Memory Management: What Every Driver Writer Needs to Know

February 28, 2005

Abstract

This paper provides information about allocating and using memory in kernel-mode drivers for the Microsoft® Windows® family of operating systems. It describes the types of memory that are available for driver use, the appropriate techniques for allocating and using each type, and the best ways to test for memory-related problems.

This information applies for the following operating systems:

Microsoft Windows Vista™
Microsoft Windows Server™ 2003
Microsoft Windows XP
Microsoft Windows 2000

The current version of this paper is maintained on the Web at:
http://www.microsoft.com/whdc/driver/kernel/mem-mgmt.mspx

References and resources discussed here are listed at the end of this paper.

Contents

Disclaimer

1 Introduction

Kernel-mode drivers allocate memory for various purposes, such as storing internal data or using as I/O buffers. To help driver writers use memory correctly, this paper explains the following:

- Fundamental information about physical and virtual memory and address spaces.
- Types of memory that are available to drivers and when to use each type.
- Techniques for allocating memory to satisfy a driver's diverse requirements.
- Techniques for accessing memory allocated by other components for I/O operations.
- Techniques for sharing memory with other kernel-mode and user-mode components.
- Techniques for testing and troubleshooting memory allocation and usage problems.

Note:
This paper does not cover memory allocation for direct memory access (DMA). You can find detailed information about all aspects of DMA in "DMA Support in Windows Drivers," which is listed in the Resources section at the end of this paper.

2 Virtual and Physical Memory

The amount of virtual and physical memory that is supported on any computer that runs the Microsoft® Windows® operating system is determined by the hardware configuration and the edition of Windows in use. On 32-bit hardware, the virtual address space is 4 GB and the maximum amount of physical memory ranges from 4 to128 GB. On 64-bit hardware, the virtual address space is 16 terabytes and the maximum amount of memory ranges from 64 GB to 1 terabyte.

Table 1 lists the amount of virtual memory and the maximum amount of physical memory that each edition of Windows supports.

Table 1. Virtual and Physical Memory Support in Windows

Operating system version	Edition	Virtual memory	Maximum physical memory
Microsoft Windows Server™ 2003 SP 1	Standard	4 GB	4 GB
	Web	4 GB	2 GB
	Enterprise	4 GB	64 GB, if hardware supports Physical Address Extension (PAE)
	Enterprise (64-bit)	16 terabytes	1 terabyte
	Datacenter	4 GB	128 GB, if hardware supports PAE
	Datacenter (64-bit)	16 terabytes	1 terabyte

Operating system version	Edition	Virtual memory	Maximum physical memory
Windows Server 2003	Standard	4 GB	4 GB
	Web	4 GB	2 GB
	Enterprise	4 GB	32 GB, if hardware supports PAE
	Enterprise (64-bit)	16 terabytes	64 GB
	Datacenter	4 GB	128 GB, if hardware supports PAE
	Datacenter (64-bit)	16 terabytes	512 GB
Windows XP	Home	4 GB	4 GB
	Professional	4 GB	4 GB
	64-bit Edition Version 2003	16 terabytes	128 GB
Windows 2000	Professional	4 GB	4 GB
	Server	4 GB	4 GB
	Advanced Server	4 GB	8 GB
	Datacenter Server	4 GB	32 GB, if hardware supports PAE

3 Virtual Address Space

A virtual address identifies a location in virtual memory. The virtual address space is divided into two ranges: user space and system space.

User space is the portion of the virtual address space into which Windows maps user-mode processes, per-process data, and user-mode dynamic link libraries (DLLs). Each process has its own context, that is, the code and data that the process uses. While the process is running, a portion of the context, called the working set, is resident in memory.

System space, also called *kernel space*, is the portion of the address space in which the operating system and kernel-mode drivers are resident. It is accessible only to kernel-mode code.

The distinction between user space and system space is important in maintaining system security. A user-mode thread can access data only in the context of its own process. It cannot access data within other process contexts or within the system address space. This restriction prevents user-mode threads from reading and writing data that belongs to other processes or to the operating system and drivers, which could result in security vulnerabilities and possibly even a system crash.

Kernel-mode drivers are trusted by the operating system and consequently can access both user space and system space. When a driver routine is called in the context of a user thread, all of the thread's data remains in the user-mode address space. The driver can access the user-space data for the thread—if it takes the necessary security precautions—in addition to accessing system-space data. The user-mode thread, however, does not have access to the system-space data of the driver.

The sizes of the system and user address spaces depend on whether the hardware is 32-bit or 64-bit and on the options applied to Boot.ini.

3.1 Virtual Address Space on 32-bit Hardware

On 32-bit machines, the lower 2 GB of the virtual address space is user space and the upper 2 GB is reserved as system space by default. If user-mode applications require additional virtual address space, an administrator can reserve 3 GB for user space, thus leaving only 1 GB of system space, by applying the /3GB switch to Boot.ini and then restarting the machine. This switch is also useful during testing to see how a driver performs with limited system address space. The switch is supported on Windows Server 2003 (all editions), Windows XP (all versions), Windows 2000 Advanced Server, and Windows 2000 Datacenter Server.

Figure 1 shows how the virtual address space is organized on 32-bit systems. (The figure is not drawn to scale.)

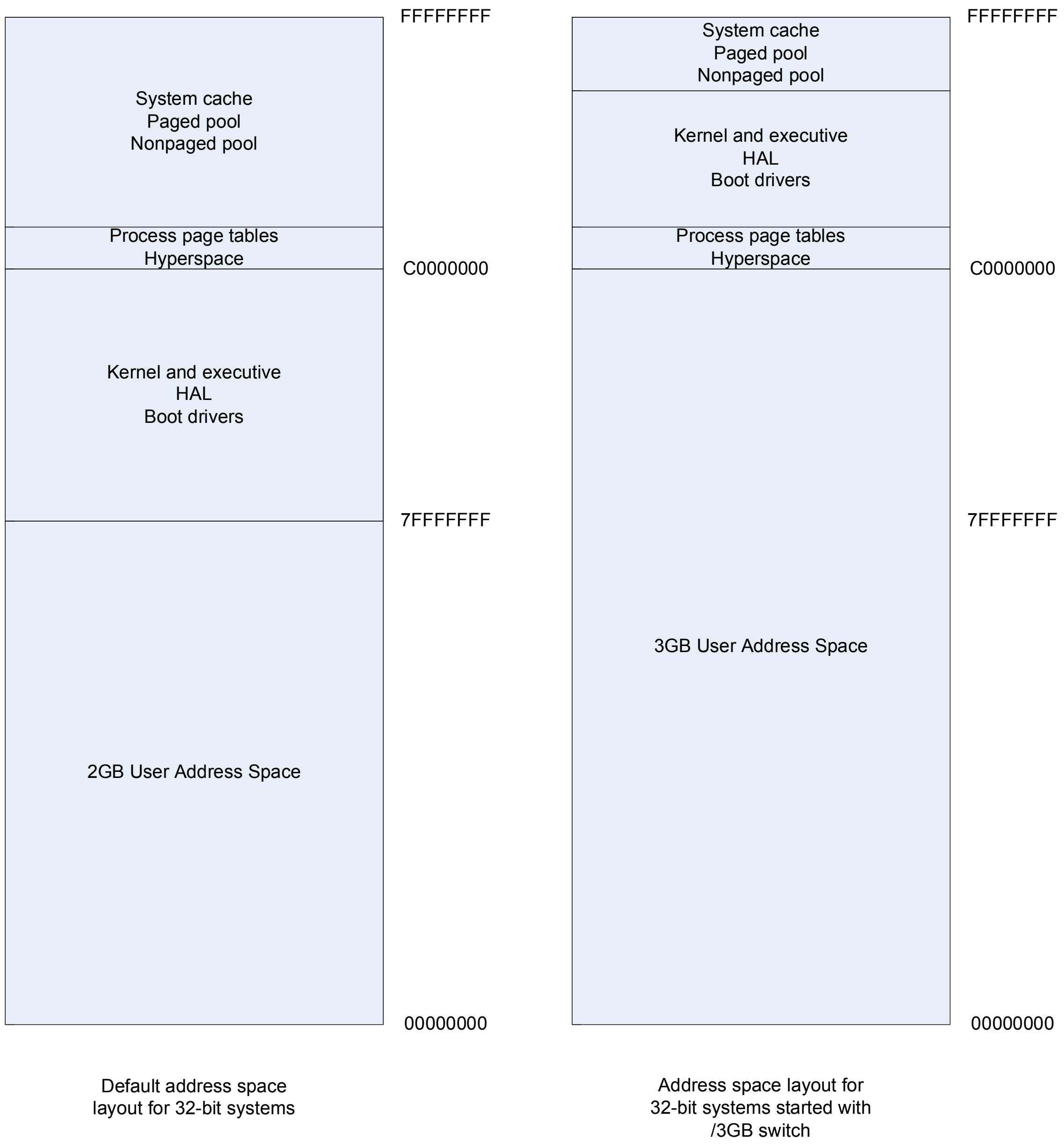

Figure 1. Layout of virtual address space on 32-bit systems

As the figure shows, the user address space occupies the lower range of the virtual address space and the system address space occupies the upper range. By default, both user and system space are 2 GB. When Windows is started with the /3GB switch, the size of the user space increases to 3 GB. The remaining 1 GB of virtual address space contains the system address space. The process page tables and hyperspace remain at the same addresses, but other components do not. When the system address space is limited to 1 GB, the maximum sizes of the paged and nonpaged pools are reduced and less space is available for system page table entries (PTEs) and the file system cache as well.

The file system cache, which is implemented in software, is not the same as the processor cache, which is implemented in hardware. Keep in mind, however, that this limit applies only to the amount of the cache that is visible at any given time. The system uses all the physical memory in the machine for caching.

The exact sizes of the paged and nonpaged pools, PTEs, and file system cache vary widely from release to release.

On the Intel Pentium Pro and later processors, which support PAE, Windows can support up to 128 GB of physical memory. PAE provides only additional physical memory and does not increase the size of the virtual address space. When PAE is enabled, the virtual address space remains unchanged at 4 GB and the layout of the 2-GB system space is largely unchanged.

