

Startup and Shutdown

In this chapter, we'll describe the steps required to boot Windows and the options that can affect system startup. Understanding the details of the boot process will help you diagnose problems that can arise during a boot. Then we'll explain the kinds of things that can go wrong during the boot process and how to resolve them. Finally, we'll explain what occurs on an orderly system shutdown.

Boot Process

In describing the Windows boot process, we'll start with the installation of Windows and proceed through the execution of boot support files. Device drivers are a crucial part of the boot process, so we'll explain the way that they control the point in the boot process at which they load and initialize. Then we'll describe how the executive subsystems initialize and how the kernel launches the user-mode portion of Windows by starting the Session Manager process (Smss.exe), which starts the initial two sessions (session 0 and session 1). Along the way, we'll highlight the points at which various on-screen messages appear to help you correlate the internal process with what you see when you watch Windows boot.

The early phases of the boot process differ significantly on systems with a BIOS (basic input output system) versus systems with an EFI (Extensible Firmware Interface). EFI is a newer standard that does away with much of the legacy 16-bit code that BIOS systems use and allows the loading of preboot programs and drivers to support the operating system loading phase. The next sections describe the portions of the boot process specific to BIOS-based systems and are followed with a section describing the EFI-specific portions of the boot process.

To support these different firmware implementations (as well as EFI 2.0, which is known as Unified EFI, or UEFI), Windows provides a boot architecture that abstracts many of the differences away from users and developers in order to provide a consistent environment and experience regardless of the type of firmware used on the installed system.

BIOS Preboot

The Windows boot process doesn't begin when you power on your computer or press the reset button. It begins when you install Windows on your computer. At some point during the execution of the Windows Setup program, the system's primary hard disk is prepared with code that takes part in the boot process. Before we get into what this code does, let's look at how and where Windows places

the code on a disk. Since the early days of MS-DOS, a standard has existed on x86 systems for the way physical hard disks are divided into volumes.

Microsoft operating systems split hard disks into discrete areas known as *partitions* and use file systems (such as FAT and NTFS) to format each partition into a *volume*. A hard disk can contain up to four primary partitions. Because this apportioning scheme would limit a disk to four volumes, a special partition type, called an *extended partition*, further allocates up to four additional partitions within each extended partition. Extended partitions can contain extended partitions, which can contain extended partitions, and so on, making the number of volumes an operating system can place on a disk effectively infinite. Figure 13-1 shows an example of a hard disk layout, and Table 13-1 summarizes the files involved in the BIOS boot process. (You can learn more about Windows partitioning in Chapter 9, “Storage Management.”)

TABLE 13-1 BIOS Boot Process Components

Component	Processor Execution	Responsibilities	Location
Master Boot Record (MBR)	16-bit real mode	Reads and loads the volume boot record (VBR)	Per storage device
Boot sector (also called volume boot record)	16-bit real mode	Understands the file system on the partition and locates Bootmgr by name, loading it into memory	Per active (bootable) partition
Bootmgr	16-bit real mode and 32-bit without paging	Reads the Boot Configuration Database (BCD), presents boot menu, and allows execution of preboot programs such as the Memory Test application (Memtest.exe). If a 64-bit installation is booted, switches to 64-bit long mode before loading Winload.	Per system
Winload.exe	32-bit protected mode with paging, 64-bit protected mode if booting a Win64 installation	Loads Ntoskrnl.exe and its dependencies (Bootvid.dll on 32-bit systems, Hal.dll, Kdcom.dll, Ci.dll, Clfs.sys, Pshed.dll) and boot-start device drivers.	Per Windows installation
Winresume.exe	32-bit protected mode, 64-bit protected mode if resuming a Win64 installation	If resuming after a hibernation state, resumes from the hibernation file (Hiberfil.sys) instead of typical Windows loading.	Per Windows installation
Memtest.exe	32-bit protected mode	If selected from the Boot Manager, starts up and provides a graphical interface for scanning memory and detecting damaged RAM.	Per system
Ntoskrnl.exe	Protected mode with paging	Initializes executive subsystems and boot and system-start device drivers, prepares the system for running native applications, and runs Smss.exe.	Per Windows installation
Hal.dll	Protected mode with paging	Kernel-mode DLL that interfaces Ntoskrnl and drivers to the hardware. It also acts as a driver for the motherboard itself, supporting soldered components that are not otherwise managed by another driver.	Per Windows installation

Component	Processor Execution	Responsibilities	Location
Smss.exe	Native application	Initial instance starts a copy of itself to initialize each session. The session 0 instance loads the Windows subsystem driver (Win32k.sys) and starts the Windows subsystem process (Csrss.exe) and Windows initialization process (Wininit.exe). All other per-session instances start a Csrss and Winlogon process.	Per Windows installation
Wininit.exe	Windows application	Starts the service control manager (SCM), the Local Security Authority process (LSASS), and the local session manager (LSM). Initializes the rest of the registry and performs user-mode initialization tasks.	Per Windows installation
Winlogon.exe	Windows application	Coordinates logon and user security, launches LogonUI.	Per Windows installation
Logonui.exe	Windows application	Presents interactive logon dialog box.	Per Windows installation
Services.exe	Windows application	Loads and initializes auto-start device drivers and Windows services.	Per Windows installation

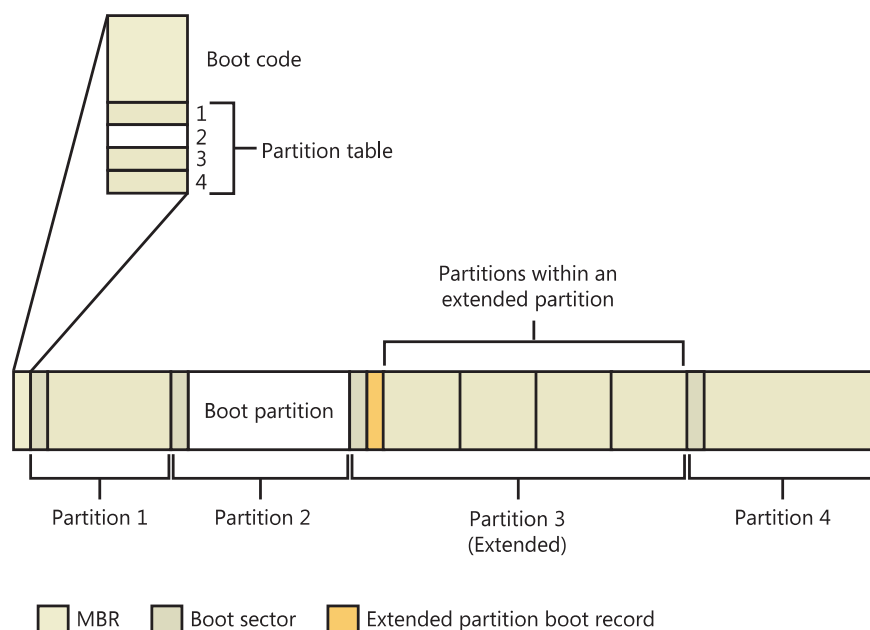


FIGURE 13-1 Sample hard disk layout

Physical disks are addressed in units known as *sectors*. A hard disk sector on a BIOS PC is typically 512 bytes (but moving to 4,096 bytes; see Chapter 9 for more information). Utilities that prepare hard disks for the definition of volumes, such as the Windows Setup program, write a sector of data called a Master Boot Record (MBR) to the first sector on a hard disk. (MBR partitioning is described in Chapter 9.) The MBR includes a fixed amount of space that contains executable instructions (called *boot code*) and a table (called a *partition table*) with four entries that define the locations of the primary

partitions on the disk. When a BIOS-based computer boots, the first code it executes is called the BIOS, which is encoded into the computer's flash memory. The BIOS selects a boot device, reads that device's MBR into memory, and transfers control to the code in the MBR.

The MBRs written by Microsoft partitioning tools, such as the one integrated into Windows Setup and the Disk Management MMC snap-in, go through a similar process of reading and transferring control. First, an MBR's code scans the primary partition table until it locates a partition containing a flag (Active) 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. This type of partition is called a *system partition*, and the first sector of such a partition is called a *boot sector* or *volume boot record* (VBR). The volume defined for this partition is called the *system volume*.

Operating systems generally write boot sectors to disk without a user's involvement. For example, when Windows Setup writes the MBR to a hard disk, it also writes the file system boot code (part of the boot sector) to a 100-MB bootable partition of the disk, marked as hidden to prevent accidental modification after the operating system has loaded. This is the system volume described earlier.

Before writing to a partition's boot sector, Windows Setup ensures that the boot partition (the *boot partition* is the partition on which Windows is installed, which is typically not the same as the system partition, where the boot files are located) is formatted with NTFS, the only supported file system that Windows can boot from when installed on a fixed disk, or formats the boot partition (and any other partition) with NTFS. Note that the format of the system partition can be any format that Windows supports (such as FAT32). If partitions are already formatted appropriately, you can instruct Setup to skip this step. After Setup formats the system partition, Setup copies the Boot Manager program (Bootmgr) that Windows uses to the system partition (the system volume).

Another of Setup's roles is to prepare the Boot Configuration Database (BCD), which on BIOS systems is stored in the \Boot\BCD file on the root directory of the system volume. This file contains options for starting the version of Windows that Setup installs and any preexisting Windows installations. If the BCD already exists, the Setup program simply adds new entries relevant to the new installation. For more information on the BCD, see Chapter 3, "System Mechanisms," in Part 1.

The BIOS Boot Sector and Bootmgr

Setup must know the partition format before it writes a boot sector because the contents of the boot sector vary depending on the format. For a partition that is in NTFS format, Windows writes NTFS-capable code. The role of the boot-sector code is to give Windows information about the structure and format of a volume and to read in the Bootmgr file from the root directory of the volume. Thus, the boot-sector code contains just enough read-only file system code to accomplish this task. After the boot-sector code loads Bootmgr into memory, it transfers control to Bootmgr's entry point. If the boot-sector code can't find Bootmgr in the volume's root directory, it displays the error message "BOOTMGR is missing".

Bootmgr is actually a concatenation of a .com file (Startup.com) and an .exe file (Bootmgr.exe), so it begins its existence while a system is executing in an x86 operating mode called *real mode*, associated

with .com files. In real mode, no virtual-to-physical translation of memory addresses occurs, which means that programs that use the memory addresses interpret them as physical addresses and that only the first 1 MB of the computer's physical memory is accessible. Simple MS-DOS programs execute in a real-mode environment. However, the first action Bootmgr 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 the system is in protected mode, Bootmgr can access all of physical memory. After creating enough page tables to make memory below 16 MB accessible with paging turned on, Bootmgr enables paging. Protected mode with paging enabled is the mode in which Windows executes in normal operation.

After Bootmgr enables protected mode, it is fully operational. However, it still relies on functions supplied by BIOS to access IDE-based system and boot disks as well as the display. Bootmgr's BIOS-interfacing functions briefly switch the processor back to real mode so that services provided by the BIOS can be executed. Bootmgr next reads the BCD file from the \Boot directory using built-in file system code. Like the boot sector's code, Bootmgr contains a lightweight NTFS file system library (Bootmgr also supports other file systems, such as FAT, El Torito CDFS, and UDFS, as well as WIM and VHD files); unlike the boot sector's code, Bootmgr's file system code can also read subdirectories.



Note Bootmgr and other boot applications can still write to preallocated files on NTFS volumes, because only the data needs to be written, instead of performing all the complex allocation work that is typically required on an NTFS volume. This is how these applications can write to bootsect.dat, for example.

Bootmgr next clears the screen. If Windows enabled the BCD setting to inform Bootmgr of a hibernation resume, this shortcuts the boot process by launching Winresume.exe, which will read the contents of the hibernation file into memory and transfer control to code in the kernel that resumes a hibernated system. That code is responsible for restarting drivers that were active when the system was shut down. Hiberfil.sys is only valid if the last computer shutdown was hibernation, since the hibernation file is invalidated after a resume, to avoid multiple resumes from the same point. (See the section "The Power Manager" in Chapter 8, "I/O System," for information on hibernation.)

If there is more than one boot-selection entry in the BCD, Bootmgr presents the user with the boot-selection menu (if there is only one entry, Bootmgr bypasses the menu and proceeds to launch Winload.exe). Selection entries in the BCD direct Bootmgr to the partition on which the Windows system directory (typically \Windows) of the selected installation resides. If Windows was upgraded from an older version, this partition might be the same as the system partition, or, on a clean install, it will always be the 100-MB hidden partition described earlier.

Entries in the BCD can include optional arguments that Bootmgr, Winload, and other components involved in the boot process interpret. Table 13-2 contains a list of these options and their effects for Bootmgr, Table 13-3 shows a list of BCD options for boot applications, and Table 13-4 shows BCD options for the Windows boot loader.

The Bcdedit.exe tool provides a convenient interface for setting a number of the switches. Some options that are included in the BCD are stored as command-line switches ("/DEBUG", for example) to

the registry value HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions; otherwise, they are stored only in the BCD binary format in the BCD hive.

TABLE 13-2 BCD Options for the Windows Boot Manager (Bootmgr)

BCD Element	Values	Meaning
<i>bcdfilepath</i>	Path	Points to the Boot Configuration Database (usually \Boot\BCD) file on the disk.
<i>displaybootmenu</i>	Boolean	Determines whether the Boot Manager shows the boot menu or picks the default entry automatically.
<i>keyringaddress</i>	Physical address	Specifies the physical address where the BitLocker key ring is located.
<i>noerrordisplay</i>	Boolean	Silences the output of errors encountered by the Boot Manager.
<i>Resume</i>	Boolean	Specifies whether or not resuming from hibernation should be attempted. This option is automatically set when Windows hibernates.
<i>Timeout</i>	Seconds	Number of seconds that the Boot Manager should wait before choosing the default entry.
<i>resumeobject</i>	GUID	Identifier for which boot application should be used to resume the system after hibernation.
<i>displayorder</i>	List	Definition of the Boot Manager's display order list.
<i>toolsdisplayorder</i>	List	Definition of the Boot Manager's tool display order list.
<i>bootsequence</i>	List	Definition of the one-time boot sequence.
<i>Default</i>	GUID	The default boot entry to launch.
<i>customactions</i>	List	Definition of custom actions to take when a specific keyboard sequence has been entered.
<i>bcddevice</i>	GUID	Device ID of where the BCD store is located.

TABLE 13-3 BCD Options for Boot Applications

BCD Element	Values	Meaning
<i>avoidlowmemory</i>	Integer	Forces physical addresses below the specified value to be avoided by the boot loader as much as possible. Sometimes required on legacy devices (such as ISA) where only memory below 16 MB is usable or visible.
<i>badmemoryaccess</i>	Boolean	Forces usage of memory pages in the Bad Page List (see Chapter 10, "Memory Management," for more information on the page lists).
<i>badmemorylist</i>	Array of page frame numbers (PFNs)	Specifies a list of physical pages on the system that are known to be bad because of faulty RAM.
<i>baudrate</i>	Baud rate in bps	Specifies an override for the default baud rate (19200) at which a remote kernel debugger host will connect through a serial port.
<i>bootdebug</i>	Boolean	Enables remote boot debugging for the boot loader. With this option enabled, you can use Kd.exe or Windbg.exe to connect to the boot loader.

