Summit Soft Consulting
Professional Hardware / Software Co-Design

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA

Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

History of Windows Device Drivers

By John Gulbrandsen,
Summit Soft Consulting
October 27, 2015

Summit Soft Consulting
Professional Hardware / Software Co-Design

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA

Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

Summit Soft Consulting
Professional Hardware / Software Co-Design

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA

Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

The evolution of Windows

In order to understand the architectural differences between the various Windows platforms and the different device driver models, a historical walkthrough of the MS-DOS and Windows platforms is provided below. As will be explained, Windows 95 / 98 / ME have their roots in MS-DOS 1.0 that dates back to 1981 while Windows XP and newer platforms have their roots in Windows NT version 3.1 that was introduced in 1992.

MS-DOS 1.0

MS-DOS 1.0 only supported a fixed number of device types via *resident* device drivers compiled into the operating system. There was no way to add drivers for custom devices without modifying the operating system binary images via patching them. Custom versions of DOS were created using the *DOS OEM Adaption Kit* (OAK), which was not sold to the public. The OAK contained full source code to IO.sys as well as object files and binaries to the rest of the operating system.

Applications used a CPU register – based call interface to the operating system and BIOS routines. Due to limitations in memory size and CPU speed, applications were developed in assembly language, guaranteeing the most efficient implementation.

MS-DOS 2.0

MS-DOS 2.0 introduced *Installable Device Drivers* which enabled a custom device to be added to an existing system. Prior versions of MS-DOS only supported a fixed set of hardware via *Resident Device Drivers* specifically compiled into the operating system binary image by the system manufacturer.

Since DOS and BIOS routines exposed their services via a low-level INT (interrupt instruction) interface, Installable Device Drivers were developed in assembly language. Parameters and return values were passed via CPU registers, just as in MS-DOS 1.0.

There are two kinds of MS-DOS Installable Device Drivers; *Block Device Drivers* and *Character Device Drivers*. Examples of *Character Device Drivers* are the console

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA
Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

(keyboard, screen) and a modem, which both transmit a byte at a time. Examples of *Block Device Drivers* are Floppy Disks and Hard Disks, which transfer data in larger chunks.

Note that Character Device Drivers have names such as COM1, PRN, CON while Block Device Drivers are named A:, B:, C: etc. These device names carry through to today's modern Windows platforms.
Key parts of an MS-DOS device driver are two driver routines called "Strategy" and "Interrupt" routines. These, together, execute a specific I/O operation on a particular device. Note that, despite the "interrupt" routine name, the MS-DOS device driver interface is a *polling* interface and *is not* interrupt driven. Instead, the "Strategy" routine is passed the address of a *Request Header* structure which specifies the specifics of the I/O operation. Immediately after, the *Interrupt* routine is called to do the actual I/O. The result of the I/O operation is placed in the Request Header's 'Status' field.

Installable Device Drivers are loaded into MS-DOS 2.0 (and newer) via the 'device=XXX.sys" syntax in the config.sys system file.

As an example of the simplicity of MS-DOS device drivers, note that the complete Device Driver Reference section in the MS-DOS Programmer's Reference is only 47 pages long!

Normally, a custom hardware device used I/O mapped registers, accessible via the INP/OUTP 8088 machine instructions. However, a hardware device could also be designed to allow memory-mapped register access via direct x86 MOV instructions. Such memory-mapped locations were normally jumper-configurable to avoid clashes with other expansion cards in the system. DMA and I/O locations would also be jumper configurable.

Windows 1.x

Windows 1.x was merely a graphical application running on top of MSDOS. It ran on Intel 8086 CPUs or newer CPUs (80286 etc) emulating the 8086. Windows 1.x used MSDOS and the machine's BIOS for all system calls like memory allocations, file handling, graphics etc. Since Windows 1.x was running on top of MSDOS the infamous memory limitation of 640KB existed. Drivers for the Windows 1.x platforms talked directly to the hardware via the IN and OUT assembly instructions. Since the 8086 didn't support the concept of user and kernel execution modes, no protection between different applications existed. The application that currently executed owned the whole machine since no multitasking was supported. Microsoft calls Windows executing in 8086-mode REAL mode Windows.