Operating system support for PAE is provided by a special version of the Windows kernel, which is named Ntkrnlpa.exe (single-processor version) or Ntkrpamp.exe (multi-processor version). In this version of the kernel, PTEs and page directory entries (which are involved in mapping virtual memory to physical memory) are 64 bits wide. If the appropriate device and driver support is present, the PAE kernels support direct I/O for the entire physical address space (including any physical memory above 4 GB).

3.2 Virtual Address Space on 64-bit Hardware

On 64-bit machines, the virtual address space is 16 terabytes, with 8 terabytes of user space and another 8 terabytes of system space. The layout is similar to that for 32-bit Windows, except that the sizes are proportionately larger. As with 32-bit hardware, the sizes vary from release to release.

Currently, Windows runs on two 64-bit architectures: Intel Itanium and x64, which includes the AMD64 and the Intel Extended Memory 64 Technology. On x86 and x64 hardware, the page size is 4 KB. On Itanium hardware, the page size is 8 KB. Pointers on both Itanium and x64 architectures are 64 bits long.

3.3 Types, Constants, and Macros Used in Addressing

Because of the differences between the x86, x64, and Itanium architectures, drivers should not use hard-coded values or make assumptions about the page size, pointer size, or address ranges of the hardware. For example, on 64-bit hardware, a virtual address in which the highest-order bit is set is not necessarily a system-space address. (The same is true for 32-bit hardware when Windows is started with the /3GB switch.)

Instead, driver code should always use the appropriate constants, types, and macros, which are defined in Wdm.h and Ntddk.h. These include the following:

Constant, type, or macro	**Usage**
PVOID (or other data-specific pointer type)	Virtual addresses
PHYSICAL_ADDRESS	Physical addresses
PAGE_SIZE	Number of bytes in a page on the current hardware (used in memory allocation)

Constant, type, or macro	Usage
BYTES_TO_PAGES macro	Number of pages required to hold the input number of bytes on the current hardware
ULONG_PTR	Variables that might hold either an address or data of a scalar type
PVOID64	Pointers on Itanium and x64 hardware only; does not return valid information on x86 hardware

Code that uses these macros and types can depend on Windows to do the right thing, regardless of the underlying hardware platform.

Most drivers run on PAE systems without modification. For PAE compatibility, drivers that perform DMA must use the system's DMA routines to ensure that address size issues are handled correctly. In addition, drivers should not use ULONG, PVOID, or PVOID64 for physical addresses. On PAE systems, the ULONG and PVOID types are only 32 bits long and therefore cannot store physical addresses above 4 GB. The PVOID64 type does not return valid information on the x86 architecture. Instead of using ULONG, PVOID, or PVOID64, drivers should use ULONG_PTR. Additional limitations apply for certain device types. For further details, see the Windows DDK, which is listed in the Resources section at the end of this paper.

3.4 Mapping Virtual Memory to Physical Memory

This section provides a brief overview of how the Windows memory manager maps pages in virtual memory to pages in physical memory. A general understanding of this mapping is helpful in writing a driver, but you do not need to understand the details. For a more detailed explanation, see *Inside Windows 2000*, which is listed in the Resources section at the end of this paper.

Every page in virtual memory is listed in a page table, which in turn identifies the corresponding physical page. The system and CPU use information from the virtual address to find the correct entry in the page table for a specific page.

Figure 2 shows how the memory manager maps virtual memory to physical memory. In the figure, a virtual address points to a specific location on a virtual page. The virtual address contains a byte offset and several index values that the system and the CPU use to locate the page table entry that maps the virtual page into physical memory. After the memory manager finds the page table entry, it uses the offset to find a byte in physical memory, which is identified by a physical address.

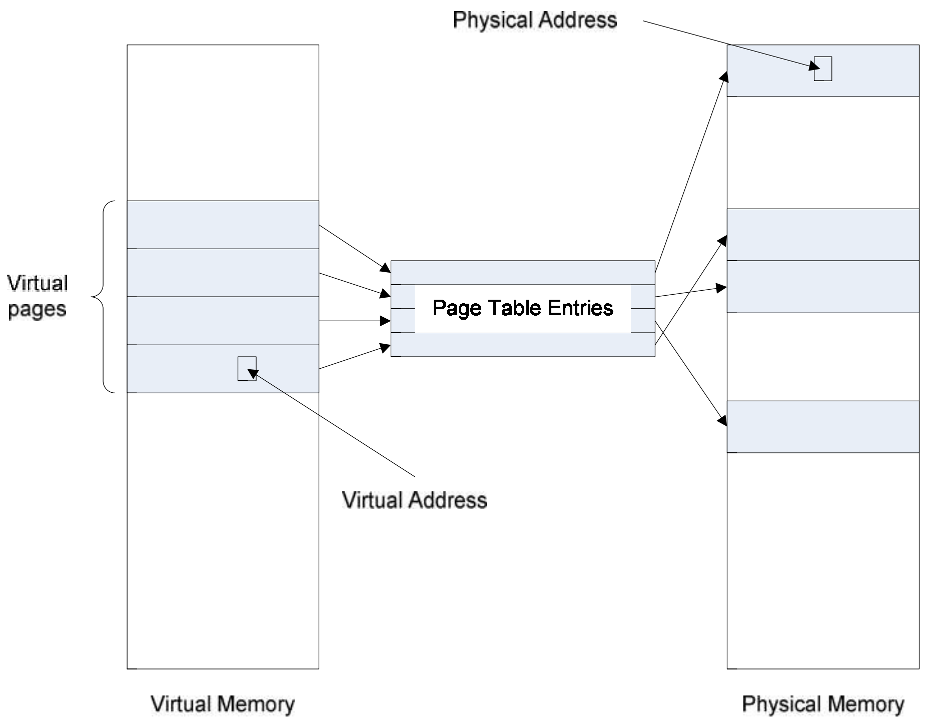

Figure 2. Mapping Virtual to Physical Memory

The length and layout of a virtual address depend on the hardware. Virtual addresses on x86 hardware are 32 bits long and contain two page-table indexes: one is an index into a page directory, which identifies the page table that maps the virtual address, and the other index identifies the page table itself. In PAE mode, the address contains a third index that is 2 bits wide.

Virtual addresses are 48 bits long on x64 hardware and 43 bits long on Itanium hardware, and the number of indexes is also different. However, on all 64-bit hardware, Windows stores and manipulates virtual addresses as 64-bit values. Drivers must not rely on the 43-bit and 48-bit internal address lengths (for example, by attempting to encode additional information in pointers) because Microsoft might change these values at any time.

4 Physical Address Space

The physical address space encompasses the set of addresses that physical memory can occupy. The maximum size of the physical address space is determined by the number of bits of physical address that the CPU and chipset can decode. This size also establishes the theoretical maximum amount of physical memory (RAM) on the system.

As hardware has evolved, the number of address bits has increased, leading to larger physical address spaces and potentially greater amounts of RAM. Current x86 CPUs use 32, 36, or 40 bits for physical addresses in the modes that Windows supports, although the chipsets that are attached to some 40-bit processors limit the sizes to fewer bits. Current releases of 32-bit Windows support a maximum of 37 bits of physical address for use as general-purpose RAM (more may be used for I/O space RAM), for a maximum physical address space of 128 GB. (These values may increase in the future.) Windows also continues to support older processors that decode only 32 bits of physical address (and thus can address a maximum of 4 GB).

A processor that uses full 64-bit virtual addresses can theoretically address 16 exabytes. (An exabyte is a billion gigabytes, and a terabyte is a thousand gigabytes.) Current releases of 64-bit Windows support from 40 to 50 address bits and 128 GB to 1 terabyte of RAM, depending on the specific edition of the operating system.

The number of bits that are supported in physical addresses is important to driver writers because it establishes the range of destination addresses for DMA. Currently, drivers must be able to handle I/O transfers that use 37-bit physical addresses, and this number could increase in the future. For drivers that use the Windows DMA routines instead of the obsolete hardware abstraction layer (HAL) routines, increased address lengths are not a problem even if the device hardware supports only 32 address bits because the DMA routines perform any double-buffering that might be required to handle the addressing differences.

Some older drivers, however, do not use the Windows DMA routines. If such a driver assumes that its device will never be required to perform I/O to an address greater than 4 GB, it is likely to fail on some newer x86 platforms on which physical addresses can extend up to 128 GB. These drivers should be revised to use the Windows DMA routines.

The physical address space is used to address more than just RAM. It is also used to address all of the memory and some of the registers presented by devices. Consequently, if a machine is configured with the maximum amount of physical memory, some of that memory will be unusable because some of the physical address space is mapped for other uses.

Device memory and registers are mapped into the physical address space at address ranges that the chipset controls. Memory locations are read and written through LOAD and STORE instructions; in the x86 assembly language, load and store are implemented through the MOV instruction.

Although some device registers are mapped into the physical address space, others are I/O space instead, depending on the design of the device hardware and the chipset in the individual machine. I/O space is a holdover from the early days of microprocessors when memory was a limited resource and few devices had their own addressable memory. Creating a separate, limited address space through which to address device registers saved important main memory space for the operating system and applications.

On the hardware platforms that Windows currently supports, I/O space is implemented as a separate 64-KB physical address space. Locations in I/O space are read and written through IN and OUT instructions.

The addresses that are used to identify locations in a physical address space are often called *physical addresses*. However, this term can sometimes be confusing because not all physical addresses are alike—every physical address identifies a location relative to the bus on which it is placed. The following terms more precisely describe the physical addresses:

- Processor-relative physical addresses
- Device-bus relative physical addresses

Understanding which type of physical address each component uses can help you ensure that your driver uses the correct type of address for each context. Figure 3 shows the types of addresses used by various system components. In the figure:

- The light gray area at the top shows the virtual address space.
- The unshaded area shows the operating system and drivers, which can use either virtual addresses or physical addresses, depending on the situation.
- The darker gray areas at the bottom show components that use physical addresses. The components in the lighter area use processor-relative physical addresses, and those in the darker area use device bus-relative physical addresses.