BCD Element	Values	Meaning
<i>bootems</i>	Boolean	Used to cause Windows to enable Emergency Management Services (EMS) for boot applications, which reports boot information and accepts system management commands through a serial port.
<i>busparams</i>	String	If a physical PCI debugging device is used to provide FireWire or serial debugging, specifies the PCI bus, function, and device number for the device.
<i>channel</i>	Channel between 0 and 62	Used in conjunction with {debugtype, 1394} to specify the IEEE 1394 channel through which kernel debugging communications will flow.
<i>configaccesspolicy</i>	Default, DisallowMmConfig	Configures whether the system uses memory mapped I/O to access the PCI manufacturer's configuration space or falls back to using the HAL's I/O port access routines. Can sometimes be helpful in solving platform device problems.
<i>debugaddress</i>	Hardware address	Specifies the hardware address of the serial (COM) port used for debugging.
<i>debugport</i>	COM port number	Specifies an override for the default serial port (usually COM2 on systems with at least two serial ports) to which a remote kernel debugger host is connected.
<i>debugstart</i>	Active, AutoEnable, Disable	Specifies settings for the debugger when kernel debugging is enabled. AutoEnable enables the debugger when a breakpoint or kernel exception, including kernel crashes, occurs.
<i>debugtype</i>	Serial, 1394, USB	Specifies whether kernel debugging will be communicated through a serial, FireWire (IEEE 1394), or USB 2.0 port. (The default is serial.)
<i>emsbaudrate</i>	Baud rate in bps	Specifies the baud rate to use for EMS.
<i>emSPORT</i>	COM port number	Specifies the serial (COM) port to use for EMS.
<i>extendedinput</i>	Boolean	Enables boot applications to leverage BIOS support for extended console input.
<i>firstmegabytepolicy</i>	UseNone, UseAll, UsePrivate	Specifies how the low 1 MB of physical memory is consumed by the HAL to mitigate corruptions by the BIOS during power transitions.
<i>fontpath</i>	String	Specifies the path of the OEM font that should be used by the boot application.
<i>graphicsmodedisabled</i>	Boolean	Disables graphics mode for boot applications.
<i>graphicsresolution</i>	Resolution	Sets the graphics resolution for boot applications.
<i>initialconsoleinput</i>	Boolean	Specifies an initial character that the system inserts into the PC/AT keyboard input buffer.
<i>integrityservices</i>	Default, Disable, Enable	Enables or disables code integrity services, which are used by Kernel Mode Code Signing. Default is Enabled.
<i>locale</i>	Localization string	Sets the locale for the boot application (such as EN-US).
<i>noumex</i>	Boolean	Disables user-mode exceptions when kernel debugging is enabled. If you experience system hangs (freezes) when booting in debugging mode, try enabling this option.
<i>novesa</i>	Boolean	Disables the usage of VESA display modes.

BCD Element	Values	Meaning
<i>recoveryenabled</i>	Boolean	Enables the recovery sequence, if any. Used by fresh installations of Windows to present the Windows PE-based Startup And Recovery interface.
<i>recoverysequence</i>	List	Defines the recovery sequence (described above).
<i>relocatephysical</i>	Physical address	Relocates an automatically selected NUMA node's physical memory to the specified physical address.
<i>targetname</i>	String	Defines the target name for the USB debugger when used with USB2 debugging {debugtype, usb}.
<i>testsigning</i>	Boolean	Enables test-signing mode, which allows driver developers to load locally signed 64-bit drivers. This option results in a watermarked desktop.
<i>traditionalksegmappings</i>	Boolean	Determines whether the kernel will honor the traditional KSEG0 mapping that was originally required for MIPS support. With KSEG0 mappings, the bottom 24 bits of the kernel's initial virtual address space will map to the same physical address (that is, 0x80800000 virtual is 0x800000 in RAM). Disabling this requirement allows more low memory to be available, which can help with some hardware.
<i>truncatememory</i>	Address in bytes	Disregards physical memory above the specified physical address.

TABLE 13-4 BCD Options for the Windows Boot Loader (Winload)

BCD Element	Values	Meaning
<i>advancedoptions</i>	Boolean	If false, executes the default behavior of launching the auto-recovery command boot entry when the boot fails; otherwise, displays the boot error and offers the user the advanced boot option menu associated with the boot entry. This is equivalent to pressing F8.
<i>bootlog</i>	Boolean	Causes Windows to write a log of the boot to the file %SystemRoot%\Ntbtlog.txt.
<i>bootstatuspolicy</i>	DisplayAllFailures, IgnoreAllFailures, IgnoreShutdownFailures, IgnoreBootFailures	Overrides the system's default behavior of offering the user a troubleshooting boot menu if the system did not complete the previous boot or shutdown.
<i>bootux</i>	Disabled, Basic, Standard	Defines the boot graphics user experience that the user will see. Disabled means that no graphics will be seen during boot time (only a black screen), while Basic will display only a progress bar during load. Standard displays the usual Windows logo animation during boot.
<i>clustermodeaddressing</i>	Number of processors	Defines the maximum number of processors to include in a single Advanced Programmable Interrupt Controller (APIC) cluster.
<i>configflags</i>	Flags	Specifies processor-specific configuration flags.
<i>dbgtransport</i>	Transport image name	Overrides using one of the default kernel debugging transports (Kdcom.dll, Kd1394, Kdusb.dll) and instead uses the given file, permitting specialized debugging transports to be used that are not typically supported by Windows.
<i>debug</i>	Boolean	Enables kernel-mode debugging.

BCD Element	Values	Meaning
<i>detecthal</i>	Boolean	Enables the dynamic detection of the HAL.
<i>driverloadfailurepolicy</i>	Fatal, UseErrorControl	Describes the loader behavior to use when a boot driver has failed to load. Fatal will prevent booting, while UseErrorControl causes the system to honor a driver's default error behavior, specified in its service key.
<i>ems</i>	Boolean	Instructs the kernel to use EMS as well. (If only <i>bootems</i> is used, only the boot loader will use EMS.)
<i>evstore</i>	String	Stores the location of a boot preloaded hive.
<i>exportascd</i>	Boolean	If this option is set, the kernel will treat the ramdisk file specified as an ISO image and not a Windows Installation Media (WIM) or System Deployment Image (SDI) file.
<i>groupaware</i>	Boolean	Forces the system to use groups other than zero when associating the group seed to new processes. Used only on 64-bit Windows.
<i>groupsize</i>	Integer	Forces the maximum number of logical processors that can be part of a group (maximum of 64). Can be used to force groups to be created on a system that would normally not require them to exist. Must be a power of 2, and is used only on 64-bit Windows.
<i>hal</i>	HAL image name	Overrides the default file name for the HAL image (hal.dll). This option can be useful when booting a combination of a checked HAL and checked kernel (requires specifying the <i>kernel</i> element as well).
<i>halbreakpoint</i>	Boolean	Causes the HAL to stop at a breakpoint early in HAL initialization. The first thing the Windows kernel does when it initializes is to initialize the HAL, so this breakpoint is the earliest one possible (unless boot debugging is used). If the switch is used without the /DEBUG switch, the system will elicit a blue screen with a STOP code of 0x00000078 (PHASE0_ EXCEPTION).
<i>hypervisorbaudrate</i>	Baud rate in bps	If using serial hypervisor debugging, specifies the baud rate to use.
<i>hypervisorchannel</i>	Channel number from 0 to 62	If using FireWire (IEEE 1394) hypervisor debugging, specifies the channel number to use.
<i>hypervisordebug</i>	Boolean	Enables debugging the hypervisor.
<i>hypervisordebugport</i>	COM port number	If using serial hypervisor debugging, specifies the COM port to use.
<i>hypervisordebugtype</i>	Serial, 1394	Specifies which hardware port to use for hypervisor debugging.
<i>hypervisordisableslat</i>	Boolean	Forces the hypervisor to ignore the presence of the Second Layer Address Translation (SLAT) feature if supported by the processor.
<i>hypervisorlaunchtype</i>	Off, Auto	Enables loading of the hypervisor on a Hyper-V system, or forces it to be disabled.
<i>hypervisorpath</i>	Hypervisor binary image name	Specifies the path of the hypervisor binary.
<i>hypervisoruselargevtlb</i>	Boolean	Enables the hypervisor to use a larger amount of virtual TLB entries.

BCD Element	Values	Meaning
<i>increaseuserva</i>	Size in MB	Increases the size of the user process address space from 2 GB to the specified size, up to 3 GB (and therefore reduces the size of system space). Giving virtual-memory-intensive applications such as database servers a larger address space can improve their performance. (See the section "Address Space Layout" in Chapter 9 for more information.)
<i>kernel</i>	Kernel image name	Overrides the default file name for the kernel image (Ntldr.exe). This option can be useful when booting a combination of a checked HAL and checked kernel (requires specifying the <i>hal</i> element to be used as well).
<i>lastknowngood</i>	Boolean	Boots the last known good configuration, instead of the current control set.
<i>loadoptions</i>	Extra command-line parameters	This option is used to add other command-line parameters that are not defined by BCD elements. These parameters could be used to configure or define the operation of other components on the system that might not be able to use the BCD (such as legacy components).
<i>maxgroup</i>	Boolean	Maximizes the number of processor groups that are created during processor topology configuration. See Chapter 3 in Part 1 for more information about group selection and its relationship to NUMA.
<i>maxproc</i>	Boolean	Forces the maximum number of supported processors that Windows will report to drivers and applications to accommodate the arrival of additional CPUs via dynamic processor support.
<i>msi</i>	Default, ForceDisable	Allows disabling support for message signaled interrupts.
<i>nocrashautoreboot</i>	Boolean	Disables the automatic reboot after a system crash (blue screen).
<i>nointegritychecks</i>	Boolean	Disables integrity checks performed by Windows when loading drivers. Automatically removed at the next reboot.
<i>nowmem</i>	Boolean	Requires that PAE be enabled and that the system have more than 4 GB of physical memory. If these conditions are met, the PAE-enabled version of the Windows kernel, Ntldr.exe, won't use the first 4 GB of physical memory. Instead, it will load all applications and device drivers and allocate all memory pools from above that boundary. This switch is useful only to test device-driver compatibility with large memory systems.
<i>numproc</i>	Number of processors	Specifies the number of CPUs that can be used on a multiprocessor system. Example: /NUMPROC=2 on a four-way system will prevent Windows from using two of the four processors.
<i>nx</i>	OptIn, OptOut, AlwaysOff, AlwaysOn	This option is available only on 32-bit versions of Windows when running on processors that support no-execute memory and only when PAE (explained further in the <i>pae</i> entry) is also enabled. It enables no-execute protection. No-execute protection is always enabled on 64-bit versions of Windows on x64 processors. See Chapter 9 for a description of this behavior.
<i>onecpu</i>	Boolean	Causes Windows to use only one CPU on a multiprocessor system.

BCD Element	Values	Meaning
<i>optionsedit</i>	Boolean	Enables the options editor in the Boot Manager. With this option, Boot Manager allows the user to interactively set on-demand command-line options and switches for the current boot. This is equivalent to pressing F10.
<i>osdevice</i>	GUID	Specifies the device on which the operating system is installed.
<i>pae</i>	Default, ForceEnable, ForceDisable	Default allows the boot loader to determine whether the system supports PAE and loads the PAE kernel. ForceEnable forces this behavior, while ForceDisable forces the loader to load the non-PAE version of the Windows kernel, even if the system is detected as supporting x86 PAEs and has more than 4 GB of physical memory.
<i>pciexpress</i>	Default, ForceDisable	Can be used to disable support for PCI Express buses and devices.
<i>perfmem</i>	Size in MB	Size of the buffer to allocate for performance data logging. This option acts similarly to the <i>removememory</i> element, since it prevents Windows from seeing the size specified as available memory.
<i>quietboot</i>	Boolean	Instructs Windows not to initialize the VGA video driver responsible for presenting bitmapped graphics during the boot process. The driver is used to display boot progress information, so disabling it will disable the ability of Windows to show this information.
<i>ramdiskimagelength</i>	Length in bytes	Size of the ramdisk specified.
<i>ramdiskimageoffset</i>	Offset in bytes	If the ramdisk contains other data (such as a header) before the virtual file system, instructs the boot loader where to start reading the ramdisk file from.
<i>ramdisksdipath</i>	Image file name	Specifies the name of the SDI ramdisk to load.
<i>ramdisktftpblocksize</i>	Block size	If loading a WIM ramdisk from a network Trivial FTP (TFTP) server, specifies the block size to use.
<i>ramdisktftpclientport</i>	Port number	If loading a WIM ramdisk from a network TFTP server, specifies the port.
<i>ramdisktftpwindowsize</i>	Window size	If loading a WIM ramdisk from a network TFTP server, specifies the window size to use.
<i>removememory</i>	Size in bytes	Specifies an amount of memory Windows won't use.
<i>restrictapiccluster</i>	Cluster number	Defines the largest APIC cluster number to be used by the system.
<i>resumeobject</i>	Object GUID	Describes which application to use for resuming from hibernation, typically Winresume.exe.
<i>safeboot</i>	Minimal, Network, DsRepair	Specifies options for a safe-mode boot. Minimal corresponds to safe mode without networking, Network to safe mode with networking, and DsRepair to safe mode with Directory Services Restore mode. (Safe mode is described later in this chapter.)
<i>safebootalternateshell</i>	Boolean	Tells Windows to use the program specified by the HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell value as the graphical shell rather than the default, which is Windows Explorer. This option is referred to as Safe Mode With Command Prompt in the alternate boot menu.

