

Inside the Boot Process

Mark Russinovich

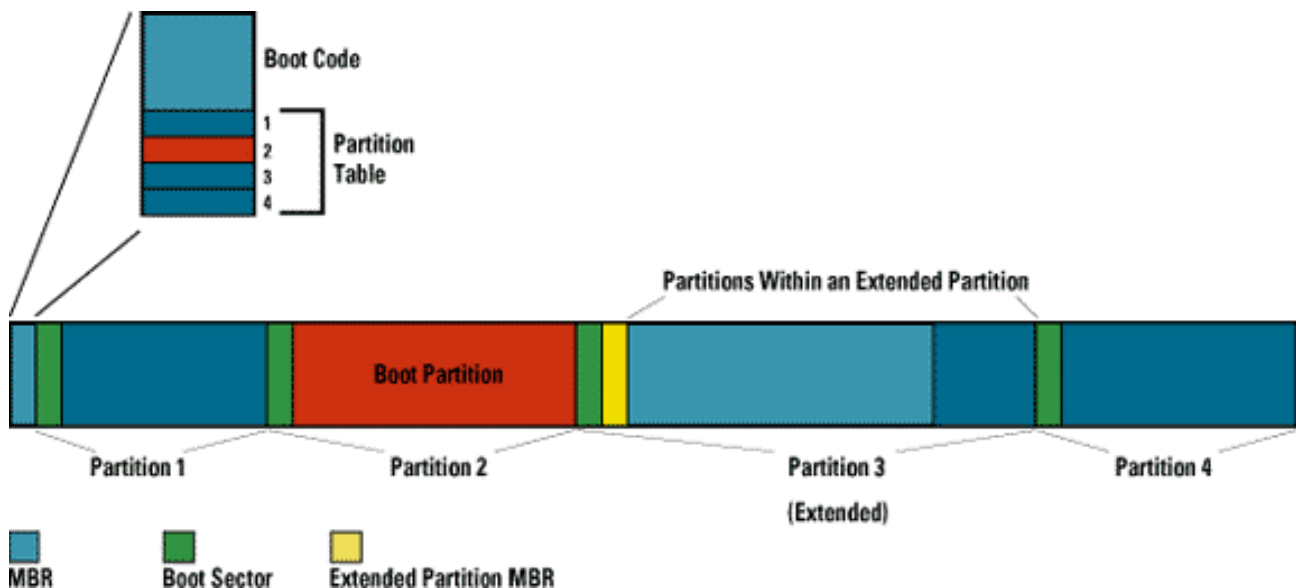
(Reprinted from WindowsItPro Magazine)

The boot process is the first of three major phases Windows NT goes through during one cycle of normal operation. The other phases are normal operation and shutdown. Although this column and other articles in Windows NT Magazine have extensively covered the details of NT's policies and mechanisms during normal operation, the boot process is not usually covered in depth (for a selection of these articles, see "Related Articles in Windows NT Magazine"). Even Microsoft's literature on the boot process, including Microsoft Windows NT Server 4.0 Resource Kit and Microsoft Windows NT Device Driver Kit, glosses over the details and presents a somewhat inaccurate picture of the steps NT goes through during a boot.

This month, I will begin a detailed tour through the NT boot process. I'll start at the point at which you install NT and proceed through the way NTLDR and NTDETECT execute. Device drivers are a crucial part of the boot process, so I'll present the way they control where in the boot they load and initialize. Next time, I'll describe how the Executive subsystems initialize, and then I'll show you how the Kernel launches the user-mode portion of NT by starting the Session Manager process, the Win32 subsystem, and the logon process. Along the way, I'll highlight the points at which various text appears on the screen to help you correlate the internal process with what you see when you watch NT boot.

Preboot

The NT boot process doesn't begin when you power on your computer or press the reset key. It begins when you install NT on your computer. At some point during the execution of NT's Setup program, NT prepares the system's primary hard disk with code that takes part in the boot process. Before I discuss what this code does, let me explain how and where NT places it on a disk. Since the early days of DOS, a standard has existed on x86 systems for the way physical hard disks are divided into logical disks. Microsoft operating systems (OSs) split hard disks into discrete areas known as partitions and use file systems (e.g., FAT, NTFS) to format each partition to be a logical drive. A hard disk can contain up to four primary partitions. Because this apportioning scheme would otherwise limit a disk to four logical drives, a special partition type, called an extended partition, further allocates up to four additional partitions within the primary partitions. Extended partitions can include extended partitions, which can contain extended partitions, and so on, making the number of drives an OS can place on a disk effectively infinite. Figure 1, page 60, gives an example of a hard disk layout.



Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Physical disks are addressed in units known as sectors. A hard-disk sector on an IBM-compatible PC is 512 bytes (sectors can be 1024 bytes on Alpha systems). Utilities that prepare hard disks for the definition of logical drives, including the DOS FDISK utility or the NT Setup program, write a sector of data called a Master Boot Record (MBR) to the first sector on a hard disk. The MBR includes a fixed amount of space that contains executable instructions and a table with four entries that define the locations of the primary partitions on the disk. When IBM-compatible computers boot, the first code they execute is called the BIOS, which is encoded into the computers' ROM. The BIOS reads the MBR off the first sector of a hard disk into memory and transfers control to the code in the MBR.

The MBRs that Microsoft partitioning tools write go through a similar process. First, an MBR's code scans through the primary partition table and locates the first partition with a flag that signals the partition is bootable. When the MBR finds at least one such flag, it reads the first sector from the flagged partition into memory and transfers control to code within the partition. The first sector of such a defined partition is called a boot sector.

OSs generally write boot sectors to disks without a user's involvement. For example, when NT Setup writes the MBR to a hard disk, it also writes a boot record to the first bootable partition of the disk. If you're familiar with the DOS SYS command, you've probably used it to manually write DOS boot sectors to disks. NT Setup checks to see whether the boot sector it will overwrite with an NT boot sector is a valid DOS boot sector. If the boot sector is a valid DOS boot sector, NT Setup copies the boot sector's contents to a file named bootsect.dos in the root directory of the partition. I'll discuss the role bootsect.dos plays in dual-boot environments shortly.