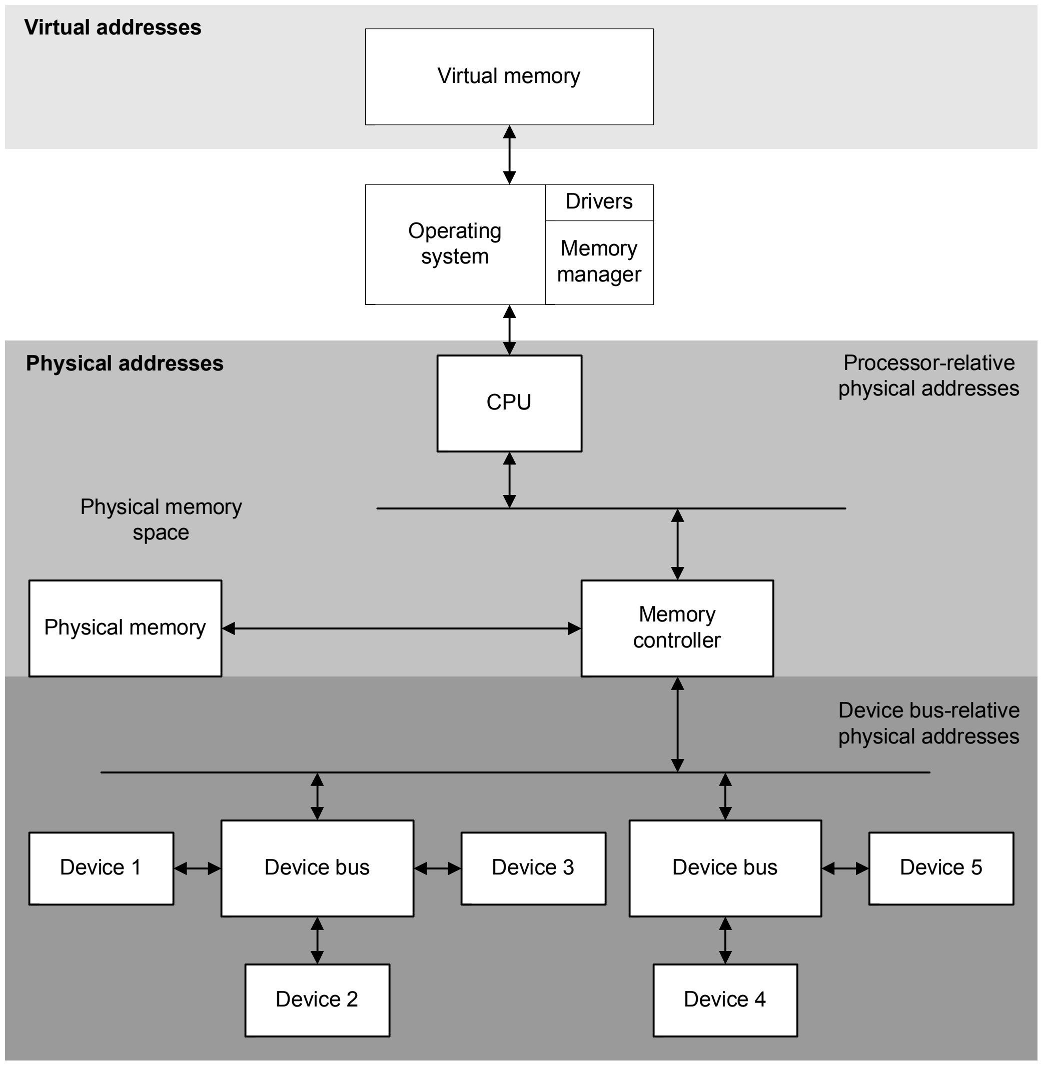

Figure 3. Physical addresses

The following sections describe processor-relative and device-bus-relative physical address in more detail.

4.1 Processor-Relative Physical Addresses

A processor-relative physical address is an address that the CPU places on the CPU bus. This address identifies a location in the physical memory space, which contains all addresses that can be the target of a MOV instruction. When the memory manager and the CPU translate a virtual address to a physical address, the result is a processor-relative physical address.

After the CPU issues an instruction with a processor-relative physical address, the memory controller (also called the North Bridge) decodes the address and directs it appropriately. If the address identifies a location in physical memory, the memory controller, in theory, translates it to the specific location relative to the memory bus at which the memory is located. On standard PC platforms, however, the processor-relative physical address is almost always the same as the address in physical memory space, so the translation is simply a one-to-one identity mapping.

If the address identifies a location that is mapped to a device, the memory controller translates it to a device-bus-relative physical address. It can then place the translated address on the appropriate bus.

4.2 Device-Bus-Relative Physical Addresses

The *device-bus relative address space* defines a set of addresses that identify locations on a specific bus.

Note
The Windows DDK uses the term “logical address space” to describe the device-bus relative address spaces and the term “logical address” to describe an address that is used to identify a location in them. Like “physical address,” these terms are sometimes confusing because they are often used in other contexts and with other meanings.

In theory, each device bus can have its own address space. In practice, however, most PC platforms implement only a single root device bus. The few platforms that implement more than one root device bus simply segment the available address space without translating it, so that all the root buses appear to share a single address space.

In addition, the PCI Specification defines the relationship between the address spaces of parent and child PCI buses and prohibits translations. Any translations that are performed in such machines must take place at the level of the root device bus.

When a device receives a device-bus-relative address, it decodes the address to determine whether the bus cycle is intended for it and, if so, for which of its registers.

Devices issue device-bus relative physical addresses when they perform DMA. These addresses are later translated to processor-relative physical addresses (typically by the memory controller). The map registers that are central to the Windows DMA model are essentially an abstraction of this translation step. The use of map registers ensures that any device can perform DMA to any location in physical memory, regardless of any addressing limitations in the device. Depending on the individual machine, the map registers might be implemented in hardware or in software.

5 Memory Descriptor Lists

A memory descriptor list (MDL) describes a list of pages in physical memory. Internally, the Windows kernel uses MDLs in numerous ways. For example, when the I/O manager sets up a direct I/O request to send to a driver, it creates an MDL to describe the I/O buffer. The driver receives a pointer to the MDL in the request at **Irp->MdlAddress**.

When the driver receives the request, the pages described by the MDL have already been locked into memory by the creator of the MDL. To use virtual addresses to access the buffer described by the MDL, the driver calls **MmGetSystemAddressForMdlSafe** to map the buffer into system space. This function returns a valid system-space virtual address that the driver can use to access the buffer.

Drivers create MDLs in the following situations:

- The driver receives an I/O request for neither buffered nor direct (METHOD_NEITHER) I/O.
- The driver initiates an I/O request to send to another driver.
- The driver splits a large I/O buffer into two or more smaller buffers.
- The driver allocates physical memory from a specific physical address range.

When a driver receives a request for METHOD_NEITHER I/O, a pointer to a buffer is part of the request. If the originator of the I/O request is another kernel-mode component, the buffer can be in system space. More often, however, the originator of the request is a user-mode process, so the buffer is in user space. To read and write to this user-space buffer, the driver must create an MDL that describes these pages and must ensure that they remain accessible until the driver has completed the I/O request. Creating the MDL locks the pages into memory so that any subsequent access is guaranteed to be protected. If the driver must use virtual addresses to access the pages described by the MDL, it must map those pages into the system address space using **MmGetSystemAddressForMdlSafe**. This function returns a valid system-space virtual address with which the driver can safely access the buffer.

When a driver initiates an I/O request, it allocates an MDL to describe the buffer in the I/O request. For example, a driver might send a device I/O control request (IRP_MJ_INTERNAL_DEVICE_CONTROL) to a lower driver in its stack to request some kind of intervention with the hardware. The driver would call **IoAllocateMdl** to create an MDL and later use a different system function to fill in the MDL, depending on the type of request. If the request specifies direct I/O, the lower driver would receive a pointer to the MDL in the request at **Irp->MdlAddress**.

To split a large I/O buffer into smaller buffers, a driver might also use an MDL. Typically, splitting a buffer in this way is required only during direct I/O operations, if the supplied buffer is too large for the device or if the driver or device can operate more efficiently with several smaller buffers. To split a buffer, the driver calls **IoAllocateMdl** to create the new MDL, and then calls **IoBuildPartialMdl** to map a subset of the range described by the original MDL into the new one.

A driver that allocates physical memory from a specific range of addresses must also create an MDL to describe the allocated memory. In this case, the driver calls **MmAllocatePagesForMdl** (which allocates the memory), creates the MDL, and locks the pages. If the driver requires access to the pages through virtual addresses, the driver also calls **MmGetSystemAddressForMdlSafe** to map the pages into virtual memory and return a virtual address for the driver's use.

Table 2 summarizes Windows routines that create and manipulate MDLs.

Table 2. Summary of MDL Routines

Name	Description
MmAllocateMappingAddress	Reserves a range of virtual addresses for mapping, but does not actually map them to physical memory. To map the virtual addresses in the reserved range to physical addresses, the driver later calls **IoAllocateMdl**, **MmProbeAndLockPages**, and **MmMapLockedPagesWithReservedMapping**. This technique is less efficient than calling **MmGetSystemAddressForMdlSafe** or **MmMapLockedPages*Xxx***, but is useful because it allows driver execution to continue in situations when calling these latter functions fails.
IoAllocateMdl	Allocates an MDL for a buffer, given the starting virtual address of the buffer and its length. This routine does not associate the physical pages in the buffer with the allocated MDL. To do so, the driver must call **MmBuildMdlForNonPagedPool**, **IoBuildPartialMdl**, or **MmProbeAndLockPages**.
MmProbeAndLockPages	Confirms that the pages described by an MDL permit the requested type of access and, if so, locks those pages into memory.
MmBuildMdlForNonPagedPool	Fills in an MDL to describe a buffer that is allocated in system-space memory from the nonpaged pool. The MDL must have been allocated by **IoAllocateMdl**. A driver can also use this function on I/O space, typically by mapping the space with **MmMapIoSpace** and then passing the returned virtual address to **MmBuildMdlForNonPagedPool**.
MmAllocatePagesForMdl	Allocates nonpaged, noncontiguous pages from physical memory, specifically for an MDL. A driver calls this function to allocate physical memory from a particular address range. Do not use this function to allocate memory for DMA; use the system's DMA routines instead. This function might return fewer pages than the caller requested. Therefore, the driver must call **MmGetMdlByteCount** to verify the number of bytes allocated.
MmGetSystemAddressForMdlSafe	Maps the physical pages described by an MDL into system space and returns a virtual address for the MDL. The returned virtual address can be used at any IRQL and in any process context.
IoBuildPartialMdl	Maps part of a buffer described by an existing MDL into a new MDL that the driver previously allocated by calling **IoAllocateMdl**.
MmAdvanceMdl	Increases the starting address of an MDL. A driver calls this function to update the MDL to describe a partial buffer when I/O operations are not (or cannot be) completed, either because not enough memory is available or because the device can write only part of the buffer at a time.
MmGetMdlVirtualAddress	Returns the starting virtual address of an MDL.

