



Device Driver and BIOS Development for AMD Multiprocessor and Multi-Core Systems

A White Paper written for
Advanced Micro Devices Inc.

Camden Associates
San Bruno, Calif.
August 2006

2006 Advanced Micro Devices, Inc. All rights reserved.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. 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 designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product 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.

Trademarks

AMD, the AMD Arrow logo, AMD Athlon, and AMD Opteron and combinations thereof are trademarks of Advanced Micro Devices, Inc.

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.

Microsoft and Windows registered trademarks of Microsoft Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Table of Contents

Introduction.....	4
Multiprocessor Development Challenges	5
Simultaneous Thread Execution	5
Synchronizing Access, Enforcing Run Order.....	7
PCI Bus Device Development Notes.....	8
BIOS Notes for Multiprocessing	9
BIOS Notes for Systems Management Interrupt	9
BIOS Notes for Dual-Core Specific Resources	10
References.....	11
Acknowledgements.....	12

Introduction

Device drivers and Basic Input/Output System (BIOS) code for Windows devices are as old as Windows – and each generation of hardware has presented both new opportunities and new challenges for device driver and BIOS developers. The move from 16-bit Windows 3.x to the 32-bit Windows 95 was the first in a long line of moves -- from DOS-based Windows 98 to the Windows NT kernel was another. Along the way, device makers have accommodated new processor architectures, new device types, new buses, and new memory subsystems.

More than a decade ago, many hardware and software makers learned to create multiprocessor x86-based servers and workstations, and conquered the device driver and BIOS challenges that multiprocessing represented. Until recently, however, developers of mobile devices and other uniprocessor systems rarely had to address issues that would typically be associated with multiprocessor systems.

For example, the primary issues that the mobile industry typically faced involved maximizing performance in a battery-operated environment, handling sleep states and non-standard display and I/O subsystems, and low-voltage considerations. Thus, many device drivers were tuned to maximize reliability and performance in those single-processor mobile environments ... and many haven't even been tested in a multiprocessor system, even in the manufacturer's own test lab.

Similarly, BIOS developers have also learned to accommodate advances in processor technology, improving and extending the Basic Input/Output System to offer ever-increasing performance levels on multiprocessor and multi-core mobile, desktop and server systems. Indeed, the advent of dual-core microprocessor technology has, for the first time, introduced multiprocessing into the mainstream of personal computing, spanning mobile, desktop and server systems. Today, high-performance computers of all shapes and sizes increasingly have dual-core processors and chipsets – and in rare occasions, might experience unexpected reliability issues with device drivers and BIOS code which has not been designed, developed or rigorously tested in a multiprocessor or multi-core environment.

As dual-core processors become prevalent in the mass market for uniprocessor devices – and as processors with even more than two cores begin to enter the market– such issues will become more urgent in that domain. However, this is an issue that spans the personal computing spectrum.

This White Paper will explore the specific challenges faced by architects, developers and testers working in a multicore environment, with specific attention paid to device driver and BIOS issues that now affect uniprocessor devices as much as they affect multiprocessor devices. Many of the points in this paper will be presented from a uniprocessor computing perspective, as developers in this domain may have less

experience working with multiprocessor systems. However, these technological issues are necessarily applicable to multiprocessor systems as well.

Multiprocessor Development Challenges

For the purposes of this White Paper, any device with more than one processor core – that is, a Central Processing Unit or CPU – will be considered a multiprocessor device. This may be a device with more than one single-core processor, or a single processor that contains two or more cores within the silicon, or indeed more than one processor, each having multiple cores. While the hardware and performance characteristics will vary depending on the actual hardware used, the important issue is that the Windows operating system automatically scales to exploit all available CPUs, and will schedule software threads to run on all of them.

These threads will be from the operating system kernel and other OS applications and utilities, end users applications, as well as kernel-mode and PCI (Peripheral Component Interconnect) device drivers. Kernel threads will potentially be running on all processor cores, as will device drivers. Some drivers may be fully contained within a single core, and others may have different threads running on two or more cores.