Before writing a partition's boot sector, NT Setup ensures that the partition is formatted with a file system that NT supports, such as FAT and NTFS (NT 5.0 will also support FAT32). NT Setup formats the boot partition--and any other partition with a file system type you specify. If partitions are already formatted, you can instruct Setup to leave them alone. After Setup formats the boot partition, setup copies the files NT uses to the logical drive, including two files that are part of the boot sequence, NTLDR and ntdetect.com.

Another of Setup's roles is to create a boot menu file, boot.ini, in the root directory of the boot partition. This file contains options for starting the version of NT that Setup installs and any preexisting NT installations. If bootsect.dos contains a valid DOS boot sector, one of the entries boot.ini creates is to boot into DOS. Listing 1 shows an example boot.ini file from a dual-boot computer on which DOS is installed before NT.

LISTING 1: Example boot.ini File for a Dual-Boot Computer

```
[boot loader]

timeout=30

default=multi(0)disk(0)rdisk(0)partition(1)\WINNT

[operating systems]

multi(0)disk(0)rdisk(0)partition(1)\WINNT=
"Windows NT Workstation Version 4.00"

multi(0)disk(0)rdisk(0)partition(1)\WINNT=
"Windows NT Workstation Version 4.00 [VGA mode]" /basevideo /sos
```

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

```
C:\="Microsoft Windows"
```

The Boot Sector and NTLDR

Setup must know the partition format before it writes a boot sector, because the contents of the boot sector depend on the format. For example, if the boot partition is a FAT partition, NT writes code to the boot sector that understands the FAT file system. But if the partition is in NTFS format, NT writes NTFS-capable code. The role of the boot-sector code is to give NT information about the structure and format of a logical drive, and to read in the NTLDR file from the root directory of the logical drive. Thus, the boot-sector code contains just enough read-only file system code to accomplish this task. After the boot sector code loads NTLDR into memory, it transfers control to NTLDR's entry point. If the boot sector code can't find NTLDR in the boot partition's root directory, the boot sector code prints the error message BOOT: Couldn't find NTLDR if the boot file system is FAT, or A kernel file is missing from the disk if the file system is NTFS.

NTLDR begins its existence while a system is executing in an x86-operating mode called Real-Mode. In Real-Mode, no virtual to physical translation of memory addresses occurs, which means that programs that use the addresses interpret them as physical addresses, and only the first 640KB of the computer's physical memory is accessible. Simple DOS programs execute in a Real-Mode environment. However, the first action NTLDR takes is to switch the system to Protected-Mode. Still no virtual-to-physical translation occurs at this point in the boot process, but a full 32 bits of memory becomes accessible. After it is in Protected-Mode, NTLDR can access all of physical memory. After creating enough page tables to make memory below 1MB accessible with paging turned on, NTLDR enables paging. Protected-Mode with paging enabled is the mode in which NT executes in normal operation.

After NTLDR enables paging, NTLDR is fully operational. It next reads the boot.ini file from the root directory using built-in file-system code. Like the boot sector's code, NTLDR contains read-only NTFS and FAT code; unlike the boot sector's code, NTLDR's file-system code can read subdirectories. NTLDR clears the screen and presents the user with the boot selection menu. If the user doesn't select an entry within the timeout period the boot.ini file specifies, NTLDR chooses a default selection. A boot.ini line directs NTLDR to the partition on which the NT system directory (<winnt>) of the selected installation resides. This partition might be the same as the boot partition, or it might be another primary partition.

If the boot.ini line refers to a DOS installation (i.e., by referring to C:\ as the system partition), NTLDR reads the contents of the bootsect.dos file into memory and executes a warm reboot. This action causes the code to execute after the computer reinitializes as if the MBR had read the code from disk. Code in bootsect.dos continues a DOS-specific boot, such as one into Windows 98 or Win95 on a computer on which these OSs are installed with NT.

A line in boot.ini can include option arguments that NTLDR and other components involved in the boot process interpret. Table 1, page 62, shows a complete list of these options and their effects. If the /MAXMEM= argument is present in the boot selection a system's user chooses, NTLDR discards the amount of memory specified in the internal tables it uses to keep track of physical memory allocations. Options such as /MAXMEM= are useful for device-driver developers who want to stress their drivers under various operating conditions.

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

TABLE 1: Boot.ini Options

Option	Effect				
/MAXMEM=	Limits NT to using only the amount of memory (MB) you specify (e.g., /MAXMEM=16 limits NT to using 16MB of the system's memory).				
/BURNMEMORY=	Causes NT to discard as unusable the amount of memory (MB) you specify; limits memory as /MAXMEM does (e.g., /BURNMEMORY=128 causes NT to discard 128MB of the physical memory on the machine as unusable).				
/ONECPU	Causes NT to enable only one CPU of a multiprocessor system.				
/NUMPROC=	Enables only the number of CPUs you specify (e.g., /NUMPROC=2 on a 4-way system causes NT not to use two of the four processors).				
/SOS	Causes NT to print information about what drivers load as the system boots.				
/BASEVIDEO	Causes NT to use the standard VGA display driver when moving to GUI mode.				
/NODEBUG	Prevents kernel-mode debugging from initializing. Overrides the specification of any of the three debug-related switches: /DEBUG, /DEBUGPORT, and /BAUDRATE.				
/CRASHDEBUG	Has the same effect as /NODEBUG, although its name implies otherwise.				
/DEBUG	Enables kernel-mode debugging.				
/DEBUGPORT=	Enables kernel-mode debugging and specifies an override for the default serial port (COM1) a remote debuggee connects to (e.g., /DEBUGPORT=COM2).				
/BAUDRATE=	Enables kernel-mode debugging and specifies an override for the default baud rate (19200) a remote debuggee will connect at (e.g., /BAUDRATE=115200).				
/KERNEL=	Specifies overrides of NTLDR's selection of ntoskrnl.exe in the system root (<winnt>\system32) as the Kernel's				
/HAL=	image file and of hal.dll as the HAL image file. Both options are useful for alternating between a checked Kernel environment and a free Kernel environment. If you want to boot into a checked environment that consists solely of the checked Kernel and HAL (typically all you need to test drivers), follow these steps on a system installed with the free build (retail NT):				
	<table border="1" style="width: 100%;"> <tr> <td style="width: 5%; text-align: center;">1.</td> <td>Copy the checked version of the Kernel from the checked build distribution CD-ROM to your <winnt>\system32 directory, naming it ntoschk.exe. If you are on a uniprocessor, copy ntoskrnl.exe; otherwise, copy ntkrnlmp.exe.</td> </tr> <tr> <td style="text-align: center;">2.</td> <td>Copy the checked version of the HAL from the checked build distribution CD-ROM to your <winnt>\system32 directory, naming it halchk.dll. To determine which HAL to copy, go into your <winnt>\repair directory and</td> </tr> </table>	1.	Copy the checked version of the Kernel from the checked build distribution CD-ROM to your <winnt>\system32 directory, naming it ntoschk.exe. If you are on a uniprocessor, copy ntoskrnl.exe; otherwise, copy ntkrnlmp.exe.	2.	Copy the checked version of the HAL from the checked build distribution CD-ROM to your <winnt>\system32 directory, naming it halchk.dll. To determine which HAL to copy, go into your <winnt>\repair directory and
1.	Copy the checked version of the Kernel from the checked build distribution CD-ROM to your <winnt>\system32 directory, naming it ntoschk.exe. If you are on a uniprocessor, copy ntoskrnl.exe; otherwise, copy ntkrnlmp.exe.				
2.	Copy the checked version of the HAL from the checked build distribution CD-ROM to your <winnt>\system32 directory, naming it halchk.dll. To determine which HAL to copy, go into your <winnt>\repair directory and				

