The Windows NT 6 boot process

You've come to this page because you've asked a question similar to the following:

> What is the Windows NT version 6 ("Windows Vista") boot process ?

This is the Frequently Given Answer to such questions.

Up to the point that the Windows NT 6 boot manager is loaded, the Windows NT 6 bootstrap process differs between systems that use EFI machine firmware and systems that use IBM PC compatible machine firmware. From that point onwards, the process is the same: the Microsoft boot manager loads and runs the Windows NT 6 boot loader, which loads and runs the Windows NT 6 kernel, which loads and runs the first user process.

How Windows NT 6's boot manager is loaded on systems that use EFI machine firmwares

On machines with EFI machine firmwares, the firmware is required to contain a boot manager, that loads and runs an EFI executable program, which is either a standalone utility program or an operating boot loader program. Microsoft's installation program adds an single entry to the EFI boot manager menu entitled "Windows Boot Manager" that names "`\EFI\Microsoft\Boot\Bootmgfw.efi`" as the EFI executable program to be run when that option is selected from the boot manager.

EFI executables are 32-bit PE-format executable programs that expect to run in protected mode, using the flat memory model and without paging enabled. The firmware itself places the processor in this mode before invoking any boot programs.

"`\EFI\Microsoft\Boot\Bootmgfw.efi`" is in fact another boot manager. Not content with simply using the EFI boot manager as designed, Microsoft employs its own boot manager as well.

This results in two boot managers being run on EFI systems. (As explained later, everything that Microsoft's boot manager does could have been done using the EFI boot manager. There is no real need for the second Microsoft boot manager on EFI systems.) Microsoft configures the EFI boot manager with a timeout of 2 seconds in order to, in Microsoft's words, "make it easier" for users. One has to think, though, that not reinventing this particular wheel in the first place and sticking with the EFI boot manager would also have made things easier for users, too.

How Windows NT 6's boot manager is loaded on systems that use IBM PC compatible machine firmwares

On machines with IBM PC compatible firmwares, the firmware enumerates its list of bootable devices (stored in NVRAM and configurable via the "BIOS Setup" utility), attempting to load a boot sector off each device in turn. This boot sector is either a Volume Boot Record or a Master Boot Record. For MBRs, in the conventional case, the bootstrap code within the MBR scans its embedded list of primary partitions and loads the VBR of the first "active" primary partition. Either way, the system ends up loading and running a VBR.

This VBR loads and runs the Windows NT 6 boot manager, which is required to be stored as a file named `bootmgr` in the root directory of the boot volume.

Unlike on EFI systems, IBM PC compatible firmwares execute boot sectors as real mode programs using 16:16 addressing. It is up to the boot loaders themselves to switch the processor into protected mode if that is required.

The Microsoft boot manager therefore contains a 16-bit stub program, prepended to the boot manager proper (which is a PE-format 32-bit executable that follows the stub program), that switches the processor into 32-bit, flat memory model, protected mode before invoking the boot manager proper. Essentially, the real mode stub sets up the same environment that would occur if EFI firmware had invoked the boot manager from the PE-format executable directly. The stub initializes mode switching function call thunks that map (a subset of) the 32-bit protected mode machine firmware services that are provided on EFI systems to the 16:16 real mode machine firmware services provided by the actual IBM PC compatible firmware.

How Windows NT 6's boot loader is invoked by Windows NT 6's boot manager

Once Microsoft's boot manager is running, the bootstrap process for EFI firmware and IBM PC compatible firmware machines is largely the same.

Microsoft's boot manager reads a Boot Configuration Data file. The file is formatted in the same way as the Windows NT 6 registry hives are. Other BCD files (which Microsoft terms "BCD stores") are allowed, but this one is required and is the one that is read by the Windows NT 6 boot manager. Microsoft terms it the "system store".

The Boot Configuration Data file comprising the "system BCD store" is located in different places according to the type of the machine firmware:

- On IBM PC compatible firmware machines, it is a file named "`\Boot\BCD`" in the boot volume.

