |= BIT1;
MmPci32 (0, 0, 0x14, 0, 0x7e) = Value32;
}

```

4. Fixup OperationRegion NVST with runtime address of NvStore

Time to execute: Late BIOS POST (note: Entirely codebase specific)

N/A

5. ASL Sample Code

Time to execute: OS run time (note: This is codebase and platform specific)

```

OperationRegion(NVST, SystemMemory, 0x55aa55aa, 0x55aa55aa)
Field (NVST, AnyAcc, Lock, Preserve) {
    ONPS, 32,           // OldNodePs - Old P-state values of all nodes
    NNPS, 32,           // NewPsPack - New P-state values of all nodes
}

Name(PSL0,0)           // New P-state Limit in processor node 0
Name(PSL1,0)           // New P-state Limit in processor node 1

Name(CPL0,0)           // Current P-state Limit for CPU0
Name(CPL1,0)           // Current P-state Limit for CPU1
Name(CPL2,0)           // Current P-state Limit for CPU2
Name(CPL3,0)           // Current P-state Limit for CPU3
Name(CPL4,0)           // Current P-state Limit for CPU4
Name(CPL5,0)           // Current P-state Limit for CPU5
Name(CPL6,0)           // Current P-state Limit for CPU6
Name(CPL7,0)           // Current P-state Limit for CPU7

Scope(\_PR) {
    Processor (C000, 0x00, 0x00000410, 0x06) {
        Method (_PPC, 0) {
            Store(UPPC(CPL0,PSL0),CPL0)
            Return (CPL0)
        }
    }
    Processor (C001, 0x01, 0x00000000, 0x00) {
        Method (_PPC, 0) {
            Store(UPPC(CPL1,PSL0),CPL1)
            Return (CPL1)
        }
    }
    Processor (C002, 0x02, 0x00000000, 0x00) {
        Method (_PPC, 0) {

```

```

        Store(UPPC(CPL2,PSL0),CPL2)
        Return (CPL2)
    }
}
Processor (C003, 0x03, 0x00000000, 0x00) {
    Method (_PPC, 0) {
        Store(UPPC(CPL3,PSL0),CPL3)
        Return (CPL3)
    }
}
Processor (C004, 0x04, 0x00000000, 0x00) {
    Method (_PPC, 0) {
        Store(UPPC(CPL4,PSL1),CPL4)
        Return (CPL4)
    }
}
Processor (C005, 0x05, 0x00000000, 0x00) {
    Method (_PPC, 0) {
        Store(UPPC(CPL5,PSL1),CPL5)
        Return (CPL5)
    }
}
Processor (C006, 0x06, 0x00000000, 0x00) {
    Method (_PPC, 0) {
        Store(UPPC(CPL6,PSL1),CPL6)
        Return (CPL6)
    }
}
Processor (C007, 0x07, 0x00000000, 0x00) {
    Method (_PPC, 0) {
        Store(UPPC(CPL7,PSL1),CPL7)
        Return (CPL7)
    }
}
}

// Update capability value per P-state Limit change
Method(UPPC,2) {
    Store("Entering _PPC method",Debug)
    If(LNotEqual(Arg0, Arg1)) { // P-state limit has changed
        Store("---->_PPC evaluated",Debug)
        Store(Arg1,Arg0)
        Store(Arg0, IO80)
        Return(Arg0)
    }
    Return (0)
}
} // end of _PR

Scope(\_GPE) {
    Method(_L1E) {
        // Update local variable with new P-state limit value
        Store(NNPS,ONPS)
        Store(And(NNPS, 0x0F),\PSL0)
        Store(And(ShiftRight(NNPS,4), 0x0F),\PSL1)

        // Notify OS that _PPC object has changed and need to be re-evaluated
        if (LNotEqual(\PSL0,\CPL0)) {
            Notify(\_PR.C000,0x80)
            Notify(\_PR.C001,0x80)
            Notify(\_PR.C002,0x80)
            Notify(\_PR.C003,0x80)
        }
        if (LNotEqual(\PSL1,\CPL4)) {
            Notify(\_PR.C004,0x80)
            Notify(\_PR.C005,0x80)
            Notify(\_PR.C006,0x80)
            Notify(\_PR.C007,0x80)
        }
    }
} // End of _GPE

```


Appendix A Functional Validation

Once the BIOS implementation has been completed, the implementer may wish to test it for proper functionality. This section provides a simple but effective way of verifying that the OS does change its behavior regarding CPU P-State assignment in response to CPU P-State limit changes. The following procedure has been validated with 2x Shanghai parts on Windows Vista32 Service Pack 1, and on Istanbul CPU's.

Test environment:

- Toonie2 (RD890+SB700) with 2P (Shanghai revC)
- Operating system: Vista32 SP1
- Tools and utilities:
 - ***RW_Everything*** – A powerful freeware to dump almost all the PC hardware registers (click [here](#) to download)
 - ***SuperPi*** – A freeware often used as CPU benchmark test tool ([SuperPi Wiki page](#))

Of course you do not necessarily have to use these two utilities for the test – there are lots of similar alternatives available out there, as long as they can perform the same as required in the test procedure.

Steps to Test:

- 1) Boot system into Vista
- 2) Launch RW_Everything, open two windows:
 - a. PCI – Select Bus 00, Device 18, Function 03 (AMD CPU Misc Control Register on Node0)
 - b. CPU MSR Registers – Click <F5> to add following MSR registers to monitor:
 - 0xc0010061 (P-state Current Limit Register, [2:0] = **CurPstateLimit**)
 - 0xc0010062 (P-state Control Register, [2:0] = **PstateCmd**)
 - 0xc0010063 (P-state Status Register, [2:0] = **CurPstate**)
- 3) Now in the PCI window, modify the STC Register (F3x68 of node0) as DWORD to 0x30000020, this should change the node0 current P-state limit value from P0 to P3
- 4) In MSR window, observe all the 4 cores on node0 have CurPstateLimit changed to 3
- 5) Launch SuperPi program and start calculating PI to say, 512K digits
- 6) **In MSR window, observe PstateCmd of all 4 cores on node0 are set to 3** (this is set by OS, we should NOT see any other values other than 3 on node0)
- 7) Repeat steps 3) – 6):
 - Set F3x68 = 0x20000020, **observe CurPstateLimit = 2; OS takes turn for all 4 cores to repeatedly program PstateCmd to 2 and 3;**
 - Set F3x68 = 0x10000020, **observe CurPstateLimit = 1; OS takes turn for all 4 cores to repeatedly program PstateCmd to 1, 2 and 3;**
 - Set F3x68 = 0x00000020, **observe CurPstateLimit = 0; OS takes turn for all 4 cores to repeatedly program PstateCmd to 0, 1, 2 and 3;**
- 8) In PCI window, select Bus 00, Device 19, Function 03 to modify P-state limit on node1 and repeat the above steps, we should observe the same result as on node0.