Name	Description
IoFreeMdl	Frees an MDL that was previously allocated by **IoAllocateMdl**.
MmFreePagesFromMdl	Frees the pages described by an MDL that was previously created by **MmAllocatePagesForMdl**.

6 Pool Memory

Windows provides pools of paged and nonpaged memory that drivers and other components can allocate. The Executive component of the operating system manages the memory in the pools and exposes the **ExAllocatePool*Xxx*** functions for use by drivers. Pool memory is a subset of available memory and is not necessarily contiguous. The size of each pool is limited and depends on the amount of physical memory that is available and varies greatly for different Windows releases.

The paged pool is exactly what its name implies: a region of virtual memory that is subject to paging. The size of the paged pool is limited and depends on both the amount of available physical memory on each individual machine and the specific operating system release. For example, the maximum size of the paged pool is about 491 MB on 32-bit hardware running Windows XP and about 650 MB on Windows Server 2003 SP1.

The nonpaged pool is a region of system virtual memory that is not subject to paging. Drivers use the nonpaged pool for many of their storage requirements because it can be accessed at any IRQL. Like the paged pool, the nonpaged pool is limited in size. On a 32-bit x86 system that is started without the /3GB switch, the nonpaged pool is limited to 256 MB; with the /3GB switch, the limit is 128 MB. On 64-bit systems, the nonpaged pool currently has a limit of 128 GB.

The pool sizes and maximums may vary greatly for different Windows releases.

6.1 IRQL Considerations

When you design your driver, keep in mind that the system cannot service a page fault at IRQL DISPATCH_LEVEL or higher. Therefore, drivers must use nonpaged pool for any data that can be accessed at DISPATCH_LEVEL or higher. You cannot move a buffer that was allocated from the paged pool into the nonpaged pool, but you can lock a paged buffer into memory so that it is temporarily nonpaged.

Locks must never be allocated in the paged pool because the system accesses them at DISPATCH_LEVEL or higher, even if the locks are used to synchronize code that runs below DISPATCH_LEVEL.

Storage for the following items can generally be allocated from the paged pool, depending on how the driver uses them:

- Information about device resources, relations, capabilities, interfaces, and other details that are handled in IRP_MN_PNP_QUERY_* requests. The Plug and Play manager sends all queries at PASSIVE_LEVEL, so unless the driver must reference this information at a higher IRQL, it can safely store the data in paged memory.
- The registry path passed to **DriverEntry**. Some drivers save this path for use during WMI initialization, which occurs at PASSIVE_LEVEL.

While running at DISPATCH_LEVEL or below, a driver can allocate memory from the nonpaged pool. A driver can allocate paged pool only while it is running at PASSIVE_LEVEL or APC_LEVEL because APC_LEVEL synchronization is used within the pool manager code for pageable requests. Furthermore, if the paged pool is nonresident, accessing it at DISPATCH_LEVEL—even to allocate it—would cause a fatal bug check.

For a complete list of standard driver routines and the IRQL at which each is called, see “Scheduling, Thread Context, and IRQL,” which is listed in the Resources section at the end of this paper. In addition, the Windows DDK lists the IRQL at which system and driver routines can be called.

6.2 Lookaside Lists

Lookaside lists are fixed-size, reusable buffers that are designed for structures that a driver might need to allocate dynamically and frequently. The driver defines the size, layout, and contents of the entries in the list to suit its requirements, and the system maintains list status and adjusts the number of available entries according to demand.

When a driver initializes a lookaside list, Windows creates the list and holds the buffers in reserve for future use by the driver. The number of buffers that are in the list at any given time depends on the amount of available memory and the size of the buffers. Lookaside lists are useful whenever a driver needs fixed-size buffers and are especially appropriate for commonly used and reused structures, such as I/O request packets (IRPs). The I/O manager allocates its own IRPs from a lookaside list.

A lookaside list can be allocated from either the paged or the nonpaged pool, according to the driver’s requirements. After the list has been initialized, all buffers from the list come from the same pool.

6.3 Caching

Drivers can allocate cached or noncached memory. Caching improves performance, especially for access to frequently used data. As a general rule, drivers should allocate cached memory. The x86, x64, and Itanium architectures all support cache-coherent DMA, so drivers can safely use cached memory for DMA buffers.

Drivers rarely require noncached memory. A driver should allocate no more noncached memory than it needs and should free the memory as soon as it is no longer required.

6.4 Alignment

The alignment of the data structures in a driver can have a big impact on the driver’s performance and efficiency. Two types of alignment are important:

- Natural alignment for the data size
- Cache-line alignment

6.4.1 Natural Alignment

Natural alignment means aligning data according to its type. The Microsoft C compiler aligns individual data items on an appropriate boundary for their size. For example, UCHARs are aligned on 1-byte boundaries, **int**s on 4-byte boundaries, and LONGs and ULONGs on 4-byte boundaries.

Individual data items within a structure are also naturally aligned—the compiler adds padding bytes if required. When you compile, structures are aligned according to the alignment requirements of the largest member. Unions are aligned according to the requirements of the first member of the union. To align individual members of a structure or union, the compiler also adds padding bytes. When you compile a 32-bit driver, pointers are 32 bits wide and occupy 4 bytes. When you compile a 64-bit driver, pointers are 64 bits and occupy 8 bytes. A structure that contains a pointer, therefore, might require different amounts of padding on 32-bit and 64-bit systems. If the structure is used only internally within the driver, differences in padding are not important. However, you must ensure that the padding is the same on 32-bit and 64-bit systems in either of the following situations:

- The structure is used by both 32-bit and 64-bit processes running on a 64-bit machine.
- The structure might be passed to or used on 32-bit hardware as a result of being saved on disk, sent over the network, or used in a device I/O control request (IOCTL).

You can resolve this issue by using pragmas (as described below) or by adding explicit dummy variables to the structure just for padding. For cross-platform compatibility, you should explicitly align data on 8-byte boundaries on both 64-bit and 32-bit systems.

Proper alignment enables the processor to access data in the minimum number of operations. For example, a 4-byte value that is naturally aligned can be read or written in one cycle. Reading a 4-byte value that does not start on a 4-byte (or multiple) boundary requires an additional cycle, and the requested bytes must be pieced together into a single 4-byte unit before the value can be returned.

If the processor tries to read or write improperly aligned data, an alignment fault can occur. On x86 hardware, the alignment faults are invisible to the user. The hardware fixes the fault as described in the previous paragraph. On x64 hardware, alignment faults are disabled by default and the hardware similarly fixes the fault. On the Intel Itanium architecture, however, if an alignment fault occurs while 64-bit kernel-mode code is running, the hardware raises an exception. (For user-mode code, this is a default setting that an individual application can change, although disabling alignment faults on the Itanium can severely degrade performance.)

To prevent exceptions and performance problems that are related to misalignment, you should lay out your data structures carefully. When allocating memory, ensure that you allocate enough space to hold not just the natural data size, but also the padding that the compiler adds. For example, the following structure includes a 32-bit value and an array. The array elements can be either 32 or 64 bits long, depending on the hardware.

```
struct Xx {
   DWORD NumberOfPointers;
   PVOID Pointers[1];
};
```

When this declaration is compiled for 64-bit hardware, the compiler adds an extra 4 bytes of padding to align the structure on an 8-byte boundary. Therefore, the driver must allocate enough memory for the padded structure. For example, if the array could have a maximum of 100 elements, the driver should calculate the memory requirements as follows:

```
FIELD_OFFSET (struct Xx, Pointers) + 100*sizeof(PVOID)
```

The FIELD_OFFSET macro returns the byte offset of the Pointers array in the structure Xx. Using this value in the calculation accounts for any bytes of padding that the compiler might add after the NumberOfPointers field.

To force alignment on a particular byte boundary, a driver can use any of the following:

- The storage class qualifier **__declspec(align())** or the DECLSPEC_ALIGN() macro
- The **pack()** pragma
- The Pshpack*N*.h and Poppack.h header files

To change the alignment of a single variable or structure, you can use **__declspec(align())** or the DECLSPEC_ALIGN() macro, which is defined in the Windows DDK. The following type definition sets alignment for the ULONG_A16 type at 16 bytes, thus aligning the two fields in the structure and the structure itself on 16-byte boundaries:

```
typedef DECLSPEC_ALIGN(16) ULONG ULONG_A16;
typedef struct {
    ULONG_A16 a;
    ULONG_A16 b;
} TEST;
```

You can also use the **pack()** pragma to specify the alignment of structures. This pragma applies to all declarations that follow it in the current file and overrides any compiler switches that control alignment. By default, the DDK build environment uses **pack (8)**. The default setting means that any data item with natural alignment up to and including 8 bytes is naturally aligned, not necessarily 8-byte aligned, and everything larger than 8 bytes is aligned on an 8-byte boundary. Thus, two adjacent ULONG fields in a 64-bit aligned structure are adjacent, with no padding between them.