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA

Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

Windows 2.x

Windows 2.x added support for the new 80286 CPU from Intel. By switching the CPU into the new 'Protected' mode, access to 16MB of RAM was made possible. When the CPU was executing in protected mode a new, more powerful, instruction set was also available. When the CPU is running in protected mode Microsoft refers to Windows running in Standard mode. Windows 2.x could still run in REAL 8086 mode either on the 8086 CPU or on the 80286 CPU running in 8086-mode. Windows 2.x still depended on MSDOS and BIOS for all system calls such as memory and file management. To call such system services while running in protected (standard) mode, Windows had to switch the CPU back to REAL mode because MSDOS and BIOS only run in REAL mode. Because there is no instruction for switching from protected to real mode, the CPU had to be reset for every system call. The device driver model is the same as for Windows 1.x; applications talk directly to the hardware by using IN and OUT instructions. Because Windows 2.x still was an application running on top of MSDOS no multitasking was supported.

Windows 3.0

Windows 3.0 was released in 1990. Windows 3.0 is interesting in that it's the first Windows platform that directly lends its architecture to Windows 95/98. The major improvement over Windows 2.x is that support for the new Intel 80386 CPU was added, which enabled access to up to 4GB of memory. Windows 3.0 also used the new 'virtual machine' feature of the 80386 CPU that enabled Windows to pre-emptily multitask several MSDOS application concurrently running in MSDOS windows on the desktop. Microsoft calls this mode 'Enhanced mode Windows'. Note that only MSDOS applications were multitasked, Windows applications could only co-operably multitask in that the applications had to give away the only thread of execution to another application or else all Windows applications would stop their execution. Since each concurrently executing MSDOS application thinks it owns the whole machine, hardware resources must somehow be shared between all the concurrently executing MSDOS applications. (Because of backward compatibility, MSDOS applications must live in an environment that looks like the old MSDOS where the running application owned the whole machine).

To support this virtualization of hardware resources, Virtual Device Drivers or VxD drivers were introduced. When Windows 3.0 is running in Enhanced mode, all applications are running in the CPUs ring 3 mode (User mode) while all VxDs are running in the CPUs ring 0 or Protected mode. User mode applications only have access to a subset of the instruction set of the CPU, especially the IN and OUT instructions are

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA
Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

off limits for the user mode applications. When a user mode MSDOS applications, for instance, tries to directly access hardware via IN and OUT instructions, the CPU notices that the applications does not have the rights to use these instructions and therefore throws an exception that switches the CPU into protected mode and gives the control to a VxD driver that virtualizes the device that the application tried to access. It's the job of the VxD driver to save the state of the shared resource, switch to the state of the resource that existed the last time the MSDOS application accessed it and then return the execution to the user mode application. The user mode application knows nothing about the VxD driver's existence and therefore old MSDOS applications still can execute unchanged in Windows 3.0.

Windows 3.0 was also the first Windows platform that had protected-mode VxD device drivers. This helps structure applications since there is a clear division between user mode applications and kernel mode drivers handling the hardware. The interface for calling these VxD drivers are assembly language, parameters and return values are passed in CPU registers. Windows 3.0 still relied on MSDOS and the BIOS for system service like memory management and file I/O. This limitation was more serious than in Windows 2.x because now all concurrently executing MSDOS applications making calls to MSDOS and BIOS had to be serialized by the system. The reason for this is that MSDOS and the BIOS routines are not re-entrant. This bottleneck in the system is one that Microsoft was trying to remove in Windows 3.1 as explained below.

Windows 3.1

Windows 3.1, released in 1992, removed many of the bottlenecks with the serialized MSDOS/BIOS calls by replacing many of the REAL mode MSDOS/BIOS routines with new routines exported by alternative VxD drivers. Since the VxD drivers are re-entrant, no serialization of system calls was necessary. These new 32-bit VxDs were also much more efficient since they did not have to execute in REAL mode 8086 mode like the MSDOS/BIOS routines. This greatly enhanced the performance of the system.

By replacing the memory management in MSDOS and the BIOS with VxD-based versions, Windows could use the new page swapping features of the 80386 CPU to support virtual memory. This was much faster than the old REAL-mode calls to MSDOS or to the BIOS to swap pages in or out of memory. Windows 3.1 however still used MSDOS and the BIOS for disk and file I/O. This limitation will be removed in the newer Windows 95 as later will be explained.

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA

Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

There also existed a 'Windows for Workgroups' version of Windows 3.1, which was basically Windows 3.1 with networking support added. The internal architecture is mostly the same as Windows 3.1.