- On EFI firmware machines, it is a file located in the "`\EFI\Microsoft\Boot\`" directory on the EFI system partition.

That the location differs according to firmware type is the reason that it is difficult to switch an installed Windows NT 6 system between EFI firmware and IBM PC compatible firmware. (Microsoft explains that it is difficult in its documentation, but fails to explain that this is why it is difficult.) Switching requires that the system BCD store file be copied to the appropriate location.

The system BCD store contains a Windows-centric equivalent of the EFI boot manager configuration data. Everything that Microsoft's boot manager does can be done using the EFI boot manager directly:

- A "Windows Boot Manager" data structure (known by the GUID {9dea862c-5cdd-4e70-acc1-f32b344d4795}, which has a shorthand {bootmgr} when using Microsoft's tools for editing BCD files) comprises configuration data that controls the operation of Microsoft's boot manager as a whole. It comprises references to the data structures for entries on the boot manager menu, and bootmanager-wide configuration settings such as the timeout before the default entry is bootstrapped.

  This is a Windows-centric equivalent of the EFI boot manager's `BootOrder` and `Timeout` variables.

- "Windows Boot Loader" data structures (known by arbitrary GUIDs) comprise control information for bootstrapping Windows NT 6, specifically, in a certain way. Individual parts of each data structure control kernel settings such as the location of the system volume, the location of the `winload.exe` file, the configuration of the kernel debugger, the use of physical address extensions, and the use of no-execute page protection.

  This is a Windows-centric equivalent of what in an EFI boot manager would be a `Boot####` variable pointing to a (hypothetical) `winload.efi` program capable of loading and running a Windows NT 6 kernel, with kernel settings as parameters.

  > This is not that hypothetical, given that Microsoft already has EFI boot loaders for Windows NT. See later.

- "Windows Resume Loader" data structures comprise control information for resuming Windows NT 6 from hibernation.

This is a Windows-centric equivalent of what in an EFI boot manager would be a `Boot####` variable pointing to a (hypothetical) `winresume.efi` program capable of resuming Windows NT 6 from hibernation. Before hibernation, the `BootNext` variable would be set to point to this variable, so that the EFI boot manager would automatically resume Windows NT on the next startup.

- "Windows NTLDR" data structures comprise control information for bootstrapping Windows NT via loading an running an NTLDR program (the mechanism used to boot versions of Windows NT prior to Windows NT 6). Specifically, they comprise the location of the NTLDR program to be loaded and run. There can be many such data structures, albeit that one is known by the distinguished GUID {466f5a88-0af2-4f76-9038-095b170dc21c} (which has a shorthand {ntldr} when using Microsoft's tools for editing BCD files).

  This is a Windows-centric equivalent of what in an EFI boot manager would be a `Boot####` variable pointing to a (hypothetical) `ntldr.efi` program capable of loading and running an NTLDR image, with the pathname of such an image as a parameter.

  > Of course, the NTLDR program proper is an ordinary, 32-bit, flat memory model, PE-format executable, and could be invoked directly by the EFI boot manager. Indeed, on ARC and 64-bit x86 systems, it is. The firmware loads and runs `OSLOADER.EXE` or `IA64LDR.EFI`, which are just NTLDR by another name and without the 16-bit real-mode stub program tacked onto the front.

- "Boot application" data structures comprise control information for running arbitrary Microsoft boot-time diagnosis and maintenance utilities, such as the "Microsoft Memory Tester", `memtest.exe`, or the tools for adjusting the bootstrap code in the VBRs of FAT and NTFS volumes, `fixfat.exe` and `fixntfs.exe`.

  This is a Windows-centric equivalent of what in an EFI boot manager would be a `Boot####` variable pointing to an arbitrary EFI executable for performing boot-time diagnosis and maintenance.

- "Boot sector" data structures comprise control information for bootstrapping the Volume Boot Record of a disc volume. These are used to in order to configure Microsoft's boot manager to load and to run the VBRs for other operating systems.