While developers can have some control over thread affinity – that is, determining which core(s) are used for which thread(s) and application(s) – generally, the Windows scheduler will determine where threads are assigned and run based on the system workload and its own scheduling algorithms.

The modern Windows kernel is fully preemptive. That is, any task running in the kernel can be preempted, or interrupted, any time the thread scheduler chooses, or when an event occurs that causes a higher-priority task to be executed. Of course, there are situations when the kernel should not be preempted: when it's handling interrupts, when a lock is in place, or when the kernel is executing the scheduler itself.

In almost all cases, Windows device drivers run at a high priority level, and when run on a single-processor system with only a single core, a lock set by the driver will prevent the driver from being preempted on its processor. However, on a multiprocessor system, this may not be the case, because events occurring on a different processor may preempt the device driver, interfere with reentrancy or concurrency of the driver's routines, or cause resource contention issues related to simultaneous thread execution.

Simultaneous Thread Execution

A separate challenge is that many developers may not realize that Microsoft provides two separate sets of Windows XP and Windows Server 2003 kernels. One set is written for single processor systems, the other for multiprocessor systems. In most ways, the kernels are identical, at least as far as device drivers are concerned. However, thread scheduling

and execution is different between the two kernel versions. This may be particularly surprising for uniprocessor developers, who may not have encountered the multiprocessor kernel in their target platforms until now. (A uniprocessor system with two or more cores uses the multiprocessor kernel.)

On a single processor system, all code – kernel, drivers, applications – run on a single processor. On a multiprocessor system, the workload is distributed across all processors. The highest priority ready thread (that is, one which is not waiting for I/O or for release from a locked state) runs at all times, on one of the processors. Windows schedules the next-highest priority ready threads across the processors, depending on system workload. Some of those threads may also be running on the same processor that's running in the highest priority thread.

Because more than one thread is running at the same time, it is likely that more than one driver routine (at the same or different priority level) may be running at the same time on different processors. Also, it is very possible that different code threads from a single driver will be running on more than one processor simultaneously – and those threads may not be synchronized.

The difference between single-processor and multiprocessor systems becomes apparent when a device interrupts the processor. Initially, the IRQL (interrupt request level) for that processor is at `PASSIVE_LEVEL`, its lowest state where interrupts are handled by Windows.

The device's driver raises its IRQL to `DIRQL`, a hardware-controlled state with a high priority, to run the interrupt service routine. The interrupt service routine stops the device from interrupting again, queues the I/O response, lowers the processor's IRQL to `DISPATCH_LEVEL`, which indicates that processing is being handled by Windows, and exits. When the I/O is complete, the driver sets the IRQL back to `PASSIVE_LEVEL` again. There is no chance that there can be another device interrupt until the IRQL is set back to `PASSIVE_LEVEL` – essentially locking the system until the driver releases the device.

However, on a multiprocessor system, an IRQL-level lock is associated only with a single CPU or core. A separate thread running on a different core – which might be associated with that same device driver or with a different driver entirely – might be running with its IRQL at `PASSIVE_LEVEL` while the interrupt handler on the first core has raised its own IRQL to `DIRQL` or `DISPATCH_LEVEL`. If a second interrupt comes through, on the second core, it could disrupt the I/O, deadlock resources, or overwrite memory.

The solution is to issue explicit or implicit locks that run across multiple processors – an extra step that a uniprocessor device developer may not have even considered taking. Even if a lock was written, if the device and driver was not tested in a multiprocessor environment, the lock code may never have been tested, debugged or tuned. This could affect synchronization and run order.

Synchronizing Access, Enforcing Run Order

Whether a device driver is running on a single processor core, or on a multiprocessor or multi-core system, it is essential that only one thread at a time can access critical data, such as data that might be overwritten by the device driver or by another process. It is also important that, when multiple threads require access to that data, the threads access that data in the proper logical order. Also, a set of operations performed on data must be performed as a single unit (atomically), without the possibility that the data might be accessed or modified by another thread before that set of operations is complete.