Another way to change the alignment of data structures in your code is to use the header files Pshpack*N*.h (pshpack1.h, pshpack2.h, pshpack4.h, pshpack8.h, and pshpack16.h) and Poppack.h, which are installed as part of the Windows DDK. The Pshpack*N*.h files change alignment to a new setting, and Poppack.h returns alignment to its setting before the change was applied. For example:

```
#include <pshpack2.h>

typedef struct _STRUCT_THAT_NEEDS_TWO_BYTE_PACKING {
    /* contents of structure
    ...
} STRUCT_THAT_NEEDS_TWO_BYTE_PACKING;

#include <poppack.h>
```

In the example, the pshpack2.h file sets 2-byte alignment for everything that follows it in the source code, until the poppack.h file is included. You should always use these header files in pairs. Like the **pack()** pragma, they override any alignment settings specified by compiler switches.

For more information about alignment and the Microsoft compilers, see the Windows DDK and the MSDN library, which are listed in the Resources section of this paper.

6.4.2 Cache-Line Alignment

When you design your data structures, you can further increase the efficiency of your driver by considering cache-line alignment in addition to natural alignment.

Memory that is cache-aligned starts at a processor cache-line boundary. When the hardware updates the processor cache, it always reads an entire cache line rather than individual data items. Therefore, using cache-aligned memory can reduce the number of cache updates necessary when the driver reads or writes the data and can prevent other components from contending for updates of the same cache line. Any memory that starts on a page boundary is cache-aligned.

Drivers typically allocate nonpaged, cache-aligned memory to hold frequently accessed driver data. If possible, lay out data structures so that individual fields are unlikely to cross cache line boundaries. The size of a cache line is generally from 16 to 128 bytes, depending on the hardware. The **KeGetRecommendedSharedDataAlignment** function returns the recommended alignment on the current hardware.

Cache-line alignment is also important for shared data that two or more threads can access concurrently. To reduce the number of cache updates, fields that are protected by the same lock and are updated together should be in the same cache line. Structures that are protected by different locks and can therefore be accessed simultaneously on two different processors should be in different cache lines. Laying out data structures in this way prevents processors from contending for the same cache line, which can have a profound effect on performance.

For more information about cache line alignment on multiprocessor systems, see “Multiprocessor Considerations for Kernel-Mode Drivers,” which is listed in the Resources section at the end of this paper.

7 Kernel Stack

The kernel stack is a limited storage area that is used for information that is passed from one function to another and for local variable storage. The size of the stack can vary among different hardware platforms and different versions of the operating system, but it is always a scarce resource.

Although the stack is mapped into system space, it is considered part of the thread context of the original calling routine, not part of the driver itself. It is therefore guaranteed to be resident whenever the thread is running, but it can be swapped out along with the thread.

Because the kernel stack is associated with the thread context, functions such as **IoGetRemainingStackSize** (which returns the amount of kernel stack space that is currently available) and **IoGetStackLimits** (which returns the addresses of the stack) provide information about the kernel stack in the thread from which they are called. A driver can use these system routines to determine whether enough stack space remains to call a function to perform a task. If not, the driver can queue the task to a work item, which runs in a separate thread. The stack size and limit functions are useful primarily in file system drivers. Drivers for physical devices typically have shallow stack usage and therefore do not often require these functions.

Drivers should use the kernel stack conservatively. Heavily recursive functions that are passed many bytes of data can quickly deplete stack space and cause a system crash. Instead of passing large amounts of data, the driver should pass a pointer to the data.

On x64-based systems, Windows does not allow third-party software to allocate memory "on the side" and use it as a kernel stack. If the operating system detects one of these modifications or any other unauthorized patch, it generates a bug check and shuts down the system. To help ensure stability and reliability of the operating system and a better experience for customers, drivers for all platforms should avoid these practices.

8 Memory Allocation Techniques

The best technique for allocating memory depends on how the memory will be used. Table 3 summarizes the Windows kernel-mode memory allocation routines that are discussed in this section.

Table 3. Summary of Memory Allocation Routines

Name	Description
ExAllocatePool*Xxx*	Allocates paged or nonpaged, cached, and cache-aligned memory from the kernel-mode pool. **ExAllocatePoolWithTag** is the primary function for memory allocation.
AllocateCommonBuffer	Allocates a buffer for common-buffer DMA and ensures cross-platform compatibility.
MmAllocateContiguousMemory [**SpecifyCache**]	Allocates nonpaged, physically contiguous, cache-aligned memory. Drivers should not use these functions to allocate memory for DMA because the addresses that they return are not guaranteed to be compatible across all hardware configurations. Drivers should use **AllocateCommonBuffer** instead.
MmAllocateMappingAddress	Reserves virtual addresses in a specific range for later use in an MDL, but does not map them to physical memory. The driver can later use **IoAllocateMdl**, **MmProbeAndLockPages**, and **MmMapLockedPagesWithReservedMapping** to map the virtual addresses to physical memory.
MmAllocateNoncachedMemory	Allocates nonpaged, noncached, page-aligned memory. Drivers rarely use this function; they must not use it to allocate buffers for DMA.
MmAllocatePagesForMdl	Allocates nonpaged, noncontiguous pages from physical memory, specifically for an MDL. An MDL can describe up to 4 GB. This function might return fewer pages than the caller requested. Therefore, the driver must call **MmGetMdlByteCount** to verify the number of allocated bytes. A driver can call this function to allocate physical memory from a particular address range, such as allocating memory for a device that cannot address memory above 4 GB. However, drivers should not use this function to allocate buffers for DMA. Only the Windows DMA routines guarantee the cross-platform compatibility that drivers require.

The following sections cover techniques for allocating memory for several specific purposes:

- Pool memory for long-term storage
- Lookaside lists
- Contiguous memory

8.1 Allocating Memory for Long-Term Storage

Pool memory is appropriate for most long-term driver storage requirements. The pool allocation routines allocate cached memory from the paged or nonpaged kernel-mode pools. Drivers should use pool memory unless they have other specific requirements, such as the following:

- The device or driver requires more than a page of physically contiguous memory. (Remember that page size varies depending on the machine architecture.)
- The device requires noncached or write-combined memory. Some video devices fall into this category.
- The device has constraints that limit the range of addresses it can use.

Pool memory is a limited resource, so drivers should allocate it economically. When you design your driver, plan your memory allocations according to memory type, allocation size, and allocation lifetime. Free the allocated memory as soon as you are done using it.

If your driver requires large amounts of nonpaged pool for long-term use, allocate the memory at startup if possible, either in the **DriverEntry** routine or in an *AddDevice* routine. Because the nonpaged pool becomes fragmented as the system runs, later attempts to allocate memory from the nonpaged pool could fail.

A driver should not, however, preallocate excessively large blocks of memory (several megabytes, for example) and try to manage its own allocations within that block. Although this technique might work for your driver, it often results in inefficient memory usage across the system and thus poor performance system-wide. Remember that other drivers and applications also require memory. Never allocate lots of memory “just in case” your driver might need it. As a general design guideline, consider preallocating only enough memory for one I/O transaction and then process the transactions serially. If the transactions all require same-sized allocations, consider using lookaside lists because these buffers are dynamically allocated.

Drivers can allocate cached memory from either the paged pool or the nonpaged pool by using the following **ExAllocatePool*Xxx*** routines:

- **ExAllocatePoolWithTag**
- **ExAllocatePoolWithTagPriority**
- **ExAllocatePoolWithQuotaTag**

Drivers must use the tagged versions of the pool allocation routines because the nontagged versions are obsolete. Both WinDbg and Driver Verifier use pool tags to track memory allocations. Tagging pool allocations can simplify finding memory-related bugs, including leaks. Be sure to use unique tags so that the debugging and testing tools can provide useful information.

Note

The pool type **NonPagedPoolMustSucceed** has been deprecated. Drivers should specify the pool type **NonPagedPool** or **NonPagedPoolCacheAligned** instead and should return the NTSTATUS value STATUS_INSUFFICIENT_RESOURCES if the allocation fails. The Driver Verifier flags any pool allocation that is specified as **NonPagedPoolMustSucceed** as fatal and causes a bug check. (Note that on versions of Windows earlier than Microsoft Windows Vista™ specifying **NonPagedPoolCacheAligned** is the same as specifying **NonPagedPool**. On

Windows Vista, however, specifying **NonPagedPoolCacheAligned** ensures that the system allocates cache-aligned memory.)

In the following example, the driver allocates a buffer from nonpaged pool, aligned on a processor cache-line boundary:

```
MyBuffer = ExAllocatePoolWithTag(NonPagedPoolCacheAligned,
                 MAX_BUFFER_SIZE,
                 'tseT');

if (MyBuffer == NULL)
{
    DebugPrint((1, " Can't allocate MyBuffer \n"));
    return(STATUS_INSUFFICIENT_RESOURCES);
    }
```

The example specifies the pool tag in reverse order so that it appears as “Test” in the debugger. (You should use a tag that uniquely identifies your driver.) In the example, MAX_BUFFER_SIZE specifies the number of bytes to allocate. Because of rounding and alignment, the driver might receive a buffer of exactly the requested size or a larger buffer. The **ExAllocatePool*Xxx*** routines align and round allocations as follows:

Size requested	Size allocated	Default alignment
Less than or equal to (PAGE_SIZE – **sizeof** (POOL_HEADER))	Number of bytes requested	8-byte boundary (16-byte on 64-bit Windows). Allocation is contiguous and does not cross page boundary.
Greater than (PAGE_SIZE – **sizeof** (POOL_HEADER))	Number of bytes requested, rounded up to next full page size	Page-aligned. Pages are not necessarily physically contiguous, but they are always virtually contiguous.

The pool header is 8 bytes long on 32-bit systems and 16-bytes long on 64-bit systems. The size of the pool header does not affect the caller.

To use the pool most efficiently, avoid calling the memory allocation routines repeatedly to request small allocations of less than PAGE_SIZE. If your driver normally uses several related structures together, consider bundling those structures into a single allocation at driver start-up. For example, the SCSI port driver bundles an IRP, a SCSI request block (SRB), and an MDL into a single allocation. By allocating the memory at start-up, you reduce the number of times your driver calls memory allocation routines and therefore the number of situations in which it must be prepared to handle an allocation failure. Eliminating such failure scenarios greatly simplifies the error paths in your driver.