The VxD device driver model in Windows 3.1 is the same as in Windows 3.0. Windows 3.1 introduced much better multimedia support than Windows 3.1 with support for both AVI video and WAVE audio playback using the WAVE APIs. Game developers however still regarded the Windows platform incapable of supporting high-performance games so they bypassed Windows altogether and ran the games directly in MSDOS. Microsoft addressed this problem by the introduction of DirectX in Windows 95 as later will be explained.

Windows NT 3.x

Windows NT 3.1, released in 1992, was a completely new operating system. It was designed from scratch to support preemptive multitasking and multithreading, virtual memory, networking, etc. It was designed to be robust which meant that Windows applications were protected from each other in separate 4GB memory spaces unlike Windows 3.1 that ran all Windows applications in the same memory space. NT 3.1 also introduced the WIN32 API that allows application developers to create applications running in 32-bit mode that is much faster that the old 16-bit WIN16 applications.

NT 3.1 is completely written in C except for small parts of the scheduler and dispatcher that are written in assembly. One of the reasons for this is that NT was designed to be portable between different hardware platforms unlike Windows 3.1 that only runs on Intel x86 CPUs. NT 3.1 also introduces a completely new, cleaner, layered device driver model in which several drivers can be stacked on top of each other. I/O requests are sent between the drivers using I/O Request Packets (IRPs) as transports. An IRP is a structure that represents an I/O request. Since these new NT-style drivers are written in C instead of in assembly as in the Windows VxD case it's much faster to write NT-style drivers.

Windows NT 3.1 required an Intel 386 CPU or newer to run (or one of the other supported Alpha, MIPS or Power-PC CPUs) since it relied on the virtual memory functionality of the CPU. Most MSDOS applications are still fully supported by executing them in 'Virtual DOS Machines', which provides the MSDOS applications with an environment that makes the DOS applications think they own the machine by themselves. NT 3.1 also supports running Windows 3.x WIN16 application by providing an execution environment called WOW (Windows On Windows). The WOW is basically a single 4GB process space in which all 16-bit applications execute.

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA
Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

Windows NT 3.1 has the same UI as Windows 3.1. Windows NT 3.5x was basically feature updates to NT 3.1.

Windows 95

Introduced in 1995. The major news is that Windows 95 now supports the WIN32 environment introduced in NT 3.1, which allows preemptive multitasking of Windows applications. With the WIN32 model comes a protected memory model in which each application has a virtual 4GB process space. Since this needs an Intel 80386 or newer CPU, Windows 95 no longer supports the older REAL or Standard execution modes. Windows 95 also eliminates the previously described limitations of needing the MSDOS and BIOS of the machine for disk and file I/O that it now handles in new 32-bit VxD device drivers. Windows 95 also comes with full networking support built into the OS.

Windows 95 uses the same VxD device driver model as was introduced in Windows 3.0. Some of the newer device drivers have a C-callable interface in that it passes parameters on the execution stack instead of in CPU registers. New device drivers for 3rd party hardware devices can therefore be written in either C or assembly, which eases the development task considerably.

One of the big introductions in Windows 95 is plug-and-play, which automates configuration of hardware devices. This however complicates the device driver model compared to the older Windows 3.1 model; Windows 3.1 VxD drivers can still be used though.

Later version of Windows 95 (the so-called OSR2 version or OEM Service Release 2 version) brings limited USB support to Windows 95. This is not very reliable and will not be mature until Windows 98 is released.

Microsoft introduces DirectX in Windows 95. DirectX is an umbrella of several APIs that attempts to attract game developers to develop high performing games running on the Windows 95 platform instead of running the games directly in DOS. The DirectX suite includes DirectDraw for high performance 2D graphics, DirectSound for low-latency sound playback and hardware accelerated sound mixing, DirectInput for joystick and force-feedback devices, Direct3D for high performance 3D graphics etc. By using the DirectX technologies instead of writing the games to talk directly to the hardware like previously done, game developers didn't have to worry about supporting all flavors of graphics cards, sound cards and joysticks in the world, the differences between the hardware is abstracted away by a thin Hardware Abstraction Layer (HAL) that the hardware vendors instead implements for the devices.

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA

Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

Windows NT 4.0

Introduced in 1997. Basically NT 3.x with the new Windows 95 UI. Although much internal code has been added or changed, not much changes for the application or systems programmer. NT 4.0 also supports DirectX (needs an add-on released after the OS itself), which enables DirectX-enabled games written for Windows 95 to be executed on Windows NT.

NT 4.0 has the same NT-style driver model that was introduced in NT 3.1. Although NT 4.0 looks very much like Windows 95 it does not support plug-and play.