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

		open setup.log in Notepad. Search for hal.dll, and you'll find a line similar to "\WINNT\system32\hal.dll = "halmps.dll","1a01c". The name to the right of the equal sign is the name of the HAL you need to copy.
	3.	Copy the default line in the system's boot.ini file.
	4.	In the string description of the boot selection, add something to identify that the new selection will be for a checked build environment (e.g., "Windows NT Server Version 4.0 CHECKED").
	5.	Add the following to the end of the new selection's line:
		/KERNEL=NTOSKCHK.EXE /HAL=HALCHK.DLL
Now you can select the new line to boot into a checked environment or select the pre-existing selection to boot into the free build.		
/3GB		Causes the split between the user and system portions of NT's virtual address map to move from 2GB (2GB user, 2GB system) to 3GB (3GB user, 1GB system). This switch made its debut in NT 4.0 Service Pack 3 (SP3), and NT supports the switch in all subsequent releases. Giving virtual memory-intensive applications such as database servers a larger address space can improve their performance. However, for an application to take advantage of this feature, two conditions must hold: The system must be part of the NT Enterprise suite (SP3 is not), and the application must be flagged as 3GB-aware.
/PCILOCK		Stops NT from dynamically assigning I/O and IRQ resources to PCI devices and leaves the devices configured by the BIOS.
/NOSERIALMICE=		Disables serial mouse detection of the specified COM ports. Use this switch if you have a component other than a
[COMx COMx,y,z, ...]		mouse attached to a serial port during the startup sequence. If you use /NOSERIALMICE without specifying a COM port, NT disables serial mouse detection on all COM ports.

NTLDR next loads and executes the nt detect.com program, which prints a message to the screen such as NTDETECT V4.0 Checking Hardware. Nt detect.com is a standard DOS-style program that uses a system's BIOS to query the computer for basic device and configuration information. This information includes the time and date information stored in the system's CMOS (nonvolatile memory); the types of buses (e.g., ISA, PCI, EISA, Micro Channel Architecture--MCA) on the system and identifiers for devices attached to the buses; the number, size, and type of disk drives on the system; the types of mouse input devices connected to the system; and the number and type of parallel ports configured on the system. This information is gathered into internal data structures that will become the HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION Registry key later in the boot. NTLDR clears the screen and prints its version information: OSLOADER V4.0. The first two files that NTLDR loads make up NT's core: ntoskrnl.exe and hal.dll. Both files are located in the <winnt>\system32 directory. Ntoskrnl.exe contains the Kernel and Executive subsystems (e.g., Memory Manager, Cache Manager, Object Manager), and hal.dll contains code that interfaces NT to the computer hardware. Hardware abstraction layers (HALs) can provide interfaces to proprietary hardware, so Microsoft makes it possible for OEMs to supply custom HAL files. If NTLDR fails to load

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

either of these files, it prints the message Windows NT could not start because the following file was missing or corrupt, followed by the name of the file.

Now the user has the option to select the Last Known Good configuration by pressing the spacebar. After a successful boot completes, NT makes a copy of the Registry tree that contains static and dynamic system and driver configuration information, HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet, and marks it as the Last Known Good configuration. Because device drivers load based on information under the Services Registry key, this key is especially crucial. If a driver installs and at the next reboot prevents the system from starting, when the user selects the Last Known Good configuration, NT reverts to using the copy of the Registry tree that existed before the driver installed. Because the good copy does not include commands to load the driver, a boot using that copy will likely succeed.

After the user selects the Last Known Good configuration or a timeout of a few seconds expires, NTLDR proceeds to read in the HKEY_LOCAL_MACHINE\SYSTEM Registry hive (a hive is a file that contains a Registry subtree) that is located at <winnt>\system32\config\system. NTLDR splices the hardware configuration information it gathered earlier into the in-memory image of the SYSTEM Registry key, then NTLDR begins to load other files necessary for the boot.

The next files NTLDR loads are device drivers that function as boot drivers. Every device driver has a Registry subkey under HKEY_LOCAL_MACHINE\SYSTEM\Services. For example, NTFS has a subkey named Ntfs, which Screen 1 shows. A driver's subkey contains several values that NT examines when it determines when to load the driver. The value with the most general effect on the driver's load time is that driver's Start value. A Start value can have one of five possible settings, which Table 2, page 66, shows. NTLDR scans the in-memory Registry image and locates all drivers that have Start values of SERVICE_BOOT_START (Boot Start) and loads them into memory. NTLDR also loads the file system driver responsible for implementing the code for the file system's type of partition, on which the installation directory resides (FAT or NTFS). NTLDR must load these drivers at this time because otherwise the Kernel would require the drivers to load themselves, a requirement that introduces a circular dependency. Boot drivers include the disk drivers for the system partition. NTLDR prints a period to the screen for every file it loads, unless the /SOS switch is specified in the boot.ini selection, in which case NTLDR displays the filenames. Note that the drivers are loaded but not initialized at this time—they initialize later in the boot sequence.