Pool memory remains allocated until the driver explicitly frees it. Before the driver exits, it must call **ExFreePool** or **ExFreePoolWithTag** to free all the memory that it allocated with any of the **ExAllocatePool*Xxx*** variants. Failing to free the memory causes memory leaks, which can eventually slow system performance and cause other components to fail.

8.2 Creating and Using Lookaside Lists

A driver calls **ExInitialize[N]PagedLookasideList** to set up a lookaside list, **ExAllocateFrom[N]PagedLookasideList** to allocate a buffer from the list, and **ExFreeTo[N]PagedLookasideList** to free a buffer to the list.

Even if a lookaside list is allocated in paged memory, the head of the list must be allocated in nonpaged memory because the system scans it at DISPATCH_LEVEL to perform various bookkeeping chores. Typically, drivers store a pointer to the head of the list in the device extension.

For example, a network adapter driver that receives data into fixed-size blocks might set up a lookaside list for the blocks as follows:

```
ExInitializeNPagedLookasideList
      (&FdoData->RecvLookasideList,
       NULL,
       NULL,
       0,
       RecvBlockSize,
       'LciN',
       0);
```

The lookaside list in the example is allocated from nonpaged memory because the driver accesses it at DISPATCH_LEVEL or higher. The first parameter identifies the location in nonpaged memory where the driver stores the head of the list. This location must be at least **sizeof** (NPAGED_LOOKASIDE_LIST). The two NULL parameters represent the driver routines that the system calls to allocate and free buffers from the list. If the driver passes NULL, the system uses **ExAllocatePoolWithTag** and **ExFreePoolWithTag**. Although a driver can supply its own routines, using the system's allocation routines is generally more efficient. The fourth parameter supplies internal flags and must be zero. The fifth parameter, RecvBlockSize, specifies the size of each buffer in the list, and the sixth is the tag used for the pool allocation. The final parameter is also reserved for internal use and must be zero.

Later, to allocate and use a buffer from the list, the driver calls **ExAllocateFromNPagedLookasideList**, as follows:

```
pRecvBlock = ExAllocateFromNPagedLookasideList (
                 &FdoData->RecvLookasideList);
```

If the allocation fails for any reason, the allocation routine returns a NULL pointer. The driver should validate the result and either return an error or retry the operation until it receives a valid pointer.

To free a buffer that was allocated from the list, the driver calls **ExFreeToNPagedLookasideList**, passing a pointer to the head of the list and a pointer to the buffer to free, as follows:

```
ExFreeToNPagedLookasideList (
                 &FdoData->RecvLookasideList,
                 pRecvBlock);
```

To avoid using memory unnecessarily, drivers should free buffers as soon as they are no longer needed.

Finally, when the driver no longer needs any buffers from the list, it calls **ExDeleteNPagedLookasideList**.

```
ExDeleteNPagedLookasideList (
                 &FdoData->RecvLookasideList);
```

The list need not be empty because this function deletes any remaining buffers and then deletes the list itself.

Drivers that run on Windows XP and later versions can use lookaside lists to allocate scatter/gather lists for DMA, as described in "DMA Support in Windows Drivers," which is listed in the Resources section of this paper.

8.3 Allocating Contiguous Memory

Drivers that require a page or more of contiguous memory can allocate it by using one of the following:

- The DMA functions **AllocateCommonBuffer** or **GetScatterGatherList**
- **MmAllocatePagesForMdl**
- **MmAllocateContiguousMemorySpecifyCache**

To allocate memory for DMA buffers, drivers should call **AllocateCommonBuffer** or **GetScatterGatherList**. For more information about these DMA functions, see the Windows DDK and “DMA Support in Windows Drivers,” which are listed in the Resources section of this paper.

MmAllocatePagesForMdl allocates pages from a specified range of physical addresses. Therefore, it is useful for drivers of devices that cannot address memory in certain ranges. For example, some devices cannot address memory above 4 GB. The driver for such a device could use this function to ensure that a buffer was allocated in memory below 4 GB. However, that this function allocates independent pages that are not physically contiguous; that is, each entry in the MDL maps a page of memory, but pages are typically not contiguous with respect to each other. If your device requires no more than a page of contiguous memory at a time, but requires many such pages, **MmAllocatePagesForMdl** is appropriate. Do not use this function to allocate buffers for DMA—use the DMA routines instead.

Drivers that need larger amounts of physically contiguous memory should use **MmAllocateContiguousMemorySpecifyCache**. Drivers should try to allocate such memory during driver initialization because physical memory is likely to become fragmented as the system runs. A driver should allocate only as much contiguous memory as it needs, and it should free the memory as soon as it is no longer needed.

MmAllocateContiguousMemorySpecifyCache returns the base virtual address of the mapped region. The following sample shows how a driver might allocate contiguous memory in a specific range:

```
LowestAcceptable.QuadPart = 0;
BoundaryMultiple.QuadPart = 0;
HighestAcceptable.QuadPart = 0xFFFFFFFF;

MemBuff = MmAllocateContiguousMemorySpecifyCache(MemLength,
                LowestAcceptable,
                HighestAcceptable,
                BoundaryMultiple,
                MmNonCached);

if (MemBuff == NULL) {
    return(STATUS_INSUFFICIENT_RESOURCES);
    }
```

In the example, the driver allocates contiguous, noncached memory in the first 4 GB of the physical address space. MemLength specifies the number of required contiguous bytes. The driver passes zero in the LowestAcceptable parameter to indicate that the range of addresses it can use starts at zero, and it passes 0xFFFFFFFF in the HighestAcceptable parameter to indicate that the range ends at 4 GB. It sets BoundaryMultiple to zero because there are no boundaries within the specified range that the memory should not cross. The function returns a full multiple of the system’s page size. If MemLength is not a multiple of the page size, the function rounds up to the next full page.

Do not use **MmAllocateContiguousMemorySpecifyCache** to allocate buffers for DMA and then call **MmGetPhysicalAddress** to translate the returned virtual address to an address with which to program your DMA controller. This approach does not work on all hardware, and your device and driver will not be cross-platform compatible. Use **AllocateCommonBuffer** or **GetScatterGatherList** instead.

9 Accessing Device Registers

Device registers are among a device's hardware resources, which are assigned to the device during Plug and Play enumeration. Registers can be mapped into memory or into the 64-KB I/O space, depending on the type of device, the bus to which it is attached, and the underlying hardware platform. The x86, x64, and Itanium processors, along with most common buses (including PCI, PCI Express, ISA, and EISA), support both memory mapping and I/O mapping. Some other hardware architectures, however, support only memory mapping.

As described earlier in "Physical Address Space," I/O mapping is a holdover from early microprocessor designs. Today, however, most machines have more than enough physical address space to hold all the device registers. As a result, newer devices are typically memory-mapped. In a memory-mapped device, the device registers are mapped to addresses in the physical memory space. The driver then maps those addresses into the virtual address space before it uses them.

Device registers that are mapped into I/O space are read and written with the READ_PORT_*Xxx* and WRITE_PORT_*Xxx* macros. Device registers that are mapped into memory are read and written with the READ_REGISTER_*Xxx* and WRITE_REGISTER_*Xxx* macros. Thus, I/O mapping is sometimes called PORT mapping and the mapped resources are called PORT resources (with or without the capital letters). Similarly, memory mapping is sometimes called REGISTER mapping and the corresponding resources are called REGISTER resources.

At device enumeration, the bus driver requests I/O or memory mapping for each device resource, in response to queries from the Plug and Play manager. The Plug and Play manager assigns raw and translated resources to the device and passes a list of the assigned resources to the device's function driver in the IRP_MN_START_DEVICE request.

The raw resource type reflects how the hardware is wired. The translated resource type indicates where the resource is mapped on the current system and thus whether the driver should use PORT or REGISTER macros to read and write the resource.

Knowing how your device hardware is wired does not necessarily tell you how its resources will be mapped because some chipsets change the mapping. Therefore, drivers must be prepared to support both types of mapping for each register. Defining wrappers for the macros is a common strategy for supporting both types of mappings.

To determine the mapping for each individual hardware resource, a driver inspects the resource lists passed in the IRP_MN_START_DEVICE request. Resources of type **CmResourceTypeMemory** are memory-mapped, and resources of type **CmResourceTypePort** are I/O-mapped. Each resource has a starting address and a length. Drivers should parse and save the translated resource lists in the device extension, and then use the translated resources to read and write device registers. Most drivers do not need the raw resources.

The resource list also contains a set of flags that provide additional information about the assigned resources. For I/O-mapped resources, the flags indicate the type of decoding that the device performs and whether the device decodes 10, 12, or 16 bits of the port address. For memory-mapped resources, the flags indicate whether the memory can be read, written, or both; whether it is cached or write-combined; whether it can be prefetched; and whether the device uses 24-bit addressing.

To read or write a register in I/O space by using the READ_PORT_*Xxx* and WRITE_PORT_*Xxx* macros, a driver should cast the port address to a pointer of type PVOID or PUCHAR (if pointer addition is required). For example:

```
PortPointer =
     (PVOID)(ULONG_PTR)Resource->u.Port.Start.QuadPart
```

Before reading or writing a resource in memory space, the driver must call **MmMapIoSpace** to get a virtual address for the translated physical address. The driver then must use the returned virtual address in calls to the READ_REGISTER_*Xxx* and WRITE_REGISTER_*Xxx* macros.

Remember that the driver should use the correct type of pointer for the data to be read or written with the PORT and REGISTER macros. For example, a driver should use PUCHAR to read or write an unsigned character, PUSHORT for a short integer, and so forth.

The following example shows how a driver gets the resources that have been assigned to its device. The driver receives the translated resource list in the start-device IRP at Parameters.StartDevice.AllocatedResourcesTranslated. The value at this location points to a CM_RESOURCE_LIST that describes the hardware resources that the Plug and Play manager assigned to the device. The base address registers (BARs) are always returned in order.