  This is a Windows-centric equivalent of what in an EFI boot manager would be a `Boot####` variable pointing to a (hypothetical) `vbrldr.efi` program capable of loading and running a VBR, passing the partition (or disc file) containing that VBR as a parameter.

The Windows NT 6 boot manager presents a menu to the user to select what to boot. (So on EFI systems users see two successive boot manager menu screens.) This menu comprises a list of

Windows Boot Loader, Windows Resume Loader, Windows NTLDR, "boot application", and "boot sector" entries, each defined by its own data structure in the BCD file and listed in the Windows Boot Manager data structure.

The two relevant types of entry for bootstrapping Windows NT 6 itself are the Windows Boot Loader and Windows Resume Loader entries.

When a Windows Resume Loader entry is selected by the user, Microsoft's boot manager invokes the program `winresume.exe` to resume Windows NT 6 from hibernation. The system BCD store contains configuration information describing what `winresume.exe` should re-load.

When a Windows Boot Loader entry is selected by the user, Microsoft's boot manager invokes the program `winload.exe` to load the operating system proper ab initio.

How Windows NT 6 is bootstrapped by its boot loader

WINLOAD, the Windows Boot Loader, loads the Windows NT 6 kernel, boot-class device drivers, and system registry hive, just as NTLDR did in earlier versions of Windows NT.

> WINLOAD is in fact capable of loading earlier Windows NT kernels. In early beta releases of Windows NT 6, before the advent of Boot Configuration Data, the `boot.ini` file was split in twain, with one section denoting operating systems that could be loaded via NTLDR and the other section denoting operating systems that could be loaded via WINLOAD. Beta testers discovered that both Windows NT version 5.10.2600 SP2 (i.e. Windows XP), and Windows NT 5.20.3790 (i.e. Windows Server 2003) could be loaded by WINLOAD, as long as `winload.exe` was copied to the `System32` directory on the target system volume.

WINLOAD is simpler than NTLDR, however. NTLDR implements a "dual boot" system, parses `boot.ini`, implements hibernation resume, and presents a boot menu to the user before actually performing the nitty-gritty of loading the operating system. With WINLOAD, all of those tasks either have already been performed by a boot manager or are the purview of other programs such as WINRESUME. WINLOAD therefore only performs those functions of NTLDR that involve actually loading the operating system.

WINLOAD doesn't even have to switch into protected mode. NTLDR is (on 32-bit x86 systems) invoked in real mode by the Volume Boot Record code. It thus comprises a real-mode stub executable, prepended to the loader proper, that switches into 32-bit, flat memory model, protected mode and then invokes the loader proper (stored as PE-format executable in the remainder of the program image file). This is unnecessary with WINLOAD. Either the EFI firmware or the real-mode stub prepended to `\Bootmgr` has already switched the processor into protected mode.

WINLOAD simply loads the operating system kernel, `system32\ntoskrnl.exe`, the hardware abstraction layer, `system32\hal.dll`, the contents of the system registry hive, `system32\config\system`, all of the "boot" class device drivers, and any kernel-mode DLLs imported by any of the aforementioned, into memory.

WINLOAD loads the system registry hive first. Immediately after doing so it verifies its own in-memory executable image against a digital signature held in a digital signature catalogue file, `system32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\nt5.cat`. If this check fails, WINLOAD will halt unless kernel debugging is enabled.

WINLOAD then loads the operating system kernel, the hardware abstraction layer, and all kernel-mode DLLs that they import. If kernel debugging is enabled, in addition WINLOAD loads one of several debugging libraries, which are kernel-mode DLLs that provide the debug kernel with library routines to communicate with the kernel debugger through a specific communications device. These files are `system32\kdcom.dll`, `system32\kd1394.dll`, and `system32\kdusb.dll`, which enable kernel debugging via an RS232 serial port, an IEEE 1394 serial port, or a USB serial port, respectively.