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

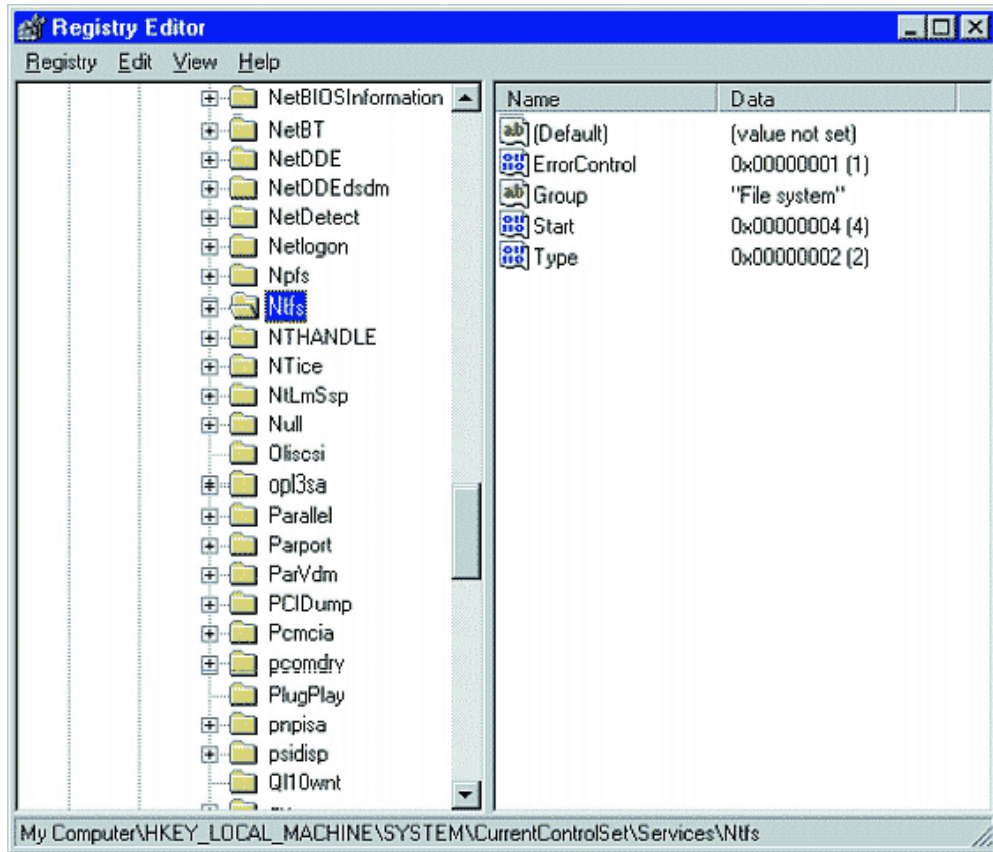


TABLE 2: Device Driver Registry Parameter Definitions That Affect Initialization Order

Value Name	Value Setting	Value Setting Description
Start	SERVICE_BOOT_START (0)	NTLDR or OSLOADER preloads driver so that it is in memory during the boot. Driver
		initializes just before SERVICE_SYSTEM_START drivers.
	SERVICE_SYSTEM_START(1)	Driver loads and initializes after SERVICE_BOOT_START drivers have initialized.
	SERVICE_AUTO_START (2)	Services subsystem loads and initializes the driver or service.
	SERVICE_DEMAND_START (3)	Services subsystem starts the driver on demand.
	SERVICE_DISABLED (4)	Driver or service does not load or initialize.
ErrorControl	IGNORE (0)	I/O Manager ignores errors the driver returns. No warning logs or displays.

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

	NORMAL (1)	If the driver reports an error, a warning displays.
	SEVERE (2)	If the driver returns an error and user didn't choose LastKnownGood, reboot into
		LastKnownGood. Otherwise, continue the boot.
	CRITICAL (3)	If the driver returns an error and user didn't choose LastKnownGood, reboot into
		LastKnownGood. Otherwise, stop the boot with a blue screen crash.
Group	Group Name	Driver initializes when its group initializes.
Tag	Tag Number	Specifies location in group initialization order.
DependOnGroup	Group Name	Driver won't load unless a driver from the group specified loads.
DependOnService	Service Name	Driver or service won't load unless the specified driver or service loads.

This action is the end of NTLDR's role in the boot. After preparing processor registers for the Kernel, NTLDR locates the main function in `ntoskrnl.exe`'s in-memory image and transfers control to `ntoskrnl.exe`. The sole parameter NTLDR passes to the Kernel is a data structure relating to the information NTLDR gathered during its execution. This data structure contains a copy of the line in `boot.ini` NTLDR selected, a pointer to the memory tables NTLDR generated to describe the physical memory on the system, a pointer to the in-memory `HKEY_LOCAL_MACHINE` Registry tree (which contains the `HARDWARE` and `SYSTEM` subkeys), and a pointer to the list of boot drivers NTLDR loaded.

The process I've described to this point pertains to x86 systems and is only partially applicable to Alpha systems. On an Alpha, the MBR does not contain boot codes. Instead, the Alpha firmware plays an active role in the boot. (Advanced RISC Consortium—ARC—defined functionality in the firmware of RISC computers.) NT Setup programs Alpha firmware to contain the boot menu entries that `boot.ini` contains on x86 systems. The programmed firmware code understands how to read FAT drives, so that when a user makes a boot selection, the firmware reads the boot partition to load `osloader.exe`, Alpha's equivalent of NTLDR. The firmware also loads the HAL from the boot partition, rather than from the NT installation directory (where the HAL resides on x86 systems), and reads in a PAL file. A PAL file contains code that NT uses to interface to a specific Alpha machine's microcode. The Alpha firmware doesn't understand NTFS drives, which is why you must create a minimal FAT partition on Alphas to contain `osloader.exe` and the HAL and PAL code. OSLOADER performs the same steps on Alphas as NTLDR performs on x86 systems, but OSLOADER has no `ntdetect.com`. Instead, the Alpha firmware's code performs detection and hands the information to OSLOADER after it loads.