BCD Element	Values	Meaning
<i>sos</i>	Boolean	Causes Windows to list the device drivers marked to load at boot time and then to display the system version number (including the build number), amount of physical memory, and number of processors.
<i>stampdisks</i>	Boolean	Specifies that Winload will write an MBR disk signature to a RAW disk when booting Windows PE (Preinstallation Environment). This can be required in deployment environments in order to create a mapping from operating system–enumerated hard disks to BIOS–enumerated hard disks to know which disk should be the system disk.
<i>systemroot</i>	String	Specifies the path, relative to <i>osdevice</i> , in which the operating system is installed.
<i>targetname</i>	Name	For USB 2.0 debugging, assigns a name to the machine that is being debugged.
<i>tpmbootentropy</i>	Default, ForceDisable, ForceEnable	Forces a specific TPM Boot Entropy policy to be selected by the boot loader and passed on to the kernel. TPM Boot Entropy, when used, seeds the kernel’s random number generator (RNG) with data obtained from the TPM (if present).
<i>usefirmwarepcisettings</i>	Boolean	Stops Windows from dynamically assigning IO/IRQ resources to PCI devices and leaves the devices configured by the BIOS. See Microsoft Knowledge Base article 148501 for more information.
<i>uselegacyapicmode</i>	Boolean	Forces usage of basic APIC functionality even though the chipset reports extended APIC functionality as present. Used in cases of hardware errata and/or incompatibility.
<i>usephysicaldestination</i>	Boolean	Forces the use of the APIC in physical destination mode.
<i>useplatformclock</i>	Boolean	Forces usage of the platform’s clock source as the system’s performance counter.
<i>vga</i>	Boolean	Forces Windows to use the VGA display driver instead of the third-party high-performance driver.
<i>winpe</i>	Boolean	Used by Windows PE, this option causes the configuration manager to load the registry SYSTEM hive as a volatile hive such that changes made to it in memory are not saved back to the hive image.
<i>x2apicpolicy</i>	Disabled, Enabled, Default	Specifies whether extended APIC functionality should be used if the chipset supports it. Disabled is equivalent to setting <i>uselegacyapicmode</i> , while Enabled forces ACPI functionality on even if errata are detected. Default uses the chipset’s reported capabilities (unless errata are present).
<i>xsavepolicy</i>	Integer	Forces the given XSAVE policy to be loaded from the XSAVE Policy Resource Driver (Hwpolicy.sys).
<i>xsaveaddfeature0-7</i>	Integer	Used while testing support for XSAVE on modern Intel processors; allows for faking that certain processor features are present when, in fact, they are not. This helps increase the size of the CONTEXT structure and confirms that applications work correctly with extended features that might appear in the future. No actual extra functionality will be present, however.
<i>xsaveremovefeature</i>	Integer	Forces the entered XSAVE feature not to be reported to the kernel, even though the processor supports it.

BCD Element	Values	Meaning
<i>xsaveprocessorsmask</i>	Integer	Bitmask of which processors the XSAVE policy should apply to.
<i>xsavedisable</i>	Boolean	Turns off support for the XSAVE functionality even though the processor supports it.

If the user doesn't select an entry from the selection menu within the timeout period the BCD specifies, Bootmgr chooses the default selection specified in the BCD (if there is only one entry, it immediately chooses this one). Once the boot selection has been made, Bootmgr loads the boot loader associated with that entry, which will be Winload.exe for Windows installations.

Winload.exe also contains code that queries the system's ACPI BIOS to retrieve basic device and configuration information. This information includes the following:

- The time and date information stored in the system's CMOS (nonvolatile memory)
- The number, size, and type of disk drives on the system
- Legacy device information, such as buses (for example, ISA, PCI, EISA, Micro Channel Architecture [MCA]), mice, parallel ports, and video adapters are not queried and instead faked out

This information is gathered into internal data structures that will be stored under the HKLM\HARDWARE\DESCRIPTION registry key later in the boot. This is mostly a legacy key as CMOS settings and BIOS-detected disk drive configuration settings, as well as legacy buses, are no longer supported by Windows, and this information is mainly stored for compatibility reasons. Today, it is the Plug and Play manager database that stores the true information on hardware.

Next, Winload begins loading the files from the *boot volume* needed to start the kernel initialization. The boot volume is the volume that corresponds to the partition on which the system directory (usually \Windows) of the installation being booted is located. The steps Winload follows here include:

1. Loads the appropriate kernel and HAL images (Ntoskrnl.exe and Hal.dll by default) as well as any of their dependencies. If Winload fails to load either of these files, it prints the message "Windows could not start because the following file was missing or corrupt", followed by the name of the file.
2. Reads in the VGA font file (by default, vgaoem.fon). If this file fails, the same error message as described in step 1 will be shown.
3. Reads in the NLS (National Language System) files used for internationalization. By default, these are I_intl.nls, c_1252.nls, and c_437.nls.
4. Reads in the SYSTEM registry hive, \Windows\System32\Config\System, so that it can determine which device drivers need to be loaded to accomplish the boot. (A hive is a file that contains a registry subtree. You'll find more details about the registry in Chapter 4, "Management Mechanisms," in Part 1.)
5. Scans the in-memory SYSTEM registry hive and locates all the *boot device drivers*. Boot device drivers are drivers necessary to boot the system. These drivers are indicated in the registry

by a start value of SERVICE_BOOT_START (0). Every device driver has a registry subkey under HKLM\SYSTEM\CurrentControlSet\Services. For example, Services has a subkey named fvevol for the BitLocker driver, which you can see in Figure 13-2. (For a detailed description of the Services registry entries, see the section “Services” in Chapter 4 in Part 1.)

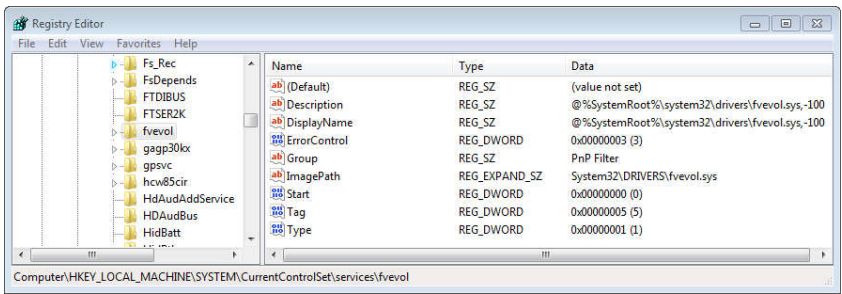


FIGURE 13-2 BitLocker driver service settings

- 6. Adds the file system driver that’s responsible for implementing the code for the type of partition (NTFS) on which the installation directory resides to the list of boot drivers to load. Winload must load this driver at this time; if it didn’t, the kernel would require the drivers to load themselves, a requirement that would introduce a circular dependency.
- 7. Loads the boot drivers, which should only be drivers that, like the file system driver for the boot volume, would introduce a circular dependency if the kernel was required to load them. To indicate the progress of the loading, Winload updates a progress bar displayed below the text “Starting Windows”. If the sos option is specified in the BCD, Winload doesn’t display the progress bar but instead displays the file names of each boot driver. Keep in mind that the drivers are loaded but not initialized at this time—they initialize later in the boot sequence.
- 8. Prepares CPU registers for the execution of Ntoskrnl.exe.

For steps 1 and 8, Winload also implements part of the Kernel Mode Code Signing (KMCS) infrastructure, which was described in Chapter 3 in Part 1, by enforcing that all boot drivers are signed on 64-bit Windows. Additionally, the system will crash if the signature of the early boot files is incorrect.

This action is the end of Winload’s role in the boot process. At this point, Winload calls the main function in Ntoskrnl.exe (*KiSystemStartup*) to perform the rest of the system initialization.

The UEFI Boot Process

A UEFI-compliant system has firmware that runs boot loader code that’s been programmed into the system’s nonvolatile RAM (NVRAM) by Windows Setup. The boot code reads the BCD’s contents, which are also stored in NVRAM. The Bcdedit.exe tool mentioned earlier also has the ability to abstract the firmware’s NVRAM variables in the BCD, allowing for full transparency of this mechanism.

The UEFI standard defines the ability to prompt the user with an EFI Boot Manager that can be used to select an operating system or additional applications to load. However, to provide a consistent user interface between BIOS systems and UEFI systems, Windows sets a 2-second timeout for selecting the EFI Boot Manager, after which the EFI-version of Bootmgr (Bootmgfw.efi) loads instead.

Hardware detection occurs next, where the boot loader uses UEFI interfaces to determine the number and type of the following devices:

- Network adapters
- Video adapters
- Keyboards
- Disk controllers
- Storage devices

On UEFI systems, all operations and programs execute in the native CPU mode with paging enabled and no part of the Windows boot process executes in 16-bit mode. Note that although EFI is supported on both 32-bit and 64-bit systems, Windows provides support for EFI only on 64-bit platforms.

Just as Bootmgr does on x86 and x64 systems, the EFI Boot Manager presents a menu of boot selections with an optional timeout. Once a boot selection is made, the loader navigates to the subdirectory on the EFI System partition corresponding to the selection and loads the EFI version of the Windows boot loader (Winload.efi).

The UEFI specification requires that the system have a partition designated as the EFI System partition that is formatted with the FAT file system and is between 100 MB and 1 GB in size or up to 1 percent of the size of the disk, and each Windows installation has a subdirectory on the EFI System partition under EFI\Microsoft.

Note that thanks to the unified boot process and model present in Windows, the components in Table 13-1 apply almost identically to UEFI systems, except that those ending in *.exe* end in *.efi*, and they use EFI APIs and services instead of BIOS interrupts. Another difference is that to avoid limitations of the MBR partition format (including a maximum of four partitions per disk), UEFI systems use the GPT (GUID Partition Table) format, which uses GUIDs to identify different partitions and their roles on the system.



Note Although the EFI standard has been available since early 2001, and UEFI since 2005, very few computer manufacturers have started using this technology because of backward compatibility concerns and the difficulty of moving from an entrenched 20-year-old technology to a new one. Two notable exceptions are Itanium machines and Apple's Intel Macintosh computers.

Booting from iSCSI

Internet SCSI (iSCSI) devices are a kind of network-attached storage, in that remote physical disks are connected to an iSCSI Host Bus Adapter (HBA) or through Ethernet. These devices, however, are different from traditional network-attached storage (NAS) because they provide block-level access to disks, unlike the logical-based access over a network file system that NAS employs. Therefore, an iSCSI-connected disk appears as any other disk drive, both to the boot loader as well as to the OS, as long as the Microsoft iSCSI Initiator is used to provide access over an Ethernet connection. By using iSCSI-enabled disks instead of local storage, companies can save on space, power consumption, and cooling.

Although Windows has traditionally supported booting only from locally connected disks, or network booting through PXE, modern versions of Windows are also capable of natively booting from iSCSI devices through a mechanism called *iSCSI Boot*. The boot loader (Winload.exe) contains a minimalistic network stack conforming to the Universal Network Device Interface (UNDI) standard, which allows compatible NIC ROMs to respond to Interrupt 13h (the legacy BIOS disk I/O interrupt) and convert the requests to network I/O. On EFI systems, the network interface driver provided by the manufacturer is used instead, and EFI Device APIs are used instead of interrupts.

Finally, to know the location, path, and authentication information for the remote disk, the boot loader also reads an iSCSI Boot Firmware Table (iBFT) that must be present in physical memory (typically exposed through ACPI). Additionally, Windows Setup also has the capability of reading this table to determine bootable iSCSI devices and allow direct installation on such a device, such that no imaging is required. Combined with the Microsoft iSCSI Initiator, this is all that's required for Windows to boot from iSCSI, as shown in Figure 13-3.

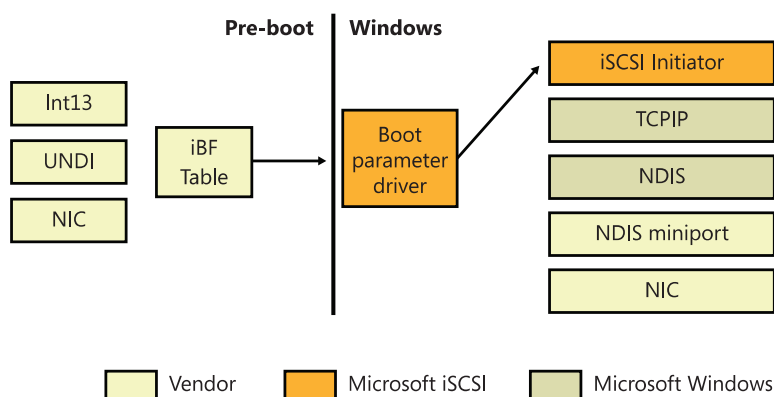


FIGURE 13-3 iSCSI boot architecture

Initializing the Kernel and Executive Subsystems

When Winload calls Ntoskrnl, it passes a data structure called the loader parameter block that contains the system and boot partition paths, a pointer to the memory tables Winload generated to describe the physical memory on the system, a physical hardware tree that is later used to build the

volatile **HARDWARE** registry hive, an in-memory copy of the **SYSTEM** registry hive, and a pointer to the list of boot drivers Winload loaded, as well as various other information related to the boot processing performed until this point.



EXPERIMENT: Loader Parameter Block

While booting, the kernel keeps a pointer to the loader parameter block in the *KeLoaderBlock* variable. The kernel discards the parameter block after the first boot phase, so the only way to see the contents of the structure is to attach a kernel debugger before booting and break at the initial kernel debugger breakpoint. If you are able to do so, you can use the *dt* command to dump the block, as shown:

```
0: kd> dt poi(nt!KeLoaderBlock) nt!_LOADER_PARAMETER_BLOCK
+0x000 OsMajorVersion      : 6
+0x004 OsMinorVersion      : 1
+0x008 Size                : 0x88
+0x00c Reserved            : 0
+0x010 LoadOrderListHead   : _LIST_ENTRY [ 0x8085b4c8 - 0x80869c70 ]
+0x018 MemoryDescriptorListHead : _LIST_ENTRY [ 0x80a00000 - 0x80a00de8 ]
+0x020 BootDriverListHead   : _LIST_ENTRY [ 0x80860d10 - 0x8085eba0 ]
+0x028 KernelStack         : 0x88e7c000
+0x02c Prcb                : 0
+0x030 Process             : 0
+0x034 Thread              : 0x88e64800
+0x038 RegistryLength      : 0x2940000
+0x03c RegistryBase        : 0x80adf000 Void
+0x040 ConfigurationRoot    : 0x8082d450 _CONFIGURATION_COMPONENT_DATA
+0x044 ArcBootDeviceName    : 0x8082d9a0 "multi(0)disk(0)rdisk(0)partition(4)"
+0x048 ArcHalDeviceName     : 0x8082d788 "multi(0)disk(0)rdisk(0)partition(4)"
+0x04c NtBootPathName       : 0x8082d828 "\\Windows\\"
+0x050 NtHalPathName        : 0x80826358 "\\\"
+0x054 LoadOptions         : 0x8080e1b0 "NOEXECUTE=ALWAYSON DEBUGPORT=COM1
                                BAUDRATE=115200"
+0x058 NlsData              : 0x808691e0 _NLS_DATA_BLOCK
+0x05c ArcDiskInformation   : 0x80821408 _ARC_DISK_INFORMATION
+0x060 OemFontFile          : 0x84a551d0 Void
+0x064 Extension            : 0x8082d9d8 _LOADER_PARAMETER_EXTENSION
+0x068 u                    : <unnamed-tag>
+0x074 FirmwareInformation  : _FIRMWARE_INFORMATION_LOADER_BLOCK
```

Additionally, the *!loadermemorylist* command can be used on the *MemoryDescriptorListHead* field to dump the physical memory ranges:

```
0: kd> !loadermemorylist 0x80a00000
Base Length      Type
1      00000001   HALCachedMemory
2      00000004   HALCachedMemory
...
4a32   00000023   NlsData
4a55   00000002   BootDriver
4a57   00000026   BootDriver
4a7d   00000014   BootDriver
4a91   0000016f   Free
```

```

4c00    0001b3f0    Free
1fff0   00000001    FirmwarePermanent
1fff1   00000002    FirmwarePermanent
1fff3   00000001    FirmwarePermanent
1fff4   0000000b    FirmwarePermanent
1ffff   00000001    FirmwarePermanent
fd000   00000800    FirmwarePermanent
fec00   00000001    FirmwarePermanent
fee00   00000001    FirmwarePermanent
ffc00   00000400    FirmwarePermanent

Summary
Memory Type      Pages
Free             0001bc50  ( 113744)
LoadedProgram    0000013d  (   317)
FirmwareTemporary 000006dd  (   1757)
FirmwarePermanent 00000c37  (   3127)
OsloaderHeap     0000022a  (    554)
SystemCode       000005dc  (   1500)
BootDriver       00000968  (   2408)
RegistryData     00002940  (  10560)
MemoryData       00000035  (    53)
NlsData          00000023  (    35)
HALCachedMemory 0000001e  (    30)
=====
Total           00020bc5  ( 134085) = ~523MB

```

Ntoskrnl then begins *phase 0*, the first of its two-phase initialization process (*phase 1* is the second). Most executive subsystems have an initialization function that takes a parameter that identifies which phase is executing.