WINLOAD checks the image files for all of these against the digital signatures held in the aforementioned digital signature catalogue. WINLOAD in fact checks all image files – kernel, HAL, DLLs, and device drivers – that it loads, as it loads them. Digital signature checking is done by the code that loads image files into memory. All images must pass the digital signature check, with a signature that is traceable back to by a known Root Certification Authority (exactly 8 of which – 7 Microsoft and 1 Verisign – are hardwired directly into the signature checking code). WINLOAD contains no code to enable the revocation of any certificates.

If the check fails, WINLOAD will halt unless kernel debugging is enabled. Even if kernel debugging is enabled, WINLOAD will halt if one of a small fixed set of image files (`winload.exe`, `ntoskrnl.exe`, `hal.dll`, `bootvid.dll`, `tpm.sys`, `ksecdd.sys`, `clfs.sys`, `ci.dll`, `kdcom.dll`, `kdusb.dll`, `kd1394.dll`, and `spldr.sys`) fails the check.

WINLOAD then scans the registry, in particular the `HKEY_LOCAL_MACHINE\SYSTEM\Services` key, for the configured device drivers. It loads all of the device drivers that are in the "boot" class (`SERVICE_BOOT_START`) into memory, which again involves checking digital signatures.

WINLOAD then enables paging.

Finally, WINLOAD passes control to the operating system kernel.

How Windows NT 6's kernel initializes

The Windows NT 6 kernel performs the usual Windows NT kernel initialization steps, that are largely unchanged from Windows NT version 3.1:

1. Request the HAL to initialize the interrupt controller.
2. Initialize the Memory Manager, the Object Manager, the Security Reference Monitor, and the Process Manager.
3. Request the HAL to enable interrupts.
4. Start all non-boot CPUs.
5. Reinitialize the Object Manager.
6. Initialize the "Executive".
7. Initialize the "Microkernel".
8. Reinitialize the Security Reference Monitor.
9. Reinitialize the Memory Manager
10. Initialize the Cache Manager.
11. Initialize the Local Procedure Call system.
12. Initialize the I/O Manager. Initialization of the I/O manager initializes all of the pre-loaded, "boot" class, device drivers.
13. Initialize the Process Manager.

The operating system kernel then scans the registry, the in-memory copy passed to it by WINLOAD, for the configured device drivers. It loads and all of the device drivers that are in the "system" class.

The kernel checks the digital signatures of all of the image files from which it loads device drivers, using routines exported from `ci.dll`, the kernel-mode DLL that provides a set of "code integrity" library functions. (Much of the content of `ci.dll` is exactly the same cryptographic code that is statically linked into `winload.exe`, including the 8 hardwired Root Certification Authorities.)

The kernel finally invokes the first user process, the so-called Session Manager Subsystem (SMSS).

The user-mode initialization in Windows NT 6

User-mode initialization involves several processes, executing in parallel and acting in concert. The first of these is the SMSS. This spawns other processes, which in their turn spawn yet other processes still. All processes run under the aegis of the "Local System" user account. (If that account is ever denied execute rights to the program image files for these various processes, the system will fail to initialize.)

The Session Manager Subsystem process' rôle in initialization

The SMSS process uses the native kernel API and manages sessions and subsystems (e.g. the Win32 subsystem, the 16-bit OS/2 subsystem, and the POSIX subsystem).

The SMSS first mounts the registry hive files. When SMSS mounts the system hive, the kernel merges into it the in-memory copy of the system registry hive that was loaded by WINLOAD, so that additions and updates to the system portion of the registry (but not deletions) that were made earlier in the boot process before the hive was mounted are preserved.

The SMSS then runs any boot-time programs specified by values beneath the `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\BootExecute` key in the registry. SMSS runs these programs synchronously, waiting for them to complete before proceeding.