Stay Tuned

Next time, I'll pick up where I've left off and discuss the steps `ntoskrnl.exe` goes through to initialize various Executive subsystems, such as the Memory Manager, Object Manager, and Process

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Manager. I'll also describe how NTOSKRNL loads more device drivers and initializes them and the boot drivers that NTLDR loaded. I'll conclude this tour of the NT boot process by showing you how ntoskrnl.exe initializes the user-mode side of the boot by loading winlogon, CSRSS (the Win32 subsystem), and services.

In November, I began this two-part series on the Windows NT boot process. I described the way NT's Setup program prepares a hard disk for a boot by placing NT code in the disk's Master Boot Record (MBR) and boot sectors. When the system boots, the NT code loads the NTLDR boot file and executes the file's code. NTLDR is responsible for the pre-NT Kernel portion of the boot; part of NTLDR's responsibility is to prepare a data structure with information about the system and to construct the HKEY_LOCAL_MACHINE\HARDWARE and HKEY_LOCAL_MACHINE\SYSTEM Registry keys. NTLDR later gives this information to the Kernel, together with images for hal.dll and the boot-start device drivers.

This month, I resume my description of the boot process at the point at which NTLDR has finished its role in the boot by calling the NT Kernel's entry point. I'll walk you through the rest of the boot process, including the initialization steps each Executive subsystem takes. I'll describe how and when device drivers from each of the start categories--Boot, System, and Auto--initialize. I'll conclude by describing how the Kernel finishes a boot by loading user-mode code that presents the logon screen display and launches the Win32 subsystem. The sidebar "Windows 2000 and the Boot," page 60, explains how the NT boot process will change in Windows 2000 (Win2K--formerly NT 5.0).

Initializing the Kernel and Executive Subsystems

The NT Kernel goes through two phases in its boot process: phase 0 and phase 1. Phase 0 initializes just enough of the Kernel and Executive subsystems so that basic services required for the completion of initialization become operational in phase 1. NT keeps interrupts disabled during phase 0 and enables them before phase 1. Most Executive subsystems implement their initialization code by having one function take a parameter that identifies which phase is executing.

The function responsible for orchestrating phase 0 is called `ExpInitializeExecutive`. This function starts by calling the hardware abstraction layer (HAL) function `HallnitSystem`. `HallnitSystem` gives proprietary versions of the HAL that various OEMs develop a chance to gain system control before NT performs significant initialization. One `HallnitSystem` responsibility is to prepare the system interrupt controller for interrupts and to configure the clock tick interrupt. When `HallnitSystem` returns control, `ExpInitializeExecutive` proceeds by honoring the `/BURNMEMORY BOOT.INI` switch (if the switch is present in the selection the user made in the boot.ini file for which installation to boot) and discarding the amount of memory the switch specifies. Next, `ExpInitializeExecutive` calls initialization routines for the Memory Manager, Object Manager, Security Reference Monitor, and Process Manager. The Memory Manager gets the ball rolling by constructing page tables and internal data structures that are necessary to provide basic memory services. The Memory Manager builds and reserves an area for the system file cache and creates memory areas for the paged and nonpaged pools. The other Executive subsystems, the Kernel, and the device drivers use these two memory pools for allocating their data structures.

After the Memory Manager finishes phase 0, `ExpInitializeExecutive` prints the text Microsoft (R) Windows NT (TM) Version 4.0 (Build 1381) to the initial blue screen you see during a boot. Build 1381 is the build number for NT 4.0 without service packs. For NT 4.0 with service packs applied, text similar to Service Pack 3 follows the build number on the initial blue screen. The system obtains the service pack number from `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Windows\CSDVersion`, where the

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

number is encoded as a hexadecimal value that can identify interim service packs (e.g., Service Pack 1a).

During phase 0 of Object Manager initialization, NT defines the objects that are necessary to construct the Object Manager namespace so that other subsystems can insert objects into it. NT also creates a handle table so that resource tracking can begin. The Security Reference Monitor initializes the token type object and then uses the object to create and prepare the first token for assignment to the initial process. The last Executive subsystem to initialize in phase 0 is the Process Manager. The Process Manager performs most of its initialization in phase 0, defining the process and thread object types and setting up lists to track active processes and threads. The Process Manager also creates a process object for the initial process and names it Idle. As its last step, the Process Manager creates the System process and launches a thread for this process that will direct phase 1 initialization in the Kernel's Phase1Initialization function.

When control returns to it, ExpInitializeExecutive has become a thread in the Idle process and immediately becomes the system's idle thread. ExpInitializeExecutive sets its priority to 0 (the lowest possible) and begins executing a loop that will run only if no other thread in the system can run.

Phase 1 starts when Phase1Initialization calls the HAL to prepare the system to accept interrupts from devices and to enable interrupts. Phase1Initialization notes the time and stores it as the time the system booted. Up to this point, only one CPU has been active. In a symmetric multiprocessing (SMP) system, the boot process has left other CPUs off. Now, with the aid of the HAL, Phase1Initialization turns on the remaining CPUs. NT prints text similar to 2 System Processors [128 MB Memory] Multiprocessor Kernel to the blue screen after all CPUs have initialized.

Phase1Initialization continues by calling every Executive subsystem in the order Table 1 shows, so that they can execute phase 1 initialization. The order in which Phase1Initialization calls the subsystems is important because of system interdependencies. For example, the Executive must define type objects like mutexes, semaphores, events, and timers because other subsystems use these objects.

TABLE 1: Executive Subsystem Phase 1 Initialization>	
Executive Subsystem	Initialization Function
Object Manager	Creates Object Manager namespace root directory, \ObjectTypes directory, \?? directory, and \DosDevices link to \?? directory.
Executive	Creates Executive object types, including semaphore, mutex, event, and timer.
Kernel	Initializes scheduler (dispatcher) data structures and System Service table.
Security Reference Monitor	Creates \Security directory in Object Manager namespace and initializes auditing data structures if auditing is enabled.
Memory Manager	Creates section object type, starts modified page writer thread and Balance Set Manager thread.