Windows 98

Windows 98 was introduced in 1998. Although looking similar to Windows 95 on the outside, it had an interesting addition internally. Windows 98 is namely the first Windows platform that supports the new WDM driver model (Windows Driver Model). WDM drivers are based on the NT-style drivers with plug-and-play and power management added. The goal of WDM drivers is that the two driver models that previously existed (VxD and NT-style drivers) will be replaced by the WDM driver model.

Windows 98 still supports the older VxD drivers that date back to Windows 3.0. Internally Windows 98 still has the same architecture as Windows 95 and therefore device driver writers have the option of writing new drivers as VxD drivers or as WDM drivers. The WDM support in the original version of Windows 98 is rather limited and few devices therefore used WDM drivers. USB drivers have to be written as WDM drivers though.

Windows 98 Second Edition/Millennium Edition

These Windows 98 versions are updates to the original version and mainly consist of driver updates. These updates also bring more applications included in the box.

Windows 98 SE and ME also have better WDM support. The WDM driver model enables source code compatibility between the Windows 98 and 2000/XP platforms although the two operating systems are completely different internally. Note that a WDM driver is not guaranteed to be binary compatible between the two platforms since the system services provided by the two platforms have slightly different call interfaces. By

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA
Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

linking with different libraries when building for the Windows 98 and Windows 2000/XP platforms, the same driver source can be used for building drivers for both platforms. The Windows 98 SE/ME VxD driver model is the same as for the original Windows 98 version. Since the system comes with better driver support, WDM drivers may now replace some older VxD drivers.

Windows 2000

Windows 2000 is built from NT technologies and, because of this; it only supports WDM drivers and NT-style drivers. Devices that have WDM device drivers should support plug-and-play (or else the drivers will not be certified by Microsoft). Windows 2000 is also the first NT-based platform to support USB devices. Like NT 4.0, Windows 2000 supports DirectX, now out of the box.

Windows XP (and newer)

This is Windows NT version 6. It's built on the same technology as Windows 2000 and therefore none of the legacy VxD drivers are supported.

Windows 2000/XP does support the older NT-style drivers but since Microsoft will not certify these drivers because they do not support plug-and-play and power management they are not likely to be created by the hardware vendors. Since the driver installation scripts for Windows 2000/XP are different from the NT 4.0 drivers, the user cannot accidentally install an NT 4.0 driver on a Windows 2000/XP platform. What this means, in practice, is that only WDM drivers are used on the Windows 2000/XP and newer platforms.

VxD driver overview (Windows 9x only)

As previously mentioned a VxD device driver is a 32-bit device driver running in kernel mode and that has an assembly or C callable interface. A VxD with an assembly callable interface take arguments passed in CPU registers and return any return value in a CPU register. A C-callable VxD driver takes arguments passed on the stack and returns a return value in a CPU register.

A VxD driver exports a structure called a Device Description Block (DDB) that defines the functions that can be called in the driver. The DDB also provides the caller (the system or another driver) with information about the driver itself. All functions callable

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA
Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

in the VxD driver are routed through the same entry point in the driver; the exact function being executed is determined by a function code passed as an argument to this entry point.

VxD drivers are traditionally written in assembly language since this was the call interface to the MSDOS and BIOS system services in Windows 3.x. Since these services are now replaced by equivalent VxD services in Windows 9x, more and more VxD services are implemented as C-callable services (takes arguments on the stack) that eases the development of new VxD device drivers.

A VxD can be statically or dynamically loaded in Windows 9x. Windows 3.x only supports statically loadable VxDs (VxD loaded at boot time). The dynamically loadable VxD feature was introduced because of plug-and-play in Windows 95. A statically loadable VxD is loaded when the system starts while a dynamically loadable VxD is loaded on demand (plug-and-play). A statically loadable VxD is loaded because of a 'device=' statement in the system.ini file or because of a similar entry in the system registry.

A VxD can choose to implement dispatch routines to handle a number of system defined messages as well as application defined messages. System defined messages are for instance initialization messages as well as shutdown messages. Application defined messages are messages defined by the VxD writer; these messages are sent from a user mode application running on Windows 9x by using the WIN32 DeviceIoControl function. The application passes parameters to the DeviceIoControl function that will be sent to the VxD driver packaged in a structure.