During phase 0, interrupts are disabled. The purpose of this phase is to build the rudimentary structures required to allow the services needed in phase 1 to be invoked. Ntoskrnl's main function calls *KiSystemStartup*, which in turn calls *HallInitializeProcessor* and *KilInitializeKernel* for each CPU. *KilInitializeKernel*, if running on the boot CPU, performs systemwide kernel initialization, such as initializing internal lists and other data structures that all CPUs share. It also checks whether virtualization was specified as a BCD option (*hypervisorlaunchtype*), and whether the CPU supports hardware virtualization technology. The first instance of *KilInitializeKernel* then calls the function responsible for orchestrating phase 0, *InitBootProcessor*, while subsequent processors only call *HallInitSystem*.

InitBootProcessor starts by initializing the pool look-aside pointers for the initial CPU and by checking for and honoring the BCD *burnmemory* boot option, where it discards the amount of physical memory the value specifies. It then performs enough initialization of the NLS files that were loaded by Winload (described earlier) to allow Unicode to ANSI and OEM translation to work. Next, it continues by calling the HAL function *HallInitSystem*, which gives the HAL a chance to gain system control before Windows performs significant further initialization. One responsibility of *HallInitSystem* is to prepare the system interrupt controller of each CPU for interrupts and to configure the interval clock timer interrupt, which is used for CPU time accounting. (See the section "Quantum Accounting" in Chapter 5, "Processes, Threads, and Jobs," in Part 1 for more on CPU time accounting.)

When *HallInitSystem* returns control, *InitBootProcessor* proceeds by computing the reciprocal for timer expiration. Reciprocals are used for optimizing divisions on most modern processors. They can perform multiplications faster, and because Windows must divide the current 64-bit time value in order to find out which timers need to expire, this static calculation reduces interrupt latency when the clock interval fires. *InitBootProcessor* then continues by setting up the system root path and searching the kernel image for the location of the crash message strings it displays on blue screens, caching their location to avoid looking up the strings during a crash, which could be dangerous and unreliable. Next, *InitBootProcessor* initializes the quota functionality part of the process manager and reads the *control vector*. This data structure contains more than 150 kernel-tuning options that are part of the HKLM\SYSTEM\CurrentControlSet\Control registry key, including information such as the licensing data and version information for the installation.

InitBootProcessor is now ready to call the phase 0 initialization routines for the executive, Driver Verifier, and the memory manager. These components perform the following initialization steps:

1. The executive initializes various internal locks, resources, lists, and variables and validates that the product suite type in the registry is valid, discouraging casual modification of the registry in order to “upgrade” to an SKU of Windows that was not actually purchased. This is only one of the many such checks in the kernel.
2. Driver Verifier, if enabled, initializes various settings and behaviors based on the current state of the system (such as whether safe mode is enabled) and verification options. It also picks which drivers to target for tests that target randomly chosen drivers.
3. The memory manager constructs page tables and internal data structures that are necessary to provide basic memory services. It also builds and reserves an area for the system file cache and creates memory areas for the paged and nonpaged pools (described in Chapter 10). The other executive subsystems, the kernel, and device drivers use these two memory pools for allocating their data structures.

Next, *InitBootProcessor* calls *HallInitializeBios* to set up the BIOS emulation code part of the HAL. This code is used both on real BIOS systems as well as on EFI systems to allow access (or to emulate access) to 16-bit real mode interrupts and memory, which are used mainly by Bootvid to display the early VGA boot screen and bugcheck screen. After the function returns, the kernel initializes the Bootvid library and displays early boot status messages by calling *InbvEnableBootDriver* and *InbvDriverInitailize*.

At this point, *InitBootProcessor* enumerates the boot-start drivers that were loaded by Winload and calls *DbgLoadImageSymbols* to inform the kernel debugger (if attached) to load symbols for each of these drivers. If the host debugger has configured the *break on symbol load* option, this will be the earliest point for a kernel debugger to gain control of the system. *InitBootProcessor* now calls *HvllInitSystem*, which attempts to connect to the hypervisor in case Windows might be running inside a Hyper-V host system’s child partition. When the function returns, it calls *HeadlessInit* to initialize the serial console if the machine was configured for Emergency Management Services (EMS).

Next, *InitBootProcessor* builds the versioning information that will be used later in the boot process, such as the build number, service pack version, and beta version status. Then it copies the NLS

tables that Winload previously loaded into paged pool, re-initializes them, and creates the kernel stack trace database if the global flags specify creating one. (For more information on the global flags, see Chapter 3 in Part 1.)

Finally, *InitBootProcessor* calls the object manager, security reference monitor, process manager, user-mode debugging framework, and the Plug and Play manager. These components perform the following initialization steps:

1. During the object manager initialization, the objects that are necessary to construct the object manager namespace are defined so that other subsystems can insert objects into it. A handle table is created so that resource tracking can begin.
2. The security reference monitor initializes the token type object and then uses the object to create and prepare the first local system account token for assignment to the initial process. (See Chapter 6, "Security," in Part 1 for a description of the local system account.)
3. 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 a system thread to execute the routine *Phase1Initialization*. This thread doesn't start running right away because interrupts are still disabled.
4. The user-mode debugging framework creates the definition of the debug object type that is used for attaching a debugger to a process and receiving debugger events. For more information on user-mode debugging, see Chapter 3 in Part 1.
5. The Plug and Play manager's phase 0 initialization then takes place, which involves simply initializing an executive resource used to synchronize access to bus resources.

When control returns to *KiInitializeKernel*, the last step is to allocate the DPC stack for the current processor and the I/O privilege map save area (on x86 systems only), after which control proceeds to the *Idle* loop, which then causes the system thread created in step 3 of the previous process description to begin executing phase 1. (Secondary processors wait to begin their initialization until step 8 of phase 1, described in the following list.)

Phase 1 consists of the following steps:

1. *Phase1InitializationDiscard*, which, as the name implies, discards the code that is part of the INIT section of the kernel image in order to preserve memory.
2. The initialization thread sets its priority to 31, the highest possible, in order to prevent preemption.
3. The NUMA/group topology relationships are created, in which the system tries to come up with the most optimized mapping between logical processors and processor groups, taking into account NUMA localities and distances, unless overridden by the relevant BCD settings.

4. *HalInitSystem* prepares the system to accept interrupts from devices and to enable interrupts.
5. The boot video driver is called, which in turn displays the Windows startup screen, which by default consists of a black screen and a progress bar. If the *quietboot* boot option was used, this step will not occur.
6. The kernel builds various strings and version information, which are displayed on the boot screen through Bootvid if the *sos* boot option was enabled. This includes the full version information, number of processors supported, and amount of memory supported.
7. The power manager's initialization is called.
8. The system time is initialized (by calling *HalQueryRealTimeClock*) and then stored as the time the system booted.
9. On a multiprocessor system, the remaining processors are initialized by *KeStartAllProcessors* and *HalAllProcessorsStarted*. The number of processors that will be initialized and supported depends on a combination of the actual physical count, the licensing information for the installed SKU of Windows, boot options such as *numproc* and *onecpu*, and whether dynamic partitioning is enabled (server systems only). After all the available processors have initialized, the affinity of the system process is updated to include all processors.
10. The object manager creates the namespace root directory (\), \ObjectTypes directory, and the DOS device name mapping directory (\Global??). It then creates the \DosDevices symbolic link that points at the Windows subsystem device name mapping directory.
11. The executive is called to create the executive object types, including semaphore, mutex, event, and timer.
12. The I/O manager is called to create the I/O manager object types, including device, driver, controller, adapter, and file objects.
13. The kernel debugger library finalizes initialization of debugging settings and parameters if the debugger has not been triggered prior to this point.
14. The transaction manager also creates its object types, such as the enlistment, resource manager, and transaction manager types.
15. The kernel initializes scheduler (dispatcher) data structures and the system service dispatch table.
16. The user-mode debugging library (Dbgk) data structures are initialized.
17. If Driver Verifier is enabled and, depending on verification options, pool verification is enabled, object handle tracing is started for the system process.
18. The security reference monitor creates the \Security directory in the object manager namespace and initializes auditing data structures if auditing is enabled.

19. The \SystemRoot symbolic link is created.
20. The memory manager is called to create the \Device\PhysicalMemory section object and the memory manager's system worker threads (which are explained in Chapter 10).
21. NLS tables are mapped into system space so that they can be easily mapped by user-mode processes.
22. Ntdll.dll is mapped into the system address space.
23. The cache manager initializes the file system cache data structures and creates its worker threads.
24. The configuration manager creates the \Registry key object in the object manager namespace and opens the in-memory SYSTEM hive as a proper hive file. It then copies the initial hardware tree data passed by Winload into the volatile HARDWARE hive.
25. The high-resolution boot graphics library initializes, unless it has been disabled through the BCD or the system is booting headless.
26. The errata manager initializes and scans the registry for errata information, as well as the INF (driver installation file, described in Chapter 8) database containing errata for various drivers.
27. Superfetch and the prefetcher are initialized.
28. The Store Manager is initialized.
29. The current time zone information is initialized.
30. Global file system driver data structures are initialized.
31. Phase 1 of debugger-transport-specific information is performed by calling the *KdDebugger-Initialize1* routine in the registered transport, such as Kdcom.dll.
32. The Plug and Play manager calls the Plug and Play BIOS.
33. The advanced local procedure call (ALPC) subsystem initializes the ALPC port type and ALPC waitable port type objects. The older LPC objects are set as aliases.
34. If the system was booted with boot logging (with the BCD *bootlog* option), the boot log file is initialized. If the system was booted in safe mode, a string is displayed on the boot screen with the current safe mode boot type.
35. The executive is called to execute its second initialization phase, where it configures part of the Windows licensing functionality in the kernel, such as validating the registry settings that hold license data. Also, if persistent data from boot applications is present (such as memory diagnostic results or resume from hibernation information), the relevant log files and information are written to disk or to the registry.

- 36.** The MiniNT/WinPE registry keys are created if this is such a boot, and the NLS object directory is created in the namespace, which will be used later to host the section objects for the various memory-mapped NLS files.
- 37.** The power manager is called to initialize again. This time it sets up support for power requests, the ALPC channel for brightness notifications, and profile callback support.
- 38.** The I/O manager initialization now takes place. This stage is a complex phase of system startup that accounts for most of the boot time.

The I/O manager first initializes various internal structures and creates the driver and device object types. It then calls the Plug and Play manager, power manager, and HAL to begin the various stages of dynamic device enumeration and initialization. (Because this process is complex and specific to the I/O system, we cover the details in Chapter 8.) Then the Windows Management Instrumentation (WMI) subsystem is initialized, which provides WMI support for device drivers. (See the section “Windows Management Instrumentation” in Chapter 4 in Part 1 for more information.) This also initializes Event Tracing for Windows (ETW). Next, all the boot-start drivers are called to perform their driver-specific initialization, and then the system-start device drivers are loaded and initialized. (Details on the processing of the driver load control information on the registry are also covered in Chapter 8.) Finally, the Windows subsystem device names are created as symbolic links in the object manager’s namespace.

- 39.** The transaction manager sets up the Windows software trace preprocessor (WPP) and ETW and initializes with WMI. (ETW and WMI are described in Chapter 4 in Part 1.)
- 40.** Now that boot-start and system-start drivers are loaded, the errata manager loads the INF database with the driver errata and begins parsing it, which includes applying registry PCI configuration workarounds.
- 41.** If the computer is booting in safe mode, this fact is recorded in the registry.
- 42.** Unless explicitly disabled in the registry, paging of kernel-mode code (in Ntoskrnl and drivers) is enabled.
- 43.** The configuration manager makes sure that all processors on an SMP system are identical in terms of the features that they support; otherwise, it crashes the system.
- 44.** On 32-bit systems, VDM (Virtual Dos Machine) support is initialized, which includes determining whether the processor supports Virtual Machine Extensions (VME).
- 45.** The process manager is called to set up rate limiting for jobs, initialize the static environment for protected processes, and look up the various system-defined entry points in the user-mode system library (Ntdll.dll).
- 46.** The power manager is called to finalize its initialization.

47. The rest of the licensing information for the system is initialized, including caching the current policy settings stored in the registry.
48. The security reference monitor is called to create the Command Server Thread that communicates with LSASS. (See the section “Security System Components” in Chapter 6 in Part 1 for more on how security is enforced in Windows.)
49. The Session Manager (Smss) process (introduced in Chapter 2, “System Architecture,” in Part 1) is started. Smss is responsible for creating the user-mode environment that provides the visible interface to Windows—its initialization steps are covered in the next section.
50. The TPM boot entropy values are queried. These values can be queried only once per boot, and normally, the TPM system driver should have queried them by now, but if this driver had not been running for some reason (perhaps the user disabled it), the unqueried values would still be available. Therefore, the kernel manually queries them too to avoid this situation, and in normal scenarios, the kernel’s own query should fail.
51. All the memory used up by the loader parameter block and all its references is now freed.

As a final step before considering the executive and kernel initialization complete, the phase 1 initialization thread waits for the handle to the Session Manager process with a timeout value of 5 seconds. If the Session Manager process exits before the 5 seconds elapse, the system crashes with a `SESSION5_INITIALIZATION_FAILED` stop code.