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Cache Manager	Initializes system file cache data structures.
Configuration Manager	Creates \Registry key object in Object Manager namespace and sucks data passed by the O/S Loader into HKEY_LOCAL_MACHINE\SYSTEM\Hardware.
Local Procedure Call Subsystem	Initializes port type object
I/O Manager	Initializes error log, creates driver and device object types. Initializes SERVICE_BOOT_START drivers and loads and initializes SERVICE_SYSTEM_START drivers.
Process Manager	Loads ntdll.dll library into the system process address space.

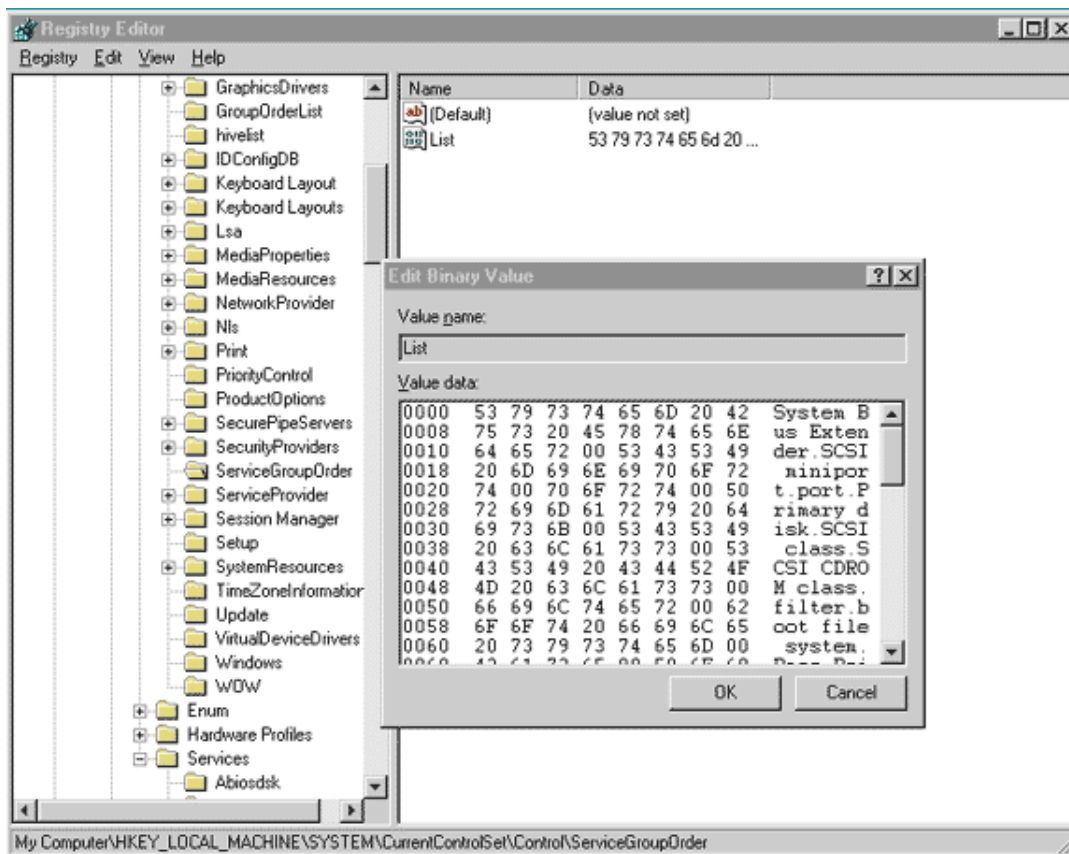
Driver Initialization

The I/O Manager's initialization is particularly interesting, because it is during phase 1 initialization that the boot drivers that NTLDR loaded finally start. When you or NT installs a device driver, the driver's Setup script or program can direct NT to define three Registry values in the driver's Registry key in addition to the Start value. These values are Group, Tag, and DependOnGroup. The first two values further refine the decision as to when the I/O Manager should initialize the driver, and the I/O Manager uses the third value when deciding whether to start a particular driver.

Before the I/O Manager initializes the boot drivers, it sorts them according to their Group values and sends drivers with no Group value to the end of the list. The Registry value HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder determines group precedence in the sorting process. Screen 1 shows ServiceGroupOrder values. Groups at the top of the list have higher precedence than groups at the bottom of the list have.

Inside the Boot Process

Mark Russinovich
(Reprinted from WindowsItPro Magazine)



After the I/O Manager sorts drivers by group, it sorts the drivers within each group according to the Tag values defined in the drivers' Registry keys. Drivers without a tag go to the end of the list in their group. You might assume that the I/O Manager initializes drivers with lower-number tags before drivers with higher-number tags, but that isn't necessarily the case. The Registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GroupOrderList` defines tag precedence within a group; with this key, Microsoft and device driver developers can take liberties with redefining the integer number system.

After the I/O Manager finishes sorting drivers, it traverses the lists it has made and initializes each driver according to its place on the list. However, if a driver has a `DependOnGroup` value, the I/O Manager will not load that driver unless it has already loaded at least one driver that belongs to the specified group. The `DependOnGroup` value makes it easy to define a driver that must not load if its operation requires another driver that isn't present on the system.

As the I/O Manager initializes each driver, it checks the return status of the driver's initialization function. If a driver reports an error, the action the I/O Manager takes depends on the `ErrorControl` value in the driver's Registry key. Table 2 shows possible `ErrorControl` values and the actions the I/O Manager takes when a driver with a particular value reports an error. In some cases, an error causes NT to reboot and try a previous version of the Registry's Control subkey in an attempt to boot successfully.

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Value Setting	Description
IGNORE (0)	I/O Manager ignores errors the driver returns. No warning logs or displays.
NORMAL (1)	If the driver reports an error, a warning displays.
SEVERE (2)	If the driver returns an error and user did not choose LastKnownGood, reboot into LastKnownGood. Otherwise, continue the boot.
CRITICAL (3)	If the driver returns an error and user did not choose LastKnownGood, reboot into LastKnownGood. Otherwise, stop the boot with a blue screen crash.