When the device driver is running on a traditional uniprocessor system, it is unlikely that synchronization issues or run-order issues will appear. However, they could easily occur in multiprocessor or multi-core systems. Indeed, without synchronization, read or write operations performed by drivers running in different thread contexts could be interleaved or happen in the wrong order, or be interfered with during an atomic set of operations. This could cause a driver to use stale or out-of-date data, or to corrupt another driver's data. This might cause a driver or system crash – or create more subtle errors that go undetected.

In some cases, these problems may be introduced by the optimizing compiler, which can aggressively reorder some instructions to improve performance – or even eliminate some instructions that it believes are redundant (but which the developer placed there as a safeguard against synchronization issues). A solution is to use the [volatile](http://msdn2.microsoft.com/en-us/library/12a04hfd.aspx) keyword in C/++, which instructs the compiler that the variable associated with that keyword might be changed by something outside the context of the current thread. The compiler then rereads that variable each time it is referenced, and writes the value of the variable to memory each time it is assigned, thereby safeguarding against most unwanted side effects.

Developers should also consider the use of Windows' [InterlockedXxx](http://www.microsoft.com/whdc/driver/kernel/locks.msp#InterlockedXxx) and [ExInterlockedXxx](http://www.microsoft.com/whdc/driver/kernel/locks.msp#ExInterlockedXxx) routines, which perform common arithmetic and logical operations atomically, using high IRQL levels to ensure that these operations may not be preempted. These routines are designed for high performance, and are recommended for use in device drivers.

There are a number of Windows options for thread locking to guard against preemption, synchronization and run-order issues; the Microsoft paper on [Locks, Deadlocks and Synchronization](http://www.microsoft.com/whdc/driver/kernel/locks.msp#Locks,DeadlocksandSynchronization) discusses many of them and provides a solid overview. Another paper, [Multiprocessor Considerations for Kernel-Mode Drivers](http://www.microsoft.com/whdc/driver/kernel/MP_issues.msp#MultiprocessorConsiderationsforKernel-ModeDrivers) goes into more detail.

PCI Bus Device Development Notes

Many devices in a uniprocessor or multiprocessor are connected via a Peripheral Component Interconnect (PCI) I/O bus. In practice, in an AMD64 system, the PCI bus is connected to the high-bandwidth HyperTransport bus via a PCI-HyperTransport adapter chip. Because device driver code running on multiple processor cores can be accessing a PCI device at the same time, and in different contexts, PCI drivers should be developed and tested to run safely on a multiprocessor system.

When developing a PCI device driver, there are several methods of accessing the PCI bus to determine which devices are currently connected. Some of these methods are legacies from older versions of Windows, such as the old Windows 4.0 code base, and others may be limited to specific Windows builds, such as 32-bit Windows only. These methods may work in some or most cases on current versions of Windows XP or Windows Server 2003 in multiprocessor environments – but also may not perform reliably under all circumstances, and may not work properly with the multiprocessor locking methods described above. They may not also work with future versions of Windows, such as Windows Vista or Windows Server “Longhorn.”

One specific challenge is when system software programs obtain configuration and location information about the PCI bus and its attached peripherals – such as the BusNumber, DeviceNumber and FunctionNumber of a device – by scanning the bus and using the HalGetBusData and HalGetBusDataByOffset API calls, which returns PCI port information directly.

However, beginning with Windows 2000, the Hardware Abstraction Layer APIs, including those mentioned above, are depreciated, and should not be used. Instead, developers should obtain all PCI bus and device information by asking the operating system, not by interrogating the bus or device directly.

According to Microsoft, the driver gets its resources from the Plug and Play (PnP) manager in its IRP_MN_START_DEVICE request. Typically, a well-written driver should not require any of this information to function. If the driver does require this information, the code sample to follow shows how to get the resources. The driver should be part of the device's driver stack because it requires the underlying physical device objects of the device to send the PnP request.

Normally, requests for PnP I/O Request Packets are sent at the PASSIVE_LEVEL. However, Microsoft explains that to access the configuration space at the DISPATCH_LEVEL, send an IRP_MN_QUERY_INTERFACE at PASSIVE_LEVEL to get the direct-call interface structure (BUS_INTERFACE_STANDARD) from the PCI bus driver. Store this in a nonpaged pool memory (typically in DeviceExtension). Then, call the SetBusData and GetBusData to access the configuration space at DISPATCH_LEVEL. Note that the PCI bus driver takes a reference count on the

interface before it returns, so the code must dereference the interface when it is no longer needed.