If the 5-second wait times out (that is, if 5 seconds elapse), the Session Manager is assumed to have started successfully, and the phase 1 initialization function calls the memory manager’s zero page thread function (explained in Chapter 10). Thus, this system thread becomes the zero page thread for the remainder of the life of the system.

Smss, Csrss, and Wininit

Smss is like any other user-mode process except for two differences. First, Windows considers Smss a trusted part of the operating system. Second, Smss is a *native* application. Because it’s a trusted operating system 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 Windows APIs—it uses only core executive APIs known collectively as the Windows native API. Smss doesn’t use the Win32 APIs because the Windows subsystem isn’t executing when Smss launches. In fact, one of Smss’s first tasks is to start the Windows subsystem.

Smss then calls the configuration manager executive subsystem to finish initializing the registry, fleshing the registry out to include all its keys. The configuration manager is programmed to know where the core registry hives are stored on disk (excluding hives corresponding to user profiles), and it records the paths to the hives it loads in the `HKLM\SYSTEM\CurrentControlSet\Control\hivelist` key.

The main thread of Smss performs the following initialization steps:

1. Marks itself as a critical process and its main thread as a critical thread. As discussed in Chapter 5 in Part 1, this will cause the kernel to crash the system if Smss quits unexpectedly. Smss

also enables the automatic affinity update mode to support dynamic processor addition. (See Chapter 5 in Part 1 for more information.)

2. Creates *protected prefixes* for the mailslot and named pipe file system drivers, creating privileged paths for administrators and service accounts to communicate through those paths. See Chapter 7, “Networking,” in Part 1 for more information.
3. Calls *SmpInit*, which tunes the maximum concurrency level for Smss, meaning the maximum number of parallel sessions that will be created by spawning copies of Smss into other sessions. This is at least four and at most the number of active CPUs.
4. *SmpInit* then creates an ALPC port object (*\SmApiPort*) to receive client requests (such as to load a new subsystem or create a session).
5. *SmpInit* calls *SmpLoadDataFromRegistry*, which starts by setting up the default environment variables for the system, and sets the SAFEBOOT variable if the system was booted in safe mode.
6. *SmpLoadDataFromRegistry* calls *SmpInitializeDosDevices* to define the symbolic links for MS-DOS device names (such as COM1 and LPT1).
7. *SmpLoadDataFromRegistry* creates the *\Sessions* directory in the object manager’s namespace (for multiple sessions).
8. *SmpLoadDataFromRegistry* runs any programs defined in HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute with *SmpExecuteCommand*. Typically, this value contains one command to run Autochk (the boot-time version of Chkdsk).
9. *SmpLoadDataFromRegistry* calls *SmpProcessFileRenames* to perform delayed file rename and delete operations as directed by HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations and HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations2.
10. *SmpLoadDataFromRegistry* calls *SmpCreatePagingFiles* to create additional paging files. Paging file configuration is stored under HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles.
11. *SmpLoadDataFromRegistry* initializes the registry by calling the native function *NtInitializeRegistry*. The configuration manager builds the rest of the registry by loading the registry hives for the HKLM\SAM, HKLM\SECURITY, and HKLM\SOFTWARE keys. Although HKLM\SYSTEM\CurrentControlSet\Control\hivelist locates the hive files on disk, the configuration manager is coded to look for them in \Windows\System32\Config.
12. *SmpLoadDataFromRegistry* calls *SmpCreateDynamicEnvironmentVariables* to add system environment variables that are defined in HKLM\SYSTEM\CurrentControlSet\Session Manager\Environment, as well as processor-specific environment variables such as NUMBER_PROCESSORS, PROCESSOR_ARCHITECTURE, and PROCESSOR_LEVEL.

13. *SmpLoadDataFromRegistry* runs any programs defined in HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SetupExecute with *SmpExecuteCommand*. Typically, this value is set only if Windows is being booted as part of the second stage of installation and Setupcl.exe is the default value.
14. *SmpLoadDataFromRegistry* calls *SmpConfigureSharedSessionData* to initialize the list of subsystems that will be started in each session (both immediately and deferred) as well as the Session 0 initialization command (which, by default, is to launch the Wininit.exe process). The initialization command can be overridden by creating a string value called S0InitialCommand in HKLM\SYSTEM\CurrentControlSet\Control\Session Manager and setting it as the path to another program.
15. *SmpLoadDataFromRegistry* calls *SmpInitializeKnownDlls* to open known DLLs, and creates section objects for them in the \KnownDlls directory of the object manager namespace. The list of DLLs considered known is located in HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDlls, and the path to the directory in which the DLLs are located is stored in the DllDirectory value of the key. On 64-bit systems, 32-bit DLLs used as part of Wow64 are stored in the DllDirectory32 value.
16. Finally, *SmpLoadDataFromRegistry* calls *SmpTranslateSystemPartitionInformation* to convert the SystemPartition value stored in HKLM\SYSTEM\Setup, which is stored in native NT object manager path format, to a volume drive letter stored in the BootDir value. Among other components, Windows Update uses this registry key to figure out what the system volume is.
17. At this point, *SmpLoadDataFromRegistry* returns to *SmpInit*, which returns to the main thread entry point. Smss then creates the number of initial sessions that were defined (typically, only one, session 0, but you can change this number through the NumberOfInitialSessions registry value in the Smss registry key mentioned earlier) by calling *SmpCreateInitialSession*, which creates an Smss process for each user session. This function's main job is to call *SmpStartCsr* to start Csrss in each session.
18. As part of Csrss's initialization, it loads the kernel-mode part of the Windows subsystem (Win32k.sys). The initialization code in Win32k.sys uses the video driver to switch the screen to the resolution defined by the default profile, so this is the point at which the screen changes from the VGA mode the boot video driver uses to the default resolution chosen for the system.
19. Meanwhile, each spawned Smss in a different user session starts the other subsystem processes, such as Psxss if the Subsystem for Unix-based Applications feature was installed. (See Chapter 3 in Part 1 for more information on subsystem processes.)
20. The first Smss from session 0 executes the Session 0 initialization command (described in step 14), by default launching the Windows initialization process (Wininit). Other Smss instances start the interactive logon manager process (Winlogon), which, unlike Wininit, is hardcoded. The startup steps of Wininit and Winlogon are described shortly.

Pending File Rename Operations

The fact that executable images and DLLs are memory-mapped when they are used makes it impossible to update core system files after Windows has finished booting (unless hotpatching technology is used, which is only for Microsoft patches to the operating system). The *MoveFileEx* Windows API has an option to specify that a file move be delayed until the next boot. Service packs and hotfixes that must update in-use memory-mapped files install replacement files onto a system in temporary locations and use the *MoveFileEx* API to have them replace otherwise in-use files. When used with that option, *MoveFileEx* simply records commands in the PendingFileRenameOperations and PendingFileRenameOperations2 keys under HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. These registry values are of type MULTI_SZ, where each operation is specified in pairs of file names: the first file name is the source location, and the second is the target location. Delete operations use an empty string as their target path. You can use the Pendmoves utility from Windows Sysinternals (<http://www.microsoft.com/technet/sysinternals>) to view registered delayed rename and delete commands.

After performing these initialization steps, the main thread in Smss waits forever on the process handle of Winlogon, while the other ALPC threads wait for messages to create new sessions or subsystems. If either Wininit or Csrss terminate unexpectedly, the kernel crashes the system because these processes are marked as *critical*. If Winlogon terminates unexpectedly, the session associated with it is logged off.

Wininit then performs its startup steps, such as creating the initial window station and desktop objects. It also configures the Session 0 window hook, which is used by the Interactive Services Detection service (UI0Detect.exe) to provide backward compatibility with interactive services. (See Chapter 4 in Part 1 for more information on services.) Wininit then creates the service control manager (SCM) process (%SystemRoot%\System32\Services.exe), which loads all services and device drivers marked for auto-start, and the Local Security Authority subsystem (LSASS) process (%SystemRoot%\System32\Lsass.exe). Finally, it loads the local session manager (%SystemRoot%\System32\Lsm.exe). On session 1 and beyond, Winlogon runs instead and loads the registered credential providers for the system (by default, the Microsoft credential provider supports password-based and smartcard-based logons) into a child process called LogonUI (%SystemRoot%\System32\Logonui.exe), which is responsible for displaying the logon interface. (For more details on the startup sequence for Wininit, Winlogon, and LSASS, see the section “Winlogon Initialization” in Chapter 6 in Part 1.)

After the SCM initializes the auto-start services and drivers and a user has successfully logged on at the console, the SCM deems the boot successful. The registry’s last known good control set (as indicated by HKLM\SYSTEM\Select\LastKnownGood) is updated to match \CurrentControlSet.



Note Because noninteractive servers might never have an interactive logon, they might not get LastKnownGood updated to reflect the control set used for a successful boot. You can override the definition of a successful boot by setting HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk to 0, writing a custom boot verification program that calls the *NotifyBootConfigStatus* Windows API when a boot is successful, and entering the path to the verification program in HKLM\SYSTEM\CurrentControlSet\Control\BootVerificationProgram.

After launching the SCM, Winlogon waits for an interactive logon notification from the credential provider. When it receives a logon and validates the logon (a process for which you can find more information in the section “User Logon Steps” in Chapter 6 in Part 1), Winlogon loads the registry hive from the profile of the user logging on and maps it to HKCU. It then sets the user’s environment variables that are stored in HKCU\Environment and notifies the Winlogon notification packages registered in HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify that a logon has occurred.

Winlogon next starts the shell by launching the executable or executables specified in HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WinLogon\Userinit (with multiple executables separated by commas) that by default points at \Windows\System32\Userinit.exe. Userinit.exe performs the following steps:

1. Processes the user scripts specified in HKCU\Software\Policies\Microsoft\Windows\System\Scripts and the machine logon scripts in HKLM\SOFTWARE\Policies\Microsoft\Windows\System\Scripts. (Because machine scripts run after user scripts, they can override user settings.)
2. If Group Policy specifies a user profile quota, starts %SystemRoot%\System32\Proquota.exe to enforce the quota for the current user.
3. Launches the comma-separated shell or shells specified in HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell. If that value doesn’t exist, Userinit.exe launches the shell or shells specified in HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell, which is by default Explorer.exe.

Winlogon then notifies registered network providers that a user has logged on. The Microsoft network provider, Multiple Provider Router (%SystemRoot%\System32\Mpr.dll), restores the user’s persistent drive letter and printer mappings stored in HKCU\Network and HKCU\Printers, respectively. Figure 13-4 shows the process tree as seen in Process Monitor after a logon (using its boot logging capability). Note the Smss processes that are dimmed (meaning that they have since exited). These refer to the spawned copies that initialized each session.

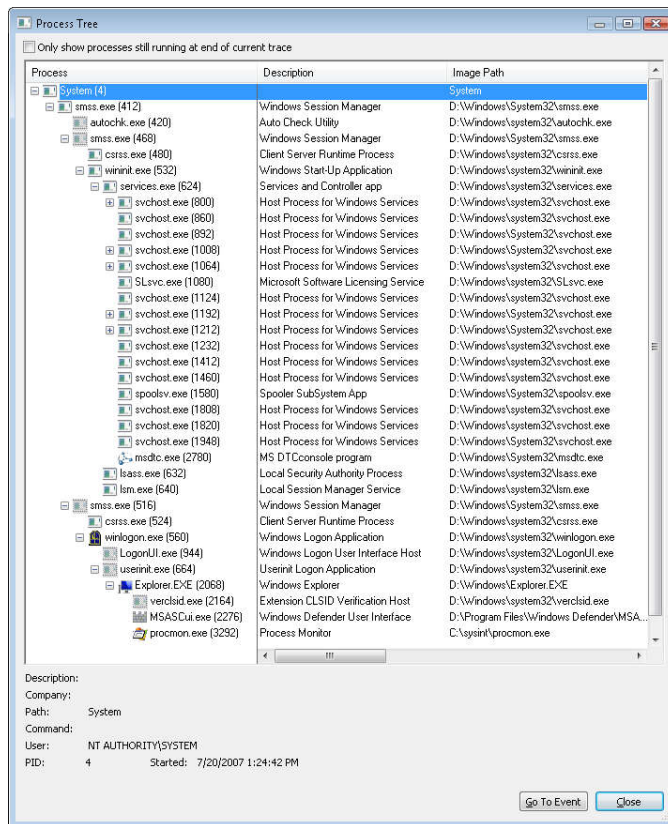


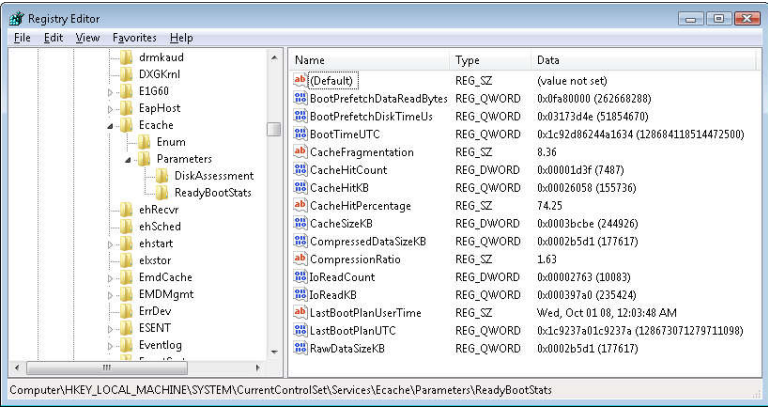
FIGURE 13-4 Process tree during login

ReadyBoot

Windows uses the standard logical boot-time prefetcher (described in Chapter 10) if the system has less than 700 MB of memory, but if the system has 700 MB or more of RAM, it uses an in-RAM cache to optimize the boot process. The size of the cache depends on the total RAM available, but it is large enough to create a reasonable cache and yet allow the system the memory it needs to boot smoothly.

After every boot, the ReadyBoost service (see Chapter 10 for information on ReadyBoost) uses idle CPU time to calculate a boot-time caching plan for the next boot. It analyzes file trace information from the five previous boots and identifies which files were accessed and where they are located on disk. It stores the processed traces in %SystemRoot%\Prefetch\Readyboot as .fx files and saves the caching plan under HKLM\SYSTEM\CurrentControlSet\Services\Rdyboost\Parameters in REG_BINARY values named for internal disk volumes they refer to.

The cache is implemented by the same device driver that implements ReadyBoost caching (Ecache.sys), but the cache's population is guided by the boot plan previously stored in the registry. Although the boot cache is compressed like the ReadyBoost cache, another difference between ReadyBoost and ReadyBoot cache management is that while in ReadyBoot mode, the cache is not encrypted. The ReadyBoost service deletes the cache 50 seconds after the service starts, or if other memory demands warrant it, and records the cache's statistics in HKLM\SYSTEM\CurrentControlSet\Services\Ecache\Parameters\ReadyBootStats, as shown in Figure 13-5.