The SMSS then issues a request to the kernel to load the device driver that is named by the `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Subsystem\KMode` value in the registry. This is normally `system32\win32k.sys`, the driver that implements the kernel-mode portion of the Win32 API. This driver initializes the Win32 graphics subsystem, switching the display from textual to graphical.

The SMSS then performs system intialization tasks such as

- executing any pending file/directory renaming or deletion operations that have been listed in the Registry (under `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations`) to be executed when the system next initializes,
- pre-loading "known DLLs" (so that they are always open, and thus will be faster to load into processes),

On other operating systems, the pre-loading of DLLs is done by ad-hoc user processes that execute with normal user privileges. (Witness `emxload` on OS/2, for example.) Because the pre-loading of DLLs is done by a process running under the aegis of the local system user account and with Trusted Computer Base privileges, one must be very careful about what is added to the registry's list of "Known DLLs" on Windows NT.

- reading the contents of the initial process environment from the registry and initializing the environment from it,

  and

- initializing additional page files.

Penultimately, the SMSS starts up the subsystem processes for sessions 0 and 1, an init process for session 0, and a logon process for session 1. (The init process is new to Windows NT 6. On prior versions of Windows NT the SMSS would create subsystem and logon processes for session 0, and much of what the init process does on Windows NT 6 would be handled by the first instance of the logon process.) It does this by spawning copies of itself, whose sole duties are to start the processes for a single session and then simply exit. The reason that it does this is that there is insufficient parallelism in its code for starting sessions. Having multiple SMSS process each starting an individual session allows multiple sessions to start in parallel. Using a single SMSS process would result in sessions being started sequentially. (Quite why Microsoft resorted to spawning a wholly new child process rather than simply employing a separate thread within the main process to start each session is unclear.)

To start the subsystems for sessions 0 and 1, the SMSS reads the registry values named by the `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Subsystem\Required` value in the registry. This value points to further values under the `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Subsystem` key. Usually it names the `Debug` and the `Windows` values under that key. These values, in turn, specify the program image files for processes that the SMSS then runs in each session.

Normally, the `Windows` value names `system32\csrss.exe`, the server process ("Client-Server Runtime SubSystem") that implements the user-mode portion of the Win32 API. Once this subsystem process is running in a session, the system is capable of running Win32 programs in that session.

The SMSS spawns the WININIT process, using the `system32\wininit.exe` program image file, in session 0 and a WINLOGON process, using the `system32\winlogon.exe` program, in session 1. Thus only session 1 is a "WINLOGON" session.

The SMSS finally enters a loop waiting for LPC requests or for WININIT, WINLOGON, or CSRSS to terminate. Other processes may communicate with the SMSS using a LPC port (`\SmApiPort`) to invoke additional subsystem processes (such as `system32\psxss.exe`) in a session, to create additional sessions (which would have their own subsystem and logon processes), or to shutdown the system. If WININIT, WINLOGON, or CSRSS ever terminate, SMSS crashes the system. (See the CSRSS Backspace Bug.)

BootExecute processes

BootExecute processes spawned synchronously by SMSS execute under the aegis of the Local System user account, and must use the native kernel API (the Win32 subsystem, both the kernel-mode and the user-mode portions, having not yet been initialized).

One such process executes `system32\autochk.exe`. This is the same program code as `chkdsk.exe`, except that the latter is an ordinary Win32 program, whereas `autochk` has been compiled to use the native kernel API for its user interface.

The Client-Server Runtime Subsystem process' `rôle` in initialization

The CSRSS process uses the native kernel API and implements the user-mode part of the Win32 subsystem. In Windows NT prior to version 4.0, it implemented the whole of the Win32 API, which was implemented wholly in user-mode. In Windows NT 6, only functionality such as console handling remains in CSRSS, most functionality having been moved to `system32\win32k.sys`. Several kernel-mode threads, created by `win32k.sys`, are also created in the CSRSS process.