Immediately after initializing the boot drivers, the I/O Manager loads and initializes all drivers marked as `SERVICE_SYSTEM_START` (system start). The I/O Manager processes the system-start driver initialization order and dependencies in the same way it processed the order for the boot-start drivers. Thus, the only difference between boot-start drivers and system-start drivers is that NTLDR preloads the boot-start drivers and NT initializes them first. When the boot-start drivers (specifically, the file system and disk drivers responsible for the system partition) are active, the I/O Manager can use them to load the system-start drivers.

As device drivers initialize during the boot, they work in a restricted environment: NTLDR and the Configuration Manager have defined only the `HKEY_LOCAL_MACHINE\SYSTEM` and `HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION` parts of the Registry, and boot drivers cannot access files on the disk. After the I/O Manager initializes the system-start drivers, it calls the HAL to define the drive-letter mappings, in which it links drive letters to the partitions the letters reference. For example, `C:\` might be linked to `\harddisk\0\partition1`. The HAL honors any drive-letter assignments a user makes using Disk Administrator, and reads this information from `HKEY_LOCAL_MACHINE\SYSTEM\Disk`.

At the end of the `Phase1Initialization` function, the NT Kernel and Executive subsystems are fully operational. The function's last action is to launch the Session Manager Subsystem (SMSS, which is in `\winnt\system32\smss.exe`) user-mode process. SMSS is responsible for creating the user-mode environment that provides the visible interface to NT.

SMSS, CSRSS, and Winlogon

SMSS is like any other user-mode process except for two things: First, NT considers SMSS a trusted part of the OS. Second, SMSS is a native application. Because it's a trusted OS component, SMSS can perform actions few other processes can perform, such as creating security tokens. Because it's a native application, SMSS doesn't use Win32 APIs--it uses only core Executive APIs known collectively as NT's Native API. SMSS doesn't use the Win32 APIs because the Win32 subsystem isn't executing when SMSS launches. In fact, one of SMSS's tasks is to start the Win32 subsystem.

SMSS first processes commands in the Registry value

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Session Manager\BootExecute`

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Typically, this value contains one command to run the CHKDSK disk consistency checking application. As CHKDSK runs, it prints two periods on the screen for each partition it examines. SMSS creates page files after CHKDSK runs so that larger applications can begin to execute. Then, SMSS calls the Configuration Manager Executive subsystem to finish initializing the Registry, fleshing the Registry out to include all its keys. To do so, the Configuration Manager loads the Registry hives for the HKEY_LOCAL_MACHINE\SAM, HKEY_LOCAL_MACHINE\SAM\SECURITY, and HKEY_LOCAL_MACHINE\SOFTWARE keys. The Configuration Manager looks in the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Hivelist key to find out where the hives are located on disk.

SMSS then loads the win32k.sys device driver, which implements the kernel-mode portion of the Win32 subsystem. SMSS determines the location of win32k.sys and other components it loads by looking for their paths in HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Session Manager. Finally, SMSS launches csrss.exe and winlogon.exe. CSRSS is the user-mode portion of the Win32 subsystem, and Winlogon is the logon manager.

Shortly after win32k.sys starts, it switches the screen to graphics mode. A little later, Winlogon starts the Services subsystem (winnt\system32\services.exe), which loads all services and device drivers marked Auto Start. (The Services subsystem is also known as the Service Control Manager--SCM.) Auto Start drivers and services can specify a dependency on a specific service by including a DependOnService value in their Registry keys in a manner similar to the way boot drivers use the DependOnGroup value. The SCM sorts and then initializes the Auto Start drivers and services according to their group and tag values in the same way the I/O Manager sorts the boot- and system-start drivers.

After the SCM initializes the Auto Start services and drivers, it deems the boot successful. The Registry subkey HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet becomes the Last Known Good control set, because the system booted successfully to this point. Earlier in the boot, NT made a copy of this subkey and named the copy HKEY_LOCAL_MACHINE\SYSTEM\CLONE. Any changes drivers make to the current control set during the boot do not change the CLONE subkey copy. The SCM copies the CLONE subkey to another control set subkey (e.g., HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001), and the SCM marks that control set subkey as Last Known Good. The SCM marks the subkey by setting the HKEY_LOCAL_MACHINE\SYSTEM\Select\LastKnownGood value to specify the three-digit identifier at the end of the control set's subkey name (e.g., 001). If a user chooses to boot to the Last Known Good menu during the first steps of a boot, or if a driver returns a severe or critical error, the system uses the Last Known Good profile subkey as HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet. Doing so increases the chances that the system will boot successfully, because at least one previous boot using the Last Known Good profile was successful.

At approximately the time the Services subsystem is starting networking services, Winlogon presents users with the initial logon dialog box. That action brings us to the end of the boot process.

Shutdown

In contrast to the boot process, system shutdown is straightforward. First, the Win32 subsystem informs all Windows applications that the system is going down. Most applications exit voluntarily, and Winlogon stops any stragglers. Winlogon is in charge of finishing the shutdown process by calling the Executive subsystem function NtShutdownSystem. This function calls the I/O Manager, the Configuration Manager, the Memory Manager, and then the I/O Manager again, and informs them that they should prepare for the shutdown.

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

The first time NtShutdownSystem calls the I/O Manager, the I/O Manager sends shutdown I/O packets to all device drivers that have requested shutdown notification. This action gives device drivers a chance to perform any special processing their device might require before NT exits. The Configuration Manager flushes any modified Registry data to disk, and the Memory Manager writes all modified pages containing file data back to their respective files. If the option to clear the paging file at shutdown is enabled, the Memory Manager clears the paging file at this time. The second time NtShutdownSystem calls the I/O Manager, the I/O Manager informs the file system drivers that the system is shutting down. Finally, if the system's user specified a reboot after the shutdown, the system calls the HAL to reboot the computer.