Note that on all systems – including servers, desktops and mobile devices – PCI bus numbers should be considered to be transient, and can change at any time. That is another reason why developers should not rely on the bus numbers, or use the bus number to access the PCI ports directly; if a number changes, this could lead to a system failure.

Microsoft has more information on this issue, including code samples, at [How To Get The Configuration And Location Information Of A PCI Device](http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q253232) (<http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q253232>).

BIOS Notes for Multiprocessing

BIOS designers, developers and testers, such as device driver creators, must be aware of the issues regarding multiprocessor and multi-core systems. Those issues include all of the ones listed earlier, such as for simultaneous thread execution, as well as well-known threaded development challenges such as managing deadlocks and race conditions. There are specific issues, however, that BIOS developers must address when creating new BIOS software, or adapting existing device BIOS code to work with a dual-core or multicore chip.

If the BIOS might even need to access a potentially shared resource, it is essential to ensure that only one core or processor can ever access the resource, at any one time, from start to finish, whenever possible. If it is necessary that multiple cores or processors need to access the same resource, the BIOS coder must ensure that all of the accesses are done serially, to prevent unwanted side effects. For example, if multiple threads need to access CMOS space – and there's a possibility that those threads might run in different cores or processors – simultaneous access might let one application processor (AP) overwrite the index value that the bootstrap processor (BSP) has written earlier. Subsequent accesses by the AP or the BSP would then retrieve the wrong data.

It is also important that the local Advanced Programmable Interrupt Controller (APIC) for each core has an enabled entry in the Advanced Configuration and Power Interface (ACPI) MADT table. (The MADT, or the Multiple APIC Description Table, describes all of the resources which can initiate or respond to interrupt.) Each processor and core must be declared using the Processor statement in the MADT. Similarly, each core's APIC must have a Processor entry in the system's Multiprocessor Configuration Table. Refer to the [Advanced Configuration and Power Interface Specification](http://www.acpi.info/spec.htm) (<http://www.acpi.info/spec.htm>) and the [Intel MultiProcessor 1.4 Specification](http://developer.intel.com/design/pentium/datashts/24201606.pdf) (<http://developer.intel.com/design/pentium/datashts/24201606.pdf>).

BIOS Notes for Systems Management Interrupt

Developers face specific issues when coding logic for the Systems Management Interrupt (SMI). If an SMI handler must access shared resources through an Index/Data pair, a best practice is to ensure that the value in the Index register is saved before writing the new Index value, and then is restored after use of the Index/Data register pair is complete. This will guarantee that if the operating system is interrupted by an SMI while accessing a shared resource, Windows will retrieve the correct value from the data register when the SMI operation is complete. Otherwise, the second SMI might overwrite the index, and the original SMI handler would never be aware of it.

Note that in a dual-core or multicore system, when a directed SMI occurs, only one core within that processor will enter the SMI handler. Do not attempt to assume which core has initiated the SMI. In order to prevent any resource-contention issues while the interrupt is being handled, the SMI handler should put all of that processor's cores into SMI handling mode for the duration of the interrupt handling process. All of the cores need to exit the SMI handler at the same time by synchronizing their use of the Return from Systems Manager (RSM) instruction.

On a similar note, in a dual-core or multi-core system, when the processor is operating in Systems Management Mode (SMM), which is a mode typically invoked whenever an SMI is asserted, such as when there are changes to the power-supply status. When in SMM, each core has access to its own model specific registers (MSRs), which describe the low-level hardware status and capabilities of the system. These registers, known as SMM_BASE, SMM_ADDR and SMM_MASK, must be programmed separately for each core, so that each core has its own private save spaces. Cache-specific settings in the SMM_MASK registers must be set to be identical.

For more information about systems management mode and SMIs, see the [AMD BIOS and Kernel Developer's Guide for Athlon 64 and Opteron Processors](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26094.PDF) (http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26094.PDF).