CSRSS listens on an LPC port for Win32 API calls and handles them. It is the CSRSS process (in particular the `winsrv.dll` dynamic link library that it links to) that creates and processes messages for the GUI windows that represent Win32 "consoles". (Albeit that there is some jiggery-pokery that goes on with injecting `console.dll` and extra threads into other processes, when certain events occur.)

CSRSS never terminates. If it does, both SMSS and the Windows NT kernel notice. (The CSRSS process has the "critical process" flag set in its process object within the kernel.)

The Windows Init process' `rôle` in initialization

WININIT is a Win32 process that does all of the stuff that the first instance of WINLOGON used to do in prior versions of Windows NT, i.e. stuff that was more related to one-time overall system initialization than to per-session initialization and to user logon. The SMSS by default starts a WININIT process in session 0. There is no need for further WININIT processes.

WININIT spawns the Local Security Authority SubSystem process, using the `system32\lsass.exe` program image file, the Service Controller process, using the `system32\services.exe` program image file, and the Local Session Manager process, using the `system32\lsm.exe` program image file. In prior versions of Windows NT, these two would be managed by the first WINLOGON process, and if either process ever terminated, WINLOGON would initiate a system shutdown and restart. WININIT spawns these processes in Windows NT 6, and WININIT is not involved in the user logon and system shutdown mechanisms.

The Windows Logon process' rôle in initialization

WINLOGON is a Win32 process that provides the user interface for logging on to, logging off from, locking, and unlocking a single session in the system, and that handles system shutdown requests. It manages the spawning of user processes (normally `userinit.exe`) when users log in, and the killing of user processes when users log out. The SMSS by default starts a WINLOGON process in session 1. The Terminal Services server requests SMSS to start further WINLOGON process in further sessions.

In prior versions of Windows NT, the first WINLOGON process would perform one-time overall system initialization actions. In Windows NT 6, this functionality is in WININIT. WINLOGON only performs per-session initialization, and the first WINLOGON process is not a special case.

WINLOGON first creates a "window station" to conceptually bind together one or more keyboards, mice, and displays, and various Win32 global properties to form the logical unit of interaction with a single user. (In Unix/Linux parlance this would be a "head".)

In this window station, WINLOGON then creates three desktops: the WINLOGON desktop, the user desktop, and the screen saver desktop. WINLOGON assigns an ACL to the WINLOGON desktop that prevents any process but itself from accessing that desktop. (It grants permissions to a unique security ID that is only included in its own process token and in no other.)

In prior versions of Windows NT, WINLOGON would load a GINA ("Graphical Identification aNd Authentication") dynamic link library. Various functions in the GINA would handle waiting for the Secure Attention Sequence (Control-Alt-Delete), displaying the various login/logout/lock/unlock dialogue boxes on the WINLOGON desktop, and even invoking the user process (`userinit.exe`).

In Windows NT 6, the GINA scheme has been replaced with a system of Credential Providers, which moves some of that functionality (in particular universal parts such as invoking the user process) into WINLOGON itself and simply separates out into DLLs the functionality of obtaining user credentials via some user interface and of performing user authentication with those credentials via the LSASS. WINLOGON even supports simplified credential providers, where the user interface comprises a set of text fields, handling most of the user interface work on behalf of such providers.

Credential Providers are DLLs that export COM interfaces: `ICredentialProvider`, `ICredentialProviderCredential`, `ICredentialProviderCredentialEvents`, `ICredentialProviderEvents`, and `ICredentialProviderFilter`. They are hosted within a COM server process, LogonIU, that is spawned by WINLOGON.

The Local Security Authority Subsystem process' rôle in initialization

The LSASS process creates an LPC port, and then enters a loop handling security requests, such as requests to verify a set of user credentials against a user account database, that come down that port. Requests arrive from WINLOGON processes, from the network logon service process, and from user processes that wish to perform user authentication.

The Service Controller process' rôle in initialization

The Service Controller process scans the registry for the configured device drivers and services. It loads all of the device drivers and services that are in the "auto" class. It then enters a loop listening on an LPC port waiting for requests to start and to stop services.