Name	Type	Data
(Default)	REG_SZ	(value not set)
BootPrefetchDataReadBytes	REG_QWORD	0x0fa00000 (262688288)
BootPrefetchDiskTimeUs	REG_QWORD	0x03173d4e (51854670)
BootTimeUTC	REG_QWORD	0x1c92d86244a1634 (128684118514472500)
CacheFragmentation	REG_SZ	8.36
CacheHitCount	REG_DWORD	0x00001d3f (7487)
CacheHitKB	REG_QWORD	0x00026058 (155736)
CacheHitPercentage	REG_SZ	74.25
CacheSizeKB	REG_DWORD	0x0003bcbe (244926)
CompressedDataSizeKB	REG_QWORD	0x0002b5d1 (177617)
CompressionRatio	REG_SZ	1.63
IoReadCount	REG_DWORD	0x00002763 (10083)
IoReadKB	REG_QWORD	0x000397a0 (235424)
LastBootPlanUserTime	REG_SZ	Wed, Oct 01 08, 12:03:48 AM
LastBootPlanUTC	REG_QWORD	0x1c9237a01c9237a (128673071279711098)
RawDataSizeKB	REG_QWORD	0x0002b5d1 (177617)

FIGURE 13-5 ReadyBoot statistics

Images That Start Automatically

In addition to the Userinit and Shell registry values in Winlogon's key, there are many other registry locations and directories that default system components check and process for automatic process startup during the boot and logon processes. The Msconfig utility (%SystemRoot%\System32\Msconfig.exe) displays the images configured by several of the locations. The Autoruns tool, which you can download from Sysinternals and that is shown in Figure 13-6, examines more locations than Msconfig and displays more information about the images configured to automatically run. By default, Autoruns shows only the locations that are configured to automatically execute at least one image, but selecting the Include Empty Locations entry on the Options menu causes Autoruns to show all the locations it inspects. The Options menu also has selections to direct Autoruns to hide Microsoft entries, but you should always combine this option with Verify Image Signatures; otherwise, you risk hiding malicious programs that include false information about their company name information.

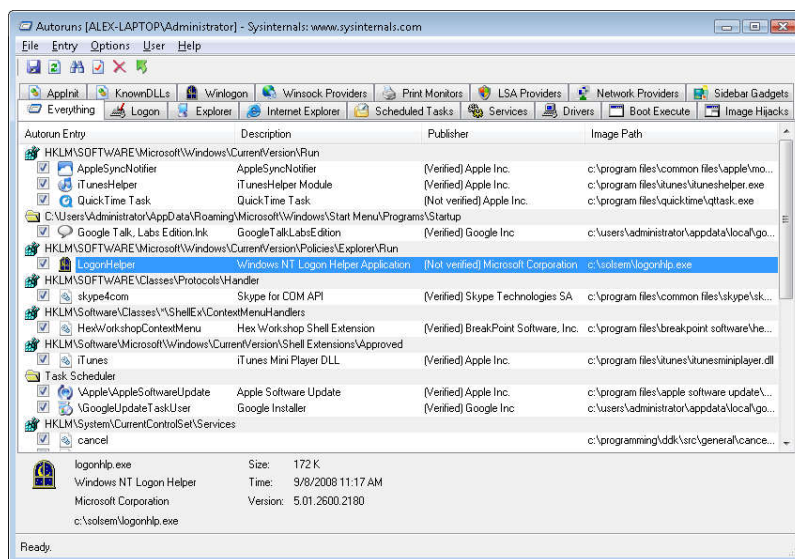


FIGURE 13-6 The Autoruns tool available from Sysinternals



EXPERIMENT: Autoruns

Many users are unaware of how many programs execute as part of their login. Original equipment manufacturers (OEMs) often configure their systems with add-on utilities that execute in the background using registry values or file system directories processed for automatic execution and so are not normally visible. See what programs are configured to start automatically on your computer by running the Autoruns utility from Sysinternals. Compare the list shown in Autoruns with that shown in Msconfig and identify any differences. Then ensure that you understand the purpose of each program.

Troubleshooting Boot and Startup Problems

This section presents approaches to solving problems that can occur during the Windows startup process as a result of hard disk corruption, file corruption, missing files, and third-party driver bugs. First we describe three Windows boot-problem recovery modes: last known good, safe mode, and Windows Recovery Environment (WinRE). Then we present common boot problems, their causes, and approaches to solving them. The solutions refer to last known good, safe mode, WinRE, and other tools that ship with Windows.

Last Known Good

Last known good (LKG) is a useful mechanism for getting a system that crashes during the boot process back to a bootable state. Because the system's configuration settings are stored in HKLM\SYSTEM\CurrentControlSet\Control and driver and service configuration is stored in HKLM\SYSTEM\CurrentControlSet\Services, changes to these parts of the registry can render a system unbootable. For example, if you install a device driver that has a bug that crashes the system during the boot, you can press the F8 key during the boot and select last known good from the resulting menu. The system marks the control set that it was using to boot the system as failed by setting the Failed value of HKLM\SYSTEM\Select and then changes HKLM\SYSTEM\Select\Current to the value stored in HKLM\SYSTEM\Select\LastKnownGood. It also updates the symbolic link HKLM\SYSTEM\CurrentControlSet to point at the LastKnownGood control set. Because the new driver's key is not present in the Services subkey of the LastKnownGood control set, the system will boot successfully.

Safe Mode

Perhaps the most common reason Windows systems become unbootable is that a device driver crashes the machine during the boot sequence. Because software or hardware configurations can change over time, latent bugs can surface in drivers at any time. Windows offers a way for an administrator to attack the problem: booting in *safe mode*. Safe mode is a boot configuration that consists of the minimal set of device drivers and services. By relying on only the drivers and services that are necessary for booting, Windows avoids loading third-party and other nonessential drivers that might crash.

When Windows boots, you press the F8 key to enter a special boot menu that contains the safe-mode boot options. You typically choose from three safe-mode variations: Safe Mode, Safe Mode With Networking, and Safe Mode With Command Prompt. Standard safe mode includes the minimum number of device drivers and services necessary to boot successfully. Networking-enabled safe mode adds network drivers and services to the drivers and services that standard safe mode includes. Finally, safe mode with command prompt is identical to standard safe mode except that Windows runs the Command Prompt application (Cmd.exe) instead of Windows Explorer as the shell when the system enables GUI mode.

Windows includes a fourth safe mode—Directory Services Restore mode—which is different from the standard and networking-enabled safe modes. You use Directory Services Restore mode to boot the system into a mode where the Active Directory service of a domain controller is offline and unopened. This allows you to perform repair operations on the database or restore it from backup media. All drivers and services, with the exception of the Active Directory service, load during a Directory Services Restore mode boot. In cases where you can't log on to a system because of Active Directory database corruption, this mode enables you to repair the corruption.

Driver Loading in Safe Mode

How does Windows know which device drivers and services are part of standard and networking-enabled safe mode? The answer lies in the HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot registry key. This key contains the Minimal and Network subkeys. Each subkey contains more subkeys

that specify the names of device drivers or services or of groups of drivers. For example, the `vga.sys` subkey identifies the VGA display device driver that the startup configuration includes. The VGA display driver provides basic graphics services for any PC-compatible display adapter. The system uses this driver as the safe-mode display driver in lieu of a driver that might take advantage of an adapter's advanced hardware features but that might also prevent the system from booting. Each subkey under the `SafeBoot` key has a default value that describes what the subkey identifies; the `vga.sys` subkey's default value is "Driver".

The `Boot file system` subkey has as its default value "Driver Group". When developers design a device driver's installation script (.inf file), they can specify that the device driver belongs to a driver group. The driver groups that a system defines are listed in the `List` value of the `HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder` key. A developer specifies a driver as a member of a group to indicate to Windows at what point during the boot process the driver should start. The `ServiceGroupOrder` key's primary purpose is to define the order in which driver groups load; some driver types must load either before or after other driver types. The `Group` value beneath a driver's configuration registry key associates the driver with a group.

Driver and service configuration keys reside beneath `HKLM\SYSTEM\CurrentControlSet\Services`. If you look under this key, you'll find the `VgaSave` key for the VGA display device driver, which you can see in the registry is a member of the `Video Save` group. Any file system drivers that Windows requires for access to the Windows system drive are automatically loaded as if part of the `Boot file system` group. Other file system drivers are part of the `File system` group, which the standard and networking-enabled safe-mode configurations also include.

When you boot into a safe-mode configuration, the boot loader (`Winload`) passes an associated switch to the kernel (`Ntldr.exe`) as a command-line parameter, along with any switches you've specified in the BCD for the installation you're booting. If you boot into any safe mode, `Winload` sets the `safeboot` BCD option with a value describing the type of safe mode you select. For standard safe mode, `Winload` sets `minimal`, and for networking-enabled safe mode, it adds `network`. `Winload` adds `minimal` and sets `safebootalternateshell` for safe mode with command prompt and `dsrepair` for Directory Services Restore mode.

The Windows kernel scans boot parameters in search of the safe-mode switches early during the boot, during the `InitSafeBoot` function, and sets the internal variable `InitSafeBootMode` to a value that reflects the switches the kernel finds. The kernel writes the `InitSafeBootMode` value to the registry value `HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\OptionValue` so that user-mode components, such as the SCM, can determine what boot mode the system is in. In addition, if the system is booting in safe mode with command prompt, the kernel sets the `HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\UseAlternateShell` value to 1. The kernel records the parameters that `Winload` passes to it in the value `HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions`.

When the I/O manager kernel subsystem loads device drivers that `HKLM\SYSTEM\CurrentControlSet\Services` specifies, the I/O manager executes the function `IoLoadDriver`. When the Plug and Play manager detects a new device and wants to dynamically load the device driver for the detected device, the Plug and Play manager executes the function `PnpCallDriverAddDevice`. Both these functions call the function `IoSafebootDriverLoad` before they load the driver in question.

lopSafebootDriverLoad checks the value of *InitSafeBootMode* and determines whether the driver should load. For example, if the system boots in standard safe mode, *lopSafebootDriverLoad* looks for the driver's group, if the driver has one, under the Minimal subkey. If *lopSafebootDriverLoad* finds the driver's group listed, *lopSafebootDriverLoad* indicates to its caller that the driver can load. Otherwise, *lopSafebootDriverLoad* looks for the driver's name under the Minimal subkey. If the driver's name is listed as a subkey, the driver can load. If *lopSafebootDriverLoad* can't find the driver group or driver name subkeys, the driver will not be loaded. If the system boots in networking-enabled safe mode, *lopSafebootDriverLoad* performs the searches on the Network subkey. If the system doesn't boot in safe mode, *lopSafebootDriverLoad* lets all drivers load.



Note An exception exists regarding the drivers that safe mode excludes from a boot: Winload, rather than the kernel, loads any drivers with a Start value of 0 in their registry key, which specifies loading the drivers at boot time. Winload doesn't check the SafeBoot registry key because it assumes that any driver with a Start value of 0 is required for the system to boot successfully. Because Winload doesn't check the SafeBoot registry key to identify which drivers to load, Winload loads all boot-start drivers (and later Ntoskrnl starts them).

Safe-Mode-Aware User Programs

When the service control manager (SCM) user-mode component (which *Services.exe* implements) initializes during the boot process, the SCM checks the value of HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\OptionValue to determine whether the system is performing a safe-mode boot. If so, the SCM mirrors the actions of *lopSafebootDriverLoad*. Although the SCM processes the services listed under HKLM\SYSTEM\CurrentControlSet\Services, it loads only services that the appropriate safe-mode subkey specifies by name. You can find more information on the SCM initialization process in the section "Services" in Chapter 4 in Part 1.

Userinit, the component that initializes a user's environment when the user logs on (%SystemRoot%\System32\Userinit.exe), is another user-mode component that needs to know whether the system is booting in safe mode. It checks the value of HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\UseAlternateShell. If this value is set, Userinit runs the program specified as the user's shell in the value HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell rather than executing *Explorer.exe*. Windows writes the program name *Cmd.exe* to the *AlternateShell* value during installation, making the Windows command prompt the default shell for safe mode with command prompt. Even though the command prompt is the shell, you can type **Explorer.exe** at the command prompt to start Windows Explorer, and you can run any other GUI program from the command prompt as well.

How does an application determine whether the system is booting in safe mode? By calling the Windows *GetSystemMetrics(SM_CLEANBOOT)* function. Batch scripts that need to perform certain operations when the system boots in safe mode look for the *SAFEBOOT_OPTION* environment variable because the system defines this environment variable only when booting in safe mode.

Boot Logging in Safe Mode

When you direct the system to boot into safe mode, Winload hands the string specified by the *bootlog* option to the Windows kernel as a parameter, together with the parameter that requests safe mode. When the kernel initializes, it checks for the presence of the *bootlog* parameter whether or not any safe-mode parameter is present. If the kernel detects a boot log string, the kernel records the action the kernel takes on every device driver it considers for loading. For example, if *lopSafeboot-DriverLoad* tells the I/O manager not to load a driver, the I/O manager calls *lopBootLog* to record that the driver wasn't loaded. Likewise, after *lopLoadDriver* successfully loads a driver that is part of the safe-mode configuration, *lopLoadDriver* calls *lopBootLog* to record that the driver loaded. You can examine boot logs to see which device drivers are part of a boot configuration.