In the sample, the driver parses the resource list to determine whether its control and status registers are mapped to I/O space (**CmResourceTypePort**) or memory (**CmResourceTypeMemory**). It then sets pointers to the macros it will use to access the registers in its device extension at FdoData->ReadPort and FdoData->WritePort.

```
partialResourceListTranslated =
       &stack->Parameters.StartDevice.
       AllocatedResourcesTranslated->List[0].
       PartialResourceList;

resourceTrans =
       &partialResourceListTranslated->PartialDescriptors[0];

for (i = 0;
     i < partialResourceListTranslated->Count;
     i++, resourceTrans++) {

     switch (resourceTrans->Type) {

       case CmResourceTypePort:
          // The control and status (CSR) registers
          // are in port space.

          // Save the base address and length.
          FdoData->IoBaseAddress =
             (PVOID)(ULONG_PTR)(
                 resourceTrans->u.Port.Start.QuadPart);
          FdoData->IoRange = resourceTrans->u.Port.Length;

          // All accesses are USHORT wide, so
          // create an accessor table to read and write
```

```
            // the ports.
            FdoData->ReadPort = READ_PORT_USHORT;
            FdoData->WritePort = WRITE_PORT_USHORT;

          break;

         case CmResourceTypeMemory:

            // The CSR registers are in memory space.
            // This device requires a CSR memory space that
            // is 0x1000 in size. Assert if it's not.
            //
            ASSERT(resourceTrans->u.Memory.Length == 0x1000);

            // The driver can read and write registers in
            // memory space by using the READ_REGISTER_*
            // macros.

            FdoData->ReadPort = READ_REGISTER_USHORT;
            FdoData->WritePort = WRITE_REGISTER_USHORT;

            // Save the starting address.
            FdoData->MemPhysAddress =
                     resourceTrans->u.Memory.Start;

            // Map the returned physical address to a
            // virtual address. We can use the VA to call
            // call READ_REGISTER* and WRITE_REGISTER* macros.
            // We can also define the CSRAddress structure that
            // maps on top of the registers. We can then
            // read and write registers directly by name.

             FdoData->CSRAddress = MmMapIoSpace(
                      resourceTrans->u.Memory.Start,
                      resourceTrans->u.Memory.Length,
                      MmNonCached);
             if (FdoData->CSRAddress == NULL) {
                DebugPrint(ERROR, DBG_INIT,
                "MmMapIoSpace failed\n");
             status = STATUS_INSUFFICIENT_RESOURCES;
             }

        break;
}
```

If the registers are in physical memory space, the driver calls **MmMapIoSpace** to map them into system-space virtual memory. Specifying noncached memory is important to ensure that new values are immediately written to the device, without being held in the processor cache. Control and status registers should always be mapped uncached. On-board memory can be mapped cached, uncached, or uncached/write-combined, depending on how it will be used.

As an alternative to using the READ_REGISTER_*Xxx* and WRITE_REGISTER_*Xxx* macros to read and write the registers, you can define a structure that matches the register layout and then read and write the fields of that structure. In the preceding sample, the structure at FdoData->CSRAddress is defined for this purpose. If the order in which you set bits in the registers is important, you should declare the register fields of the structure as **volatile** to prevent compiler reordering of read and write instructions. Depending on exactly what a particular code sequence does, you might also need to use memory barriers to prevent some hardware architectures from reordering memory access. How to use both the **volatile** keyword and memory barriers is described in “Multiprocessor Considerations for Kernel-mode Drivers,” which is listed in the Resources section at the end of this paper.

The sample code saves pointers to the macros it will use to read and write the registers in its device extension. In most cases, this technique works because the PORT and REGISTER macros have the same semantics. For example, READ_PORT_USHORT and READ_REGISTER_USHORT operate in the same way on all platforms. However, READ_PORT_BUFFER_UCHAR and READ_REGISTER_BUFFER_UCHAR are exceptions. On some platforms, READ_PORT_BUFFER_UCHAR updates the port address after it reads a byte, but on others it does not. To work around this issue, avoid using both of these macros. Instead, call READ_PORT_UCHAR or READ_REGISTER_UCHAR in a loop until the processor has read the correct number of bytes.

10 Accessing Memory for I/O Buffers

For some types of I/O, the I/O manager allocates buffers and maps them for use by the driver. For others, the driver must perform additional mapping.

Buffered I/O requests (DO_BUFFERED_IO and METHOD_BUFFERED) include a pointer to a system-space buffer that contains a copy of the data that was passed in the caller’s user-space buffer. The driver can read and write to the buffer directly through this pointer. The driver does not need to allocate buffer space.

Direct I/O requests (DO_DIRECT_IO, METHOD_IN_DIRECT, METHOD_OUT_DIRECT, METHOD_DIRECT_TO_DEVICE, and METHOD_DIRECT_FROM_DEVICE) include an MDL that describes a user-space buffer that has been locked into physical memory. The MDL maps the virtual addresses in user space to the processor-relative physical addresses that describe the buffer. To read or write to the buffer, the driver can use the user-space buffer pointer only while it is running in the context of the calling process, and it must take measures to prevent security breaches. Typically, however, the driver is running in an arbitrary process context and must instead use system space virtual addresses. To obtain the system-space virtual addresses that correspond to the addresses in the MDL, the driver passes the MDL pointer to **MmGetSystemAddressForMdlSafe**. The returned value is a system-space virtual address through which the driver can access the buffer.

Requests for neither buffered nor direct I/O (METHOD_NEITHER) include an unvalidated pointer to a buffer in user space. If the driver will access the buffer only from the context of the requesting user-mode thread, it can do so through the user-space address (the pointer)—but only after validating the address and access method and only within an exception handler. If the driver will access the buffer from an arbitrary thread context, it should create an MDL to describe the buffer, validate the user-space address, lock the buffer into the physical address space, and then map the buffer into system space.

Although the dispatch routines of many drivers run in the context of the calling thread, you can assume that this is true only in file-system drivers and other highest-level drivers. In other drivers, you must acquire a system-space virtual address for the user-space buffer. For more information about the context in which specific standard driver routines are called, see “Scheduling, Thread Context, and IRQL,” which is listed in the Resources section at the end of this paper.

A driver that accesses a user-mode buffer in the context of the user-mode thread must do so carefully to avoid corrupting memory and causing system crashes. The driver must wrap all attempts to access the buffer in an exception handler because another thread might free the buffer or change the access rights while the driver is accessing it. Furthermore, the driver must access the buffer only at IRQL PASSIVE_LEVEL or APC_LEVEL because the pages in the buffer are not locked into memory.

To validate the buffer, the driver calls **ProbeForRead** or **ProbeForWrite**, passing a pointer to the buffer, its length, and its alignment. Both of these macros raise a STATUS_ACCESS_VIOLATION exception if the address and length do not specify a valid range in user space and a STATUS_DATATYPE_MISALIGNMENT exception if the beginning of the address range is not aligned on the specified byte boundary. In addition, **ProbeForWrite** raises the STATUS_ACCESS_VIOLATION exception if the buffer is not available for write access.

The following example shows how a driver that is running in the context of the user-mode process can validate for read access a buffer that is passed in an IOCTL request:

```
inBuf = irpSp->Parameters.DeviceIoControl.Type3InputBuffer;
inBufLength =
     irpSp->Parameters.DeviceIoControl.InputBufferLength;

try {
    ProbeForRead( inBuf, inBufLength, sizeof( UCHAR ) );
}
except(EXCEPTION_EXECUTE_HANDLER)
{

    ntStatus = GetExceptionCode();
    SIOCTL_KDPRINT((
        "Exception while accessing inBuf 0X%08X in “
        “METHOD_NEITHER\n",
         ntStatus));
    break;
}
```

Validating the buffer once upon receiving the I/O request is not enough. The driver must enclose every access in an exception handler and use **ProbeForRead** or **ProbeForWrite** before reading or writing the buffer.

Most drivers, however, cannot depend upon being in the user process context when they read or write a buffer that was passed in an IOCTL. Any driver that might need to access the buffer from an arbitrary thread context must map the buffer into system space as follows:

1. Allocate and initialize an MDL that is large enough to describe the buffer.
2. Call **MmProbeAndLockPages** to probe the buffer for validity and access mode. If the buffer is valid, **MmProbeAndLockPages** locks its pages into memory.
3. Map the locked pages into system space by calling **MmGetSystemAddressForMdlSafe**. Even though the pages are resident in memory, the driver must access them through a system-space address because the user-space address could become invalid at any time either because of legitimate system actions (such as trimming the page from the working set) or inadvertent or malicious user actions (such as deleting the buffer in another thread).

The following example shows how to perform these steps:

```
// Allocate MDL, passing the virtual address supplied in the
// IRP, its length, FALSE to indicate that it is not a
// secondary buffer, TRUE to indicate that the pages should
// be charged against the quota of the requesting process,
// and NULL to indicate that the MDL is not directly
// associated with an I/O request.

mdl = IoAllocateMdl(inBuf, inBufLength, FALSE, TRUE, NULL);
if(!mdl)
{
    ntStatus = STATUS_INSUFFICIENT_RESOURCES;
    return;
}
try
{
    MmProbeAndLockPages(mdl, UserMode, IoReadAccess);
}
except(EXCEPTION_EXECUTE_HANDLER)
{
    ntStatus = GetExceptionCode();
    SIOCTL_KDPRINT((
        "Exception while locking inBuf 0X%08X in "
        "METHOD_NEITHER\n",
         ntStatus));
    IoFreeMdl(mdl);
}

//
// Map the physical pages described by the MDL into system
// space. Note that mapping a large buffer this way
// incurs a lot of system overhead.
//

buffer =
      MmGetSystemAddressForMdlSafe(mdl, NormalPagePriority );

if(!buffer) {
        ntStatus = STATUS_INSUFFICIENT_RESOURCES;
        MmUnlockPages(mdl);
        IoFreeMdl(mdl);
}
```

The buffer pointer that **MmGetSystemAddressForMdlSafe** returns is a valid system-space pointer to the user's buffer. Because the pages in the buffer are resident (locked) in memory, the driver can now access them without an exception handler.