Note also that the BIOS may crash the system if it tries to directly access some I/O resources directly through the Advanced Configuration and Power Interface system, instead of working through operating system calls. To prevent that, Microsoft has blocked access to some resources from the ACPI Machine Language (AML) interpreter. Microsoft lists those blocked system resources, and provides recommended means for accessing them through Windows XP, Windows Server 2003 and future versions of Windows, at [I/O Ports Blocked from BIOS AML](http://www.microsoft.com/whdc/system/pnppwr/powergmt/BIOSAML.mspx) (<http://www.microsoft.com/whdc/system/pnppwr/powergmt/BIOSAML.mspx>).

BIOS Notes for Dual-Core Specific Resources

For application developers or systems administrators, there's no practical difference between a system with one dual-core processor and a system with two single-core processors. Although there can be some performance characteristics that vary between

the two configurations, for the most part those are masked by the operating system, and developers can and should assume that the cores – no matter where they are located – operate entirely independently.

However, for BIOS developers, it is important to note that there are architectural differences between a single-core and dual-core processor. All cores have their own independent L1 and L2 caches, general register sets, local Advanced Programmable Interrupt Controller (APIC), microcode, model specific registers (MSRs) and Machine Check Architecture (MCA) registers,. However, there are a few exceptions for dual-core or multicore processors, where the cores within a dual-core or multicore processor share the following resources:

- FID/VID Control (MSR C001_0041h)
- FID/VID Status (MSR C001_0042h)
- MC4_CTL (MSR 0410h)
- MC4_STATUS (MSR 0411h)
- MC4_ADDR (MSR 0412h)
- MC4_MISC (MSR 0413h)
- MC4_CTL_MASK (MSR C001_0048h)

Also note that each core within a dual-core or multicore processor shares the same Northbridge. This means that all of the PCI registers (Function 0 through Function 3) are shared. This is why the two cores share the MC4 MSRs. The second core in a NUMA (non-uniform memory access) node must be launched before initiating a FID/VID change. The second core is launched by setting the CPU1En bit. See section 10.5.4, BIOS-Initiated P-State Transitions, in the [BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/32559.pdf) (http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/32559.pdf) for details.

References

Microsoft Resources

[Driver Fundamentals: Overview](http://www.microsoft.com/whdc/driver/default.mspix) (<http://www.microsoft.com/whdc/driver/default.mspix>)

[Kernel Mode Fundamentals](http://www.microsoft.com/whdc/driver/kernel/default.mspix)

(<http://www.microsoft.com/whdc/driver/kernel/default.mspix>)

[Multiprocessor Considerations for Kernel-Mode Drivers](http://www.microsoft.com/whdc/driver/kernel/MP_issues.mspix)

(http://www.microsoft.com/whdc/driver/kernel/MP_issues.mspix)

[Scheduling, Thread Context, and IRQL](http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/IRQL_thread.doc)

(http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/IRQL_thread.doc)

[Locks, Deadlocks and Synchronization](http://www.microsoft.com/whdc/driver/kernel/locks.mspix)

(<http://www.microsoft.com/whdc/driver/kernel/locks.mspix>)

[To Create a PCI Device Driver for Windows NT](http://support.microsoft.com/kb/q152044/)

(<http://support.microsoft.com/kb/q152044/>)

[How To Get The Configuration And Location Information Of A PCI Device](http://support.microsoft.com/kb/q152044/)

Common Driver Reliability Issues

(<http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q253232>)

AMD Resources

AMD BIOS and Kernel Developer's Guide for Athlon 64 and Opteron Processors

(http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26094.PDF)

BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors

(http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/32559.pdf)

Other Resources

Advanced Configuration and Power Interface (ACPI) Specification, Revision 3.0

(<http://www.acpi.info/spec30.htm>)

Intel MultiProcessor 1.4 Specification

(<http://developer.intel.com/design/pentium/datashts/24201606.pdf>).

Acknowledgements

Camden Associates would like to recognize the generous contributions of two AMD engineers to this White Paper: **Leo Duran** for device driver-related content and **Billy Hagan** for BIOS-related content.