**WINDOWS 7 OPERATING SYSTEM**

by

D. Shashank - 13BCE1036

Harshwardhan Agarwaal - 13BCE1052

S. Nikesh Kumar Reddy - 13BCE1136

*A project report submitted to the*

**SCHOOL OF COMPUTING SCIENCES AND ENGINEERING**

*In partial fulfillment of the award for degree*

*of*

**BACHELOR OF TECHNOLOGY**

*in*

**COMPUTER SCIENCE ENGINEERING**



**VIT UNIVERSITY, CHENNAI**

**Vandalur – Kelambakkam Road**

**Chennai – 600 127.**

**MAY 2015**

**ACKNOWLEDGEMENT**

**Course Code: CSE222 Batch no: 1**

This project report entitled “**WINDOWS 7 OPERATING SYSTEM”** is a bonafide work of **SHASHANK. D (13BCE1036), HARSHWARDHAN AGARWAL (13BCE1052) and NIKESH. S (13BCE1136)** carried out the Project work under the supervision and guidance of **Prof. JAGADEESH KANNAN.** We wish to express our sincere thanks and deep sense of gratitude to our project guide, **Prof.** **JAGADEESH KANNAN,** Associate Professor, School of Computing Sciences and Engineering, for his consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work.

**TABLE OF CONTENTS**

1. Lit. Survey for the Project
2. Abstract
3. Introduction
4. Module and Explanation
5. References

**ABSTRACT**

The operating system is just known as an interface between the software and hardware but the world is unaware of the programs and the number of processes that get executed by the operating system in order to meet the requirements of the user. This project deals about the study of the programs and processes executed by the operating system from the boot process to the shutdown of the system.

**INTRODUCTION**

The operating system executes each and every step or process required by the user using various types and various system calls which are not of this paper’s concern. The project is all about the study of the processes which are being executed without the intervention of the user. Such processes get executed using the startup i.e. during boot time and at shutdown.

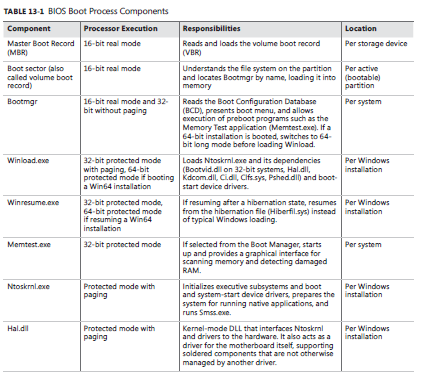
**STARTUP AND SHUTDOWN MODULE**