When the driver has finished using the buffer, it must unlock the pages and free the MDL, as follows:

```
MmUnlockPages(mdl);
IoFreeMdl(mdl);
```

The driver must call these functions in the order shown. Freeing the MDL before unlocking the pages could result in a system crash because the list of pages to be unlocked is freed along with the MDL and is thus not guaranteed to be accessible after the **IoFreeMdl** has returned.

11 Mapping Device Memory and Registers into User Space

As a general rule, drivers should avoid mapping device memory and registers into user space. Instead, drivers should share memory with user-mode components only through mapped buffers (typically passed in IOCTLs), as shown in the METHOD_NEITHER I/O example earlier in this paper.

Exposing hardware to user-mode control is risky and poses potential threats to system security. Doing so requires significantly more code than would be required if access were strictly limited to kernel mode, and it requires additional attention to security issues. If your reason for such a design is to improve performance, consider this decision carefully. The transition to kernel mode does not require much overhead when compared to the additional coding, debugging, and testing that are involved with user-mode access. User-mode processes cannot be trusted.

The additional code is required for the following reasons:

- The hardware register interfaces must be completely secure before they are exposed in user mode. You must carefully analyze the hardware to determine how a user-mode application might use the exposed interfaces. Remember to consider every possible option—including both incorrect and intentionally malicious misuse. Then you must ensure that you do not expose any paths through which a user-mode application could cause the bus to hang, perform DMA into random areas of memory, or crash the system for any other reason.
- A user-mode process might duplicate the handle to the allocated memory into another process, which can then issue I/O requests. To protect against malicious user applications, the driver must check every I/O request to ensure that it came from the process into which the memory was originally mapped. In addition, the driver must take additional steps to ensure that it is notified when the original process ends, even if another process still has an open handle to the mapped memory.
- The driver must unmap the memory when the user-mode process exits, and it must synchronize the unmapping operation with other I/O operations on the handle.
- Hibernation and stop-device requests require additional handling so that the driver can unmap the memory if the user-mode application has not closed its handle.

12 Shared Memory and Synchronization

Many drivers share memory with other components. For example:

- Adjacent drivers in the same stack might share memory mapped for their device.
- A driver might create a worker thread that uses context information or memory locations that are also accessed elsewhere in the driver.
- A driver might need to share a buffer with a user-mode application.

In addition, any driver routines that are reentrant or can run concurrently might share memory.

In all such situations, the driver must ensure the integrity of the data by synchronizing access to the shared locations. The most appropriate synchronization technique depends on how the memory is used and which other components use it. For a complete description of the synchronization techniques for kernel-mode components, see “Locks, Deadlocks, and Synchronization,” which is listed in the Resources section at the end of this paper.

In addition, some drivers must share memory with user-mode components. User-mode components cannot allocate virtual memory in the system address space.

Although it is possible to map system-space addresses into user space, drivers should avoid doing so for security reasons. Drivers and user-mode components must use other strategies for sharing memory.

“Managing User-Mode Interactions: Guidelines for Kernel-Mode Drivers” describes specific techniques for sharing buffers and section objects with user-mode components. This paper is listed in the Resources section at the end of this paper.

13 Testing and Troubleshooting

Bugs related to memory access and allocation are common, but they can be difficult to find. Symptoms can range from a gradual slowing of system performance because of memory leaks to an abrupt crash if a driver attempts to access paged memory at the wrong IRQL. Crashes caused by such bugs can occur long after a driver accesses an invalid location. Corrupted memory can cause errors and crashes in processes that are completely unrelated to the offending driver.

Some of the most common memory-related errors involve the following:

- Attempts to access memory that has already been freed.
- Attempts to access beyond the end of allocated memory (memory and buffer overruns).
- Attempts to access before the start of allocated memory (memory and buffer underruns).
- Attempts to access paged memory at IRQL DISPATCH_LEVEL or higher.
- Failure to free memory (memory leaks).
- Freeing the same memory twice.
- Failure to check the status after attempting to allocate or reference memory.

13.1 Tools for Discovering Memory-Related Errors

Many of the tools provided with the Windows DDK offer features specifically designed to find memory-related errors. For example:

- Driver Verifier
  Driver Verifier has options to use the kernel special memory pool, track memory pool usage, check DMA operations, and simulate low-resource situations.
- Global Flags (GFlags)
  GFlags sets flags that enable pool tagging, provide for buffer overrun and underrun detection, and specify tags for memory allocations from the kernel special pool.
- Call Usage Verifier (CUV)
  CUV checks to ensure that data structures that must be in nonpaged memory are actually there.
- PoolMon and PoolTag
  PoolMon and PoolTag gather and display data about memory allocations. Tracking this data can help you to find memory leaks.
- PRE*f*ast with driver-specific rules
  PRE*f*ast can detect potential errors in source code by simulating execution of all possible code paths on a function-by-function basis. Errors detected by PRE*f*ast include memory and resource leaks, excessive stack use, and incorrect usage of numerous kernel-mode functions.

The best way to test your driver is to use all of these tools repeatedly, both independently and in combination. Using CUV with Driver Verifier can be particularly useful.

You should also test on various memory configurations. To change the memory configuration of the test system, you can use various boot parameters.

The /3GB switch expands user space to 3 GB, leaving only 1 GB for all kernel-mode components.

On x64 and Itanium systems, and on x86 systems that have PAE enabled and more than 5 GB of physical memory, the /nolowmem switch loads the system, drivers, and applications into memory above the 4-GB boundary. Testing under these conditions helps you check for errors that are related to address truncation when using memory above 4 GB and inappropriate reliance on memory below 4 GB.

The /maxmem and /burnmemory switches also limit the memory available to the system by reducing the available memory by a specific amount. The /maxmem switch can be used with Windows 2000 or later versions of Windows. The /burnmemory switch is more precise and can be used with Windows XP and later versions.

13.2 Best Practices for Programming and Testing

Good programming and testing techniques can help you prevent problems as well as find them. The following should be standard practices for all driver writers:

- Always use pool tags when allocating memory. Driver Verifier, GFlags, PoolMon, and PoolTag use these tags in tracking pool allocations, and the debuggers display a buffer's tag along with its contents.
- Validate the length of every buffer before accessing it.
- Validate every user-space address by probing within an exception handler.
- Check the returned status every time you try to allocate or access memory. Never assume that an operation cannot fail. If an error occurs, return a failure status from the driver or modify and retry the operation if appropriate. Remember: if an attempt to access memory fails because of an invalid pointer and your driver fails to catch the error when it occurs, the driver could propagate the addressing error, resulting in errors that appear much later and seem unrelated to the driver.
- Always use Driver Verifier.
- Test every driver on both single-processor and multiprocessor hardware. Because Windows treats hyper-threaded processors as two CPUs, you can perform minimal multiprocessor testing on a machine with a single hyper-threaded processor. In effect, a dual-processor hyper-threaded machine provides four processors and serves as better test system. The new multicore processors would be an even better test system than the hyper-threaded systems.

For additional suggestions for testing, see "Testing for Errors in Accessing and Allocating Memory", which is listed in the Resources section at the end of this paper.

14 Summary of Guidelines

The following is a summary of guidelines for memory allocation and usage in kernel-mode drivers.

- Do not make assumptions about the hardware platform, the pointer size, or the location or size of the virtual address spaces. Make your driver platform-independent by using macros, types, and system-defined constants, such as PHYSICAL_ADDRESS, PAGE_SIZE, and so forth.
- Lay out data structures efficiently, considering natural and cache-line alignment and the processor page size. When possible, reuse allocated data structures. For more information about using memory efficiently, see “Six Tips for Efficient Memory Use,” which is listed in the Resources section at the end of this paper.
- Allocate only as much memory as your device and driver require for normal operations.
- Always check the translated resources to determine whether your device registers are mapped into memory or I/O space.
- Do not allocate excessive amounts of memory that your driver might never use. Your device must coexist with other hardware and applications.
- Use the system’s DMA routines. Replacing them with your own routines is almost guaranteed to cause your driver to fail on some hardware configurations.
- Use lookaside lists when your driver needs fixed-size, reusable buffers.
- Test your driver with as many different tools, and on as many different hardware configurations, as possible.

15 Resources

Windows DDK
http://www.microsoft.com/whdc/devtools/ddk/default.mspx

Kernel-Mode Driver Architecture Design Guide
Memory Management
Synchronization Techniques
Driver Programming Techniques (“64-bit Issues”)

Kernel-Mode Driver Architecture Reference
Standard Driver Routines
Driver Support Routines

Driver Development Tools
Boot Options for Driver Testing and Debugging
Tools for Testing Drivers ("Driver Verifier," "GFlags with Page Heap Verification," and "PoolMon")

Appendix
Processor-Specific Information

Kernel-Mode Fundamentals
Scheduling, Thread Context, and IRQL
http://www.microsoft.com/whdc/driver/kernel/IRQL.mspx

Locks, Deadlocks, and Synchronization
http://www.microsoft.com/whdc/driver/kernel/locks.mspx

Multiprocessor Considerations for Kernel-Mode Drivers
http://www.microsoft.com/whdc/driver/kernel/MP_issues.mspx

DMA
DMA Support in Windows Drivers
http://www.microsoft.com/whdc/driver/kernel/dma.mspx

Sharing Memory with User-Mode
User-Mode Interactions: Guidelines for Kernel-Mode Drivers
http://www.microsoft.com/whdc/driver/kernel/KM-UMGuide.mspx

Testing
Testing for Errors in Accessing and Allocating Memory
http://www.microsoft.com/whdc/driver/security/mem-alloc_tst.mspx

Performance
Six Tips for Efficient Memory Use
http://www.microsoft.com/whdc/driver/perform/mem-alloc.mspx

MSDN
“Windows Data Alignment on IPF, x86, and x86-64”
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vcconWindowsDataAlignmentOnIPFX86X86-64.asp

Windows Hardware and Driver Central
Includes Windows Driver Development Kits (DDK), Windows Hardware Compatibility Test (HCT) Kits, and Windows Logo Program requirements
http://www.microsoft.com/whdc/default.mspx

Books
Inside Microsoft Windows 2000, Third Edition
Solomon, David A. and Mark Russinovich. Redmond, WA: Microsoft Press, 2000.