NT-Style drivers (NT 3.x, NT 4.0)

As previously explained, this driver model was introduced in Windows NT 3.1. NT-style drivers are layered; a driver layered on top of another driver uses the services of the lower driver; all data sent to the lower-level device driver will first be passed to the upper filter driver. NT-style drivers have a C call interface; parameters to the driver are passed in a structure called an I/O Request Package (IRP). A driver processes the information contained in the IRP and passes the IRP on to any lower lever drivers. The system initially creates an IRP when, for instance, an application initiates a read or write to the device. The I/O data is attached to the IRP and the IRP is eventually sent to the device driver that sends the data associated with the IRP to the device.

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA

Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

An NT-style driver, like a VxD driver, has a number of pre-defined dispatch routines. While the VxD exports its callable routines through its Device Description Block structure an NT-style driver registers its callable functions by registering them with the system when the device driver is initialized just after being loaded. Some of these dispatch routines have to do with opening and closing the driver, writing and reading data and so on. Application-defined entry points (IOCTLs) can be handled through the special DriverDeviceIoControl dispatch routine implemented in the driver. This is being called by the system when a user mode application calls the WIN32 function DeviceIoControl.

NT-style drivers are loaded because of an entry in the system registry that specifies that the driver should be loaded at system startup or because it's being dynamically loaded. An NT-style driver doesn't support plug-and-play but rather has to scan the system buses to find instances of its device.

WDM drivers (Windows 98, 2000, XP, Vista, 7/8/10)

This is a driver model that is common to the Windows 98, 2000, XP and newer platforms. A WDM driver is very similar to an NT-style driver with the difference that a WDM driver supports plug-and-play and power management and therefore must receive and handle a handful new system IRPs. These messages are sent when a physical device is added, removed or when the power state of the machine is changed (as when the machine is suspended). Apart from plug-and-play and power management, the architecture of a WDM driver is very similar to the older NT-style driver.

With regards to device discovery, the programming model is slightly different from the older NT-style driver model. In WDM, the operating system (via special bus-specific drivers) will scan the buses for all attached devices, allocate device resources (memory, I/O and interrupts) and finally hand these device resources to the device driver. The device driver will then initialize its device and make it ready for operation. Once the device is removed (unplugged) or when the system is shut down, the device driver will be notified by the operating system such that resources can be cleaned up (reverse operation from device initialization).

WDF Drivers (Windows 2000 and newer)

In 2004, Microsoft introduced a new device driver programming model called Windows Driver Foundation (WDF). Later WDF was renamed Windows Driver Frameworks. The plural in the name is because there are two types of frameworks; the Kernel Mode Driver

26895 Aliso Creek Rd. Suite B504
Aliso Viejo, CA 92656-5301, USA

Phone: +1 877-839-2543, Fax: +1 877-349-1818
E-mail John@summitsoftconsulting.com

Framework (or KMDF) and User Mode Driver Framework (or UMDF). A KMDF driver uses the Microsoft-provided Kernel Mode Driver Framework as a wrapper to simplify the implementation compared to a WDM driver. The KMDF does this by 'wrapping' the user-written driver similar to how a MFC C++ class framework shields an application from having to interact directly with the C-based WIN32 API.

A User Mode Driver Framework – based driver is implemented (originally) as a C++ COM-based object interface (version 1) but later re-implemented with a simpler C interface (version 2). Not many people understand COM well and, likely, developers were hindered more by the COM interface details than they were helped by the UMDF framework.

The advantage with writing a UMDF driver is that no expertise is needed to implement client drivers for certain types of devices such as custom USB devices. Having that said, there are now alternative methods for writing USB client drivers (WinUSB, for instance) that are more straight-forward to use.

WDF shines best when writing kernel-mode drivers since the KMDF implements a default handling of the very complex plug-and-play and power-management state machines required by a WDM driver. Only the parts related to the hardware device needs to be implemented by the driver writer; much of the complexity with interacting with the I/O Manager has been greatly simplified. This 'wrapper' functionality provided by the KMDF, again, resembles prior Microsoft frameworks such as MFC.

Note that a WDF-based driver actually implements a WDM driver since the KMDF exposes its functionality to the I/O manager just like a standard WDM driver. This means that the I/O manager, operating system and the user-mode application using the WDF driver will be able to access the driver via standard WIN32 calls such as CreateFile, WriteFile and ReadFile.

Planned Future Article and Video subjects

In future videos and articles, I will in more detail explain the mechanics of developing WDM and WDF drivers for modern Windows platforms (XP/Vista/7/8/10). These videos will be in the form of excerpts from various books so that you can buy the same books and study the finer details.