Because the kernel wants to avoid modifying the disk until Chkdsk executes, late in the boot process, *lopBootLog* can't simply dump messages into a log file. Instead, *lopBootLog* records messages in the HKLM\SYSTEM\CurrentControlSet\BootLog registry value. As the first user-mode component to load during a boot, the Session Manager (%SystemRoot%\System32\Smss.exe) executes Chkdsk to ensure the system drives' consistency and then completes registry initialization by executing the *NtInitializeRegistry* system call. The kernel takes this action as a cue that it can safely open a log file on the disk, which it does, invoking the function *lopCopyBootLogRegistryToFile*. This function creates the file Ntbtlog.txt in the Windows system directory (%SystemRoot%) and copies the contents of the BootLog registry value to the file. *lopCopyBootLogRegistryToFile* also sets a flag for *lopBootLog* that lets *lopBootLog* know that writing directly to the log file, rather than recording messages in the registry, is now OK. The following output shows the partial contents of a sample boot log:

```
Microsoft (R) Windows (R) Version 6.1 (Build 7601)
10  4 2012 09:04:53.375
Loaded driver \SystemRoot\system32\ntkrnlpa.exe
Loaded driver \SystemRoot\system32\hal.dll
Loaded driver \SystemRoot\system32\kdcom.dll
Loaded driver \SystemRoot\system32\mcpupdate_GenuineIntel.dll
Loaded driver \SystemRoot\system32\PSHED.dll
Loaded driver \SystemRoot\system32\BOOTVID.dll
Loaded driver \SystemRoot\system32\CLFS.SYS
Loaded driver \SystemRoot\system32\CI.dll
Loaded driver \SystemRoot\system32\drivers\Wdf01000.sys
Loaded driver \SystemRoot\system32\drivers\WDFLDR.SYS
Loaded driver \SystemRoot\system32\drivers\acpi.sys
Loaded driver \SystemRoot\system32\drivers\WMILIB.SYS
Loaded driver \SystemRoot\system32\drivers\msisadrv.sys
Loaded driver \SystemRoot\system32\drivers\pci.sys
Loaded driver \SystemRoot\system32\drivers\volmgr.sys
Loaded driver \SystemRoot\system32\DRIVERS\compbatt.sys
Loaded driver \SystemRoot\system32\DRIVERS\BATTC.SYS
Loaded driver \SystemRoot\System32\drivers\mountmgr.sys
Loaded driver \SystemRoot\system32\drivers\intelide.sys
Loaded driver \SystemRoot\system32\drivers\PCIIDEX.SYS
Loaded driver \SystemRoot\system32\DRIVERS\pciide.sys
Loaded driver \SystemRoot\System32\drivers\volmgrx.sys
Loaded driver \SystemRoot\system32\drivers\atapi.sys
Loaded driver \SystemRoot\system32\drivers\ataport.SYS
Loaded driver \SystemRoot\system32\drivers\fltmgr.sys
```

```

Loaded driver \SystemRoot\system32\drivers\fileinfo.sys
...
Did not load driver @battery.inf,%acpi\acpi0003.devicedesc%;Microsoft AC Adapter
Did not load driver @battery.inf,%acpi\pnp0c0a.devicedesc%;Microsoft ACPI-Compliant
Control Method Battery
Did not load driver @oem46.inf,%nvidia_g71.dev_0297.1%;NVIDIA GeForce Go 7950 GTX
Did not load driver @oem5.inf,%nic_mpciex%;Intel(R) PRO/Wireless 3945ABG Network Connection
Did not load driver @netb57vx.inf,%bcm5750a1c1nahkd%;Broadcom NetXtreme 57xx Gigabit Controller
Did not load driver @sdbus.inf,%pci\cc_080501.devicedesc%;SDA Standard Compliant
SD Host Controller
...

```

Windows Recovery Environment (WinRE)

Safe mode is a satisfactory fallback for systems that become unbootable because a device driver crashes during the boot sequence, but in some situations a safe-mode boot won't help the system boot. For example, if a driver that prevents the system from booting is a member of a Safe group, safe-mode boots will fail. Another example of a situation in which safe mode won't help the system boot is when a third-party driver, such as a virus scanner driver, that loads at the boot prevents the system from booting. (Boot-start drivers load whether or not the system is in safe mode.) Other situations in which safe-mode boots will fail are when a system module or critical device driver file that is part of a safe-mode configuration becomes corrupt or when the system drive's Master Boot Record (MBR) is damaged.

You can get around these problems by using the Windows Recovery Environment. The Windows Recovery Environment provides an assortment of tools and automated repair technologies to automatically fix the most common startup problems. It includes five main tools:

- **Startup Repair** An automated tool that detects the most common Windows startup problems and automatically attempts to repair them.
- **System Restore** Allows restoring to a previous restore point in cases in which you cannot boot the Windows installation to do so, even in safe mode.
- **System Image Recover** Called Complete PC Restore, as well as ASR (Automated System Recovery), in previous versions of Windows, this restores a Windows installation from a complete backup, not just a system restore point, which might not contain all damaged files and lost data.
- **Windows Memory Diagnostic Tool** Performs memory diagnostic tests that check for signs of faulty RAM. Faulty RAM can be the reason for random kernel and application crashes and erratic system behavior.
- **Command Prompt** For cases where troubleshooting or repair requires manual intervention (such as copying files from another drive or manipulating the BCD), you can use the command prompt to have a full Windows shell that can launch almost any Windows program (as long as the required dependencies can be satisfied)—unlike the Recovery Console on earlier versions of Windows, which only supported a limited set of specialized commands.

When you boot a system from the Windows CD or boot disks, Windows Setup gives you the choice of installing Windows or repairing an existing installation. If you choose to repair an installation, the system displays a dialog box called System Recovery Options, shown in Figure 13-7.

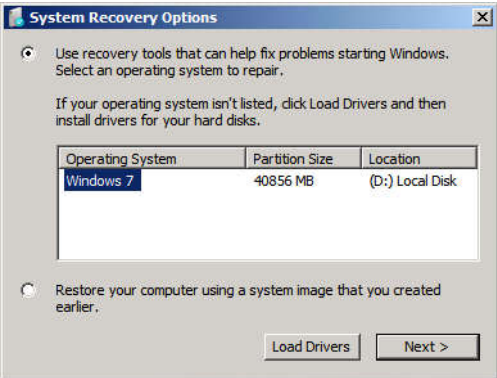


FIGURE 13-7 The System Recovery Options dialog box

Newer versions of Windows also install WinRE to a recovery partition on a clean system installation. On these systems, you can access WinRE by using the F8 option to access advanced boot options during Bootmgr execution. If you see an option Repair Your Computer, your machine has a local hard disk copy. If for some reason yours does not, you can follow the instructions at the Microsoft WinRE blog (<http://blogs.msdn.com/winre>) to install WinRE on the hard disk yourself from your Windows installation media and Windows Automated Installation Kit (AIK).

If you select the first option, WinRE will then display the dialog box in Figure 13-8, which has the various recovery options. Choosing the second option, on the other hand, is equivalent to the System Image Recovery option shown in Figure 13-8.

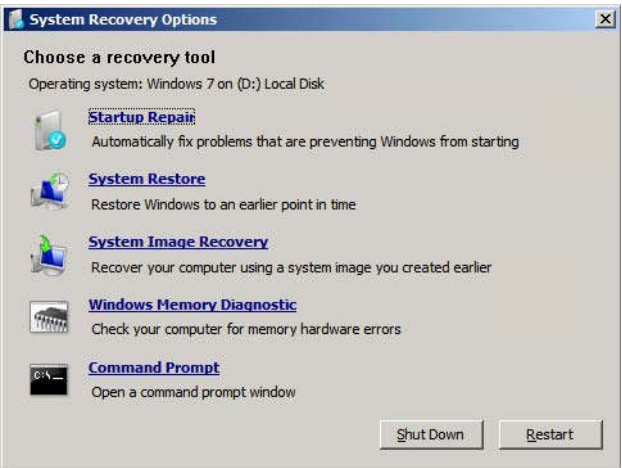


FIGURE 13-8 The Advanced System Recovery Options dialog box

Additionally, if your system failed to boot as the result of damaged files or for any other reason that Winload can understand, it instructs Bootmgr to automatically start WinRE at the next reboot cycle. Instead of the dialog box shown in Figure 13-8, the recovery environment will automatically launch the Startup Repair tool, shown in Figure 13-9.

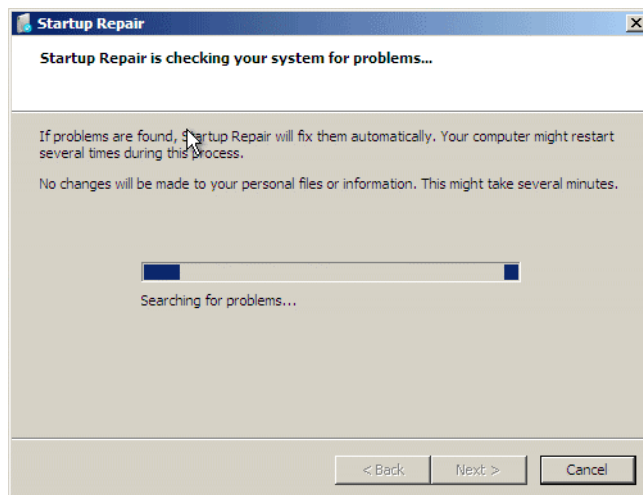


FIGURE 13-9 The Startup Repair tool

At the end of the scan and repair cycle, the tool will automatically attempt to fix any damage found, including replacing system files from the installation media. You can click the details link to see information about the damage that was fixed. For example, in Figure 13-10, the Startup Repair tool fixed a damaged boot sector.

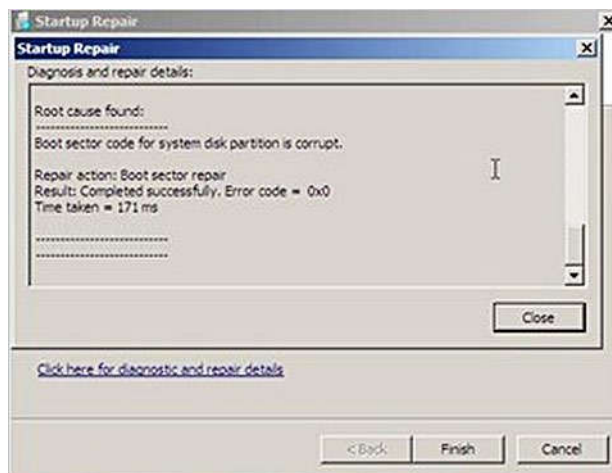


FIGURE 13-10 Details view of the Startup Repair tool

If the Startup Repair tool cannot automatically fix the damage, or if you cancel the operation, you'll get a chance to try other methods and the System Recovery Options dialog box will be displayed.

Boot Status File

Windows uses a *boot status file* (%SystemRoot%\Bootstat.dat) to record the fact that it has progressed through various stages of the system life cycle, including boot and shutdown. This allows the Boot Manager, Windows loader, and the Startup Repair tool to detect abnormal shutdown or a failure to shut down cleanly and offer the user recovery and diagnostic boot options, like Last Known Good and Safe Mode. This binary file contains information through which the system reports the success of the following phases of the system life cycle:

- Boot (the definition of a successful boot is the same as the one used for determining Last Known Good status, which was described earlier)
- Shutdown
- Resume from hibernate or suspend

The boot status file also indicates whether a problem was detected the last time the user attempted to boot the operating system and the recovery options shown, indicating that the user has been made aware of the problem and taken action. Runtime Library APIs (Rtl) in Ntdll.dll contain the private interfaces that Windows uses to read from and write to the file. Like the BCD, it cannot be edited by users.

Solving Common Boot Problems

This section describes problems that can occur during the boot process, describing their symptoms, what caused them, and approaches to solving them. To help you locate a problem that you might encounter, they are organized according to the place in the boot at which they occur. Note that for most of these problems, you should be able to simply boot into the Windows Recovery Environment and allow the Startup Repair tool to scan your system and perform any automated repair tasks.

MBR Corruption

- **Symptoms** A system that has Master Boot Record (MBR) corruption will execute the BIOS power-on self test (POST), display BIOS version information or OEM branding, switch to a black screen, and then hang. Depending on the type of corruption the MBR has experienced, you might see one of the following messages: "Invalid partition table", "Error loading operating system", or "Missing operating system".
- **Cause** The MBR can become corrupt because of hard-disk errors, disk corruption as a result of a driver bug while Windows is running, or intentional scrambling as a result of a virus.
- **Resolution** Boot into the Windows Recovery Environment, choose the Command Prompt option, and then execute the *bootrec /fixmbr* command. This command replaces the executable code in the MBR.

Boot Sector Corruption

- **Symptoms** Boot sector corruption can look like MBR corruption, where the system hangs after BIOS POST at a black screen, or you might see the messages “A disk read error occurred”, “BOOTMGR is missing”, or “BOOTMGR is compressed” displayed on a black screen.
- **Cause** The boot sector can become corrupt because of hard-disk errors, disk corruption as a result of a driver bug while Windows is running, or intentional scrambling as a result of a virus.
- **Resolution** Boot into the Windows Recovery Environment, choose the Command Prompt option, and then execute the *bootrec /fixboot* command. This command rewrites the boot sector of the volume that you specify. You should execute the command on both the system and boot volumes if they are different.

BCD Misconfiguration

- **Symptom** After BIOS POST, you’ll see a message that begins “Windows could not start because of a computer disk hardware configuration problem”, “Could not read from selected boot disk”, or “Check boot path and disk hardware”.
- **Cause** The BCD has been deleted, become corrupt, or no longer references the boot volume because the addition of a partition has changed the name of the volume.
- **Resolution** Boot into the Windows Recovery Environment, choose the Command Prompt option, and then execute the *bootrec /scanos* and *bootrec /rebuildbcd* commands. These commands will scan each volume looking for Windows installations. When they discover an installation, they will ask you whether they should add it to the BCD as a boot option and what name should be displayed for the installation in the boot menu. For other kinds of BCD-related damage, you can also use Bcdedit.exe to perform tasks such as building a new BCD from scratch or cloning an existing good copy.

System File Corruption

- **Symptoms** There are several ways the corruption of system files—which include executables, drivers, or DLLs—can manifest. One way is with a message on a black screen after BIOS POST that says, “Windows could not start because the following file is missing or corrupt”, followed by the name of a file and a request to reinstall the file. Another way is with a blue screen crash during the boot with the text, “STOP: 0xC0000135 {Unable to Locate Component}”.
- **Causes** The volume on which a system file is located is corrupt or one or more system files have been deleted or become corrupt.
- **Resolution** Boot into the Windows Recovery Environment, choose the Command Prompt option, and then execute the *chkdsk* command. Chkdsk will attempt to repair volume corruption. If Chkdsk does not report any problems, obtain a backup copy of the system file in question. One place to check is in the %SystemRoot%\winsxs\Backup directory, in which Windows places copies of many system files for access by Windows Resource Protection. (See the “Windows Resource Protection” sidebar.) If you cannot find a copy of the file there, see if

you can locate a copy from another system in the network. Note that the backup file must be from the same service pack or hotfix as the file that you are replacing.

In some cases, multiple system files are deleted or become corrupt, so the repair process can involve multiple reboots and boot failures as you repair the files one by one. If you believe the system file corruption to be extensive, you should consider restoring the system from a backup image, such as one generated by Windows Backup and Restore or from a system restore point.

When you run Backup and Restore (located in the Maintenance folder on the Start menu), you can generate a System Image Recovery image, which includes all the files on the system and boot volumes, plus a floppy disk on which it stores information about the system's disks and volumes. To restore a system from such an image, boot from the Windows setup media and select the appropriate option when prompted (or use the recovery environment shown earlier).

If you do not have a backup from which to restore, a last resort is to execute a Windows repair install: boot from the Windows setup media, and follow the wizard as if you were going to perform a new installation. The wizard will ask you whether you want to perform a repair or fresh install. When you tell it that you want to repair, Setup reinstalls all system files, leaving your application data and registry settings intact.

Windows Resource Protection

To preserve the integrity of the many components involved in the boot process, as well as other critical Windows files, libraries, and applications, Windows implements a technology called *Windows Resource Protection* (WRP). WRP is implemented through access control lists (ACLs) that protect critical system files on the machine. It is also exposed through an API (located in %SystemRoot%\System32\Sfc.dll and %SystemRoot%\System32\Sfc_os.dll) that can be accessed by the Sfc.exe utility to manually check a file for corruption and restore it.