We've arrived at shutdown for this month's column. This two-column series about the boot process is a roadmap with visual cues to the operations that take place behind the scenes when your system boots. Table 3 presents a summary of boot-process components with their execution modes and responsibilities. Understanding the details of the boot process helps you to diagnose problems that can arise during a boot, and gives you insight into the way NT pulls itself up by its bootstraps.

TABLE 3: Boot Process Components, Execution Modes, and Responsibilities		
Component	Processor Execution	Responsibilities
Master Boot Record (MBR) code	16-bit real mode	Reads and loads partition boot sectors.
Boot Sector	16-bit real mode	Reads FAT/NTFS root directory to load NTLDR.
NTLDR	32-bit protected mode; turns on paging	Reads boot.ini, presents boot menu and loads ntoskrnl.exe, hal.dll, and boot-start device drivers.
NTOSKRNL	32-bit protected mode with paging	Initializes Executive subsystems and boot and system-start device drivers, prepares the system for running native applications, and loads SMSS.
SMSS	32-bit NT native application	Loads Win32 subsystem, including win32k.sys and csrss.exe. Starts Winlogon process.
Winlogon	32-bit NT native application	Starts Service Control Manager (Services subsystem). Presents logon interface.
Services (SCM)	32-bit NT native application	Loads and initializes Auto Start device drivers and Win32

System initialization in Windows 2000 (Win2K--formerly Windows NT 5.0) is not significantly different from system initialization in Windows NT 4.0. Win2K introduces two new Executive subsystems: Plug and Play Manager and Power Manager. The Plug and Play Manager is integrated with the I/O Manager and doesn't have an initialization function. However, drivers initialize in such a way in Win2K as to accommodate Plug and Play (PnP)-aware drivers. The Power Manager has an initialization function, PolntSystem, that ExplnitializeExecutive calls for phase 0 and phase 1 initialization before calling the Process Manager's initialization function. PolntSystem prepares the Power Manager to implement the system power management policy, notify device drivers that the system power state is

Inside the Boot Process

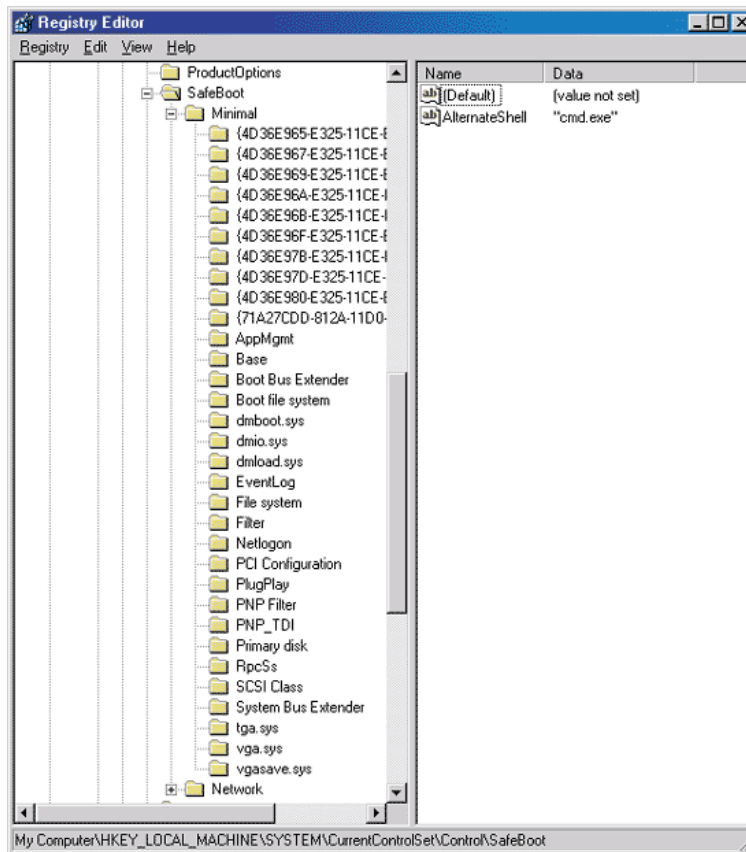
Mark Russinovich

(Reprinted from WindowsItPro Magazine)

changing, and call the hardware abstraction layer (HAL) to suspend the system or put it into hibernation mode.

A more visible change in Win2K's boot process is that the system boot menu includes an option to boot into safe mode. When you boot into safe mode, you boot into a stripped-down version of the Win2K installation. Instead of presenting Explorer as the desktop GUI, safe mode presents a command-prompt window. Only drivers and services marked as being safe load and initialize for the boot. The idea behind safe mode stems from the possibility that an errant driver, errant service, or corrupt driver file will prevent the system from booting. Safe mode increases the likelihood of booting successfully, because the system loads only the drivers and services that are necessary for the boot to succeed.

When you select a safe-mode boot, you can choose between minimal and network options. The Registry key HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SafeBoot stores safe-mode configuration settings, and the key has Network and Minimal subkeys. The Network subkey includes drivers, services, and groups necessary for network connectivity. The Minimal subkey does not contain these drivers, services, and groups. The SafeBoot key also has one value named AlternateShell, which specifies the name of the program that serves as the GUI. The value's typical setting is cmd.exe, which specifies the command-prompt program. Screen A gives an example of the partial contents of the SafeBoot Registry key.



As the I/O Manager loads drivers and services during the various safe-boot phases, it checks to see whether a subkey is under either the Minimal or the Network subkey with a name that is identical to that of the driver or service it is loading. If the I/O Manager finds such a subkey, it allows the

Inside the Boot Process

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

associated driver or service to load. If the I/O Manager does not find such a subkey, it looks for a subkey with the name of the load group that the driver or service belongs to. If the I/O Manager finds a subkey with the name of the load group, it allows the driver or service to load, because the system considers all drivers or services from a load group necessary for the safe boot. One example of a service you'll always find listed as a subkey is the Event Log service (EventLog); a load group you'll always find listed as a subkey is FileSystem.

One more visible difference between the Win2K boot and the NT 4.0 boot is that instead of presenting messages during the boot with the display in an 80 * 50 text mode, Win2K draws messages in a VGA color mode. In NT 4.0, the HAL sets the 80 * 50 mode and prints text. In Win2K, a special driver called bootvid.sys provides the boot-time display support functions.