WRP will also protect entire critical folders if required, even locking down the folder so that it is inaccessible by administrators (without modifying the access control list on the folder). The only supported way to modify WRP-protected files is through the Windows Modules Installer service, which can run under the TrustedInstaller account. This service is used for the installation of patches, service packs, hotfixes, and Windows Update. This account has access to the various protected files and is trusted by the system (as its name implies) to modify critical files and replace them. WRP also protects critical registry keys, and it may even lock entire registry trees if all the values and subkeys are considered to be critical.

WRP sets the ACL on protected files, directories, or registry keys such that only the TrustedInstaller account is able to modify or delete these files. Application developers can use the *Sfcls-FileProtected* or *SfclsKeyProtected* APIs to check whether a file or registry key is locked down.

For backward compatibility, certain installers are considered *well-known*—an application compatibility shim exists that will suppress the “access denied” error that certain installers would receive while attempting to modify WRP-protected resources. Instead, the installer receives a fake “success” code, but the modification isn't made. This virtualization is similar to the User

Access Control (UAC) virtualization technology discussed in Chapter 6 in Part 1, but it applies to write operations as well. It applies if the following are true:

- The application is a *legacy* application, meaning that it does not contain a manifest file compatible with the *requestedExecutionLevel* value set.
- The application is trying to modify a WRP-protected resource (the file or registry key contains the *TrustedInstaller* SID).
- The application is being run under an administrator account (always true on systems with UAC enabled because of automatic installer program detection).

WRP copies files that are needed to restart Windows to the *cache directory* located at %SystemRoot%\winsxs\Backup. Critical files that are not needed to restart Windows are not copied to the cache directory. The size of the cache directory and the list of files copied to the cache cannot be modified. To recover a file from the cache directory, you can use the System File Checker (Sfc.exe) tool, which can scan your system for modified protected files and restore them from a good copy.

System Hive Corruption

- **Symptoms** If the System registry hive (which is discussed along with hive files in the section “The Registry” in Chapter 4 in Part 1) is missing or corrupted, Winload will display the message “Windows could not start because the following file is missing or corrupt: \WINDOWS\SYSTEM32\CONFIG\SYSTEM”, on a black screen after the BIOS POST.
- **Causes** The System registry hive, which contains configuration information necessary for the system to boot, has become corrupt or has been deleted.
- **Resolution** Boot into the Windows Recovery Environment, choose the Command Prompt option, and then execute the *chkdsk* command. If the problem is not corrected, obtain a backup of the System registry hive. Windows makes copies of the registry hives every 12 hours (keeping the immediately previous copy with a .OLD extension) in a folder called %SystemRoot%\System32\Config\RegBack, so copy the file named System to %SystemRoot%\System32\Config.

If System Restore is enabled (System Restore is discussed in Chapter 12, “File System”), you can often obtain a more recent backup of the registry hives, including the System hive, from the most recent restore point. You can choose System Restore from the Windows Recovery Environment to restore your registry from the last restore point.

Post-Splash Screen Crash or Hang

- **Symptoms** Problems that occur after the Windows splash screen displays, the desktop appears, or you log on fall into this category and can appear as a blue screen crash or a hang,

where the entire system is frozen or the mouse cursor tracks the mouse but the system is otherwise unresponsive.

- **Causes** These problems are almost always a result of a bug in a device driver, but they can sometimes be the result of corruption of a registry hive other than the System hive.
- **Resolution** You can take several steps to try and correct the problem. The first thing you should try is the last known good configuration. Last known good (LKG), which is described earlier in this chapter and in the “Services” section of Chapter 4 in Part 1, consists of the registry control set that was last used to boot the system successfully. Because a control set includes core system configuration and the device driver and services registration database, using a version that does not reflect changes or newly installed drivers or services might avoid the source of the problem. You access last known good by pressing the F8 key early in the boot process to access the same menu from which you can boot into safe mode.

As stated earlier in the chapter, when you boot into LKG, the system saves the control set that you are avoiding and labels it as the *failed* control set. You can leverage the failed control set in cases where LKG makes a system bootable to determine what was causing the system to fail to boot by exporting the contents of the current control set of the successful boot and the failed control set to .reg files. You do this by using Regedit’s export functionality, which you access under the File menu:

1. Run Regedit, and select HKLM\SYSTEM\CurrentControlSet.
2. Select Export from the File menu, and save to a file named good.reg.
3. Open HKLM\SYSTEM\Select, read the value of Failed, and select the subkey named HKLM\SYSTEM\ControlXXX, where XXX is the value of Failed.
4. Export the contents of the control set to bad.reg.
5. Use WordPad (which is found under Accessories on the Start menu) to globally replace all instances of CurrentControlSet in good.reg with ControlSet.
6. Use WordPad to change all instances of ControlXXX (replacing XXX with the value of the Failed control set) in bad.reg with ControlSet.
7. Run Windiff from the Support Tools, and compare the two files.

The differences between a failed control set and a good one can be numerous, so you should focus your examination on changes beneath the Control subkey as well as under the Parameters subkeys of drivers and services registered in the Services subkey. Ignore changes made to Enum subkeys of driver registry keys in the Services branch of the control set.

If the problem you’re experiencing is caused by a driver or service that was present on the system since before the last successful boot, LKG will not make the system bootable. Similarly, if a problematic configuration setting changed outside the control set or was made before the last successful boot, LKG will not help. In those cases, the next option to try is safe mode (described earlier in this section). If the system boots successfully in safe mode and you know what particular driver was causing the normal boot to fail, you can disable the driver by using the Device Manager (accessible from

the System Control Panel item). To do so, select the driver in question and choose Disable from the Action menu. If you recently updated the driver, and believe that the update introduced a bug, you can choose to roll back the driver to its previous version instead, also with the Device Manager. To restore a driver to its previous version, double-click on the device to open its Properties dialog box and click Roll Back Driver on the Driver tab.

On systems with System Restore enabled, an option when LKG fails is to roll back all system state (as defined by System Restore) to a previous point in time. Safe mode detects the existence of restore points, and when they are present it will ask you whether you want to log on to the installation to perform a manual diagnosis and repair or launch the System Restore Wizard. Using System Restore to make a system bootable again is attractive when you know the cause of a problem and want the repair to be automatic or when you don't know the cause but do not want to invest time to determine the cause.

If System Restore is not an option or you want to determine the cause of a crash during the normal boot and the system boots successfully in safe mode, attempt to obtain a boot log from the unsuccessful boot by pressing F8 to access the special boot menu and choosing the boot logging option. As described earlier in this chapter, Session Manager (%SystemRoot%\System32\Smss.exe) saves a log of the boot that includes a record of device drivers that the system loaded and chose not to load to %SystemRoot%\ntbtlog.txt, so you'll obtain a boot log if the crash or hang occurs after Session Manager initializes. When you reboot into safe mode, the system appends new entries to the existing boot log. Extract the portions of the log file that refer to the failed attempt and safe-mode boots into separate files. Strip out lines that contain the text "Did not load driver", and then compare them with a text comparison tool such as Windiff. One by one, disable the drivers that loaded during the normal boot but not in the safe-mode boot until the system boots successfully again. (Then reenable the drivers that were not responsible for the problem.)

If you cannot obtain a boot log from the normal boot (for instance, because the system is crashing before Session Manager initializes), if the system also crashes during the safe-mode boot, or if a comparison of boot logs from the normal and safe-mode boots do not reveal any significant differences (for example, when the driver that's crashing the normal boot starts after Session Manager initializes), the next tool to try is Driver Verifier combined with crash dump analysis. (See Chapter 14, "Crash Dump Analysis," for more information on both these topics.)

Shutdown

If someone is logged on and a process initiates a shutdown by calling the Windows *ExitWindowsEx* function, a message is sent to that session's Csrss instructing it to perform the shutdown. Csrss in turn impersonates the caller and sends an RPC message to Winlogon, telling it to perform a system shutdown. Winlogon then impersonates the currently logged-on user (who might or might not have the same security context as the user who initiated the system shutdown) and calls *ExitWindowsEx* with

some special internal flags. Again this call causes a message to be sent to the Csrss process inside that session, requesting a system shutdown.

This time, Csrss sees that the request is from Winlogon and loops through all the processes in the logon session of the interactive user (again, not the user who requested a shutdown) in reverse order of their *shutdown level*. A process can specify a shutdown level, which indicates to the system when it wants to exit with respect to other processes, by calling *SetProcessShutdownParameters*. Valid shutdown levels are in the range 0 through 1023, and the default level is 640. Explorer, for example, sets its shutdown level to 2 and Task Manager specifies 1. For each process that owns a top-level window, Csrss sends the WM_QUERYENDSESSION message to each thread in the process that has a Windows message loop. If the thread returns TRUE, the system shutdown can proceed. Csrss then sends the WM_ENDSESSION Windows message to the thread to request it to exit. Csrss waits the number of seconds defined in HKCU\Control Panel\Desktop\HungAppTimeout for the thread to exit. (The default is 5,000 milliseconds.)

If the thread doesn't exit before the timeout, Csrss fades out the screen and displays the hung-program screen shown in Figure 13-11. (You can disable this screen by creating the registry value HKCU\Control Panel\Desktop\AutoEndTasks and setting it to 1.) This screen indicates which programs are currently running and, if available, their current state. Windows indicates which program isn't shutting down in a timely manner and gives the user a choice of either killing the process or aborting the shutdown. (There is no timeout on this screen, which means that a shutdown request could wait forever at this point.) Additionally, third-party applications can add their own specific information regarding state—for example, a virtualization product could display the number of actively running virtual machines.

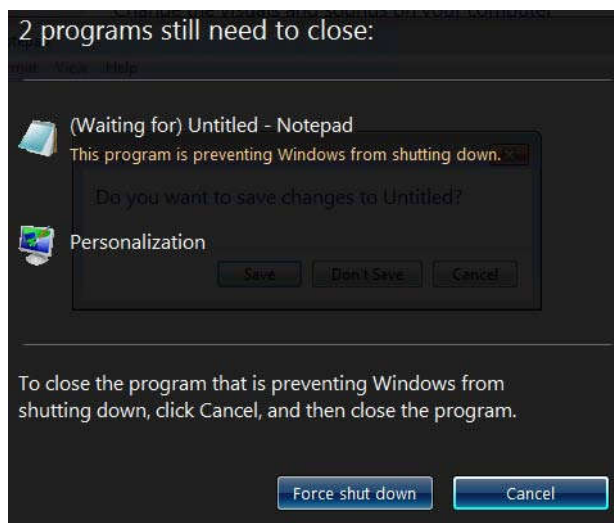


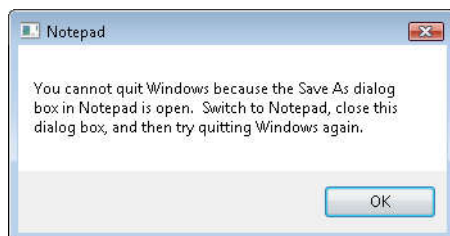
FIGURE 13-11 Hung program screen



EXPERIMENT: Witnessing the HungAppTimeout

You can see the use of the HungAppTimeout registry value by running Notepad, entering text into its editor, and then logging off. After the amount of time specified by the HungAppTimeout registry value has expired, Csrss.exe presents a prompt that asks you whether or not you want to end the Notepad process, which has not exited because it's waiting for you to tell it whether or not to save the entered text to a file. If you click the Cancel button, Csrss.exe aborts the shutdown.

As a second experiment, if you try shutting down again (with Notepad's query dialog box still open), Notepad will display its own message box to inform you that shutdown cannot cleanly proceed. However, this dialog box is merely an informational message to help users—Csrss.exe will still consider that Notepad is “hung” and display the user interface to terminate unresponsive processes.



If the thread does exit before the timeout, Csrss continues sending the WM_QUERYENDSESSION/WM_ENDSESSION message pairs to the other threads in the process that own windows. Once all the threads that own windows in the process have exited, Csrss terminates the process and goes on to the next process in the interactive session.

If Csrss finds a console application, it invokes the console control handler by sending the CTRL_LOGOFF_EVENT event. (Only service processes receive the CTRL_SHUTDOWN_EVENT event on shutdown.) If the handler returns FALSE, Csrss kills the process. If the handler returns TRUE or doesn't respond by the number of seconds defined by HKCU\Control Panel\Desktop\WaitToKillAppTimeout (the default is 20,000 milliseconds), Csrss displays the hung-program screen shown in Figure 13-11.

Next, Winlogon calls *ExitWindowsEx* to have Csrss terminate any COM processes that are part of the interactive user's session.

At this point, all the processes in the interactive user's session have been terminated. Wininit next calls *ExitWindowsEx*, which this time executes within the system process context. This causes Wininit to send a message to the Csrss part of session 0, where the services live. Csrss then looks at all the processes belonging to the system context and performs and sends the WM_QUERYENDSESSION/WM_ENDSESSION messages to GUI threads (as before). Instead of sending CTRL_LOGOFF_EVENT, however, it sends CTRL_SHUTDOWN_EVENT to console applications that have registered control handlers. Note that the SCM is a console program that does register a control handler. When it

receives the shutdown request, it in turn sends the service shutdown control message to all services that registered for shutdown notification. For more details on service shutdown (such as the shutdown timeout Csrss uses for the SCM), see the “Services” section in Chapter 4 in Part 1.

Although Csrss performs the same timeouts as when it was terminating the user processes, it doesn’t display any dialog boxes and doesn’t kill any processes. (The registry values for the system process timeouts are taken from the default user profile.) These timeouts simply allow system processes a chance to clean up and exit before the system shuts down. Therefore, many system processes are in fact still running when the system shuts down, such as Smss, Wininit, Services, and LSASS.

Once Csrss has finished its pass notifying system processes that the system is shutting down, Winlogon finishes the shutdown process by calling the executive subsystem function *NtShutdownSystem*. This function calls the function *PoSetSystemPowerState* to orchestrate the shutdown of drivers and the rest of the executive subsystems (Plug and Play manager, power manager, executive, I/O manager, configuration manager, and memory manager).

For example, *PoSetSystemPowerState* calls the I/O manager to send 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 Windows exits. The stacks of worker threads are swapped in, 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 I/O manager is called a second time to inform the file system drivers that the system is shutting down. System shutdown ends in the power manager. The action the power manager takes depends on whether the user specified a shutdown, a reboot, or a power down.

Conclusion

In this chapter, we’ve examined the detailed steps involved in starting and shutting down Windows (both normally and in error cases). We’ve examined the overall structure of Windows and the core system mechanisms that get the system going, keep it running, and eventually shut it down. The final chapter of this book explains how to deal with an unusual type of shutdown: system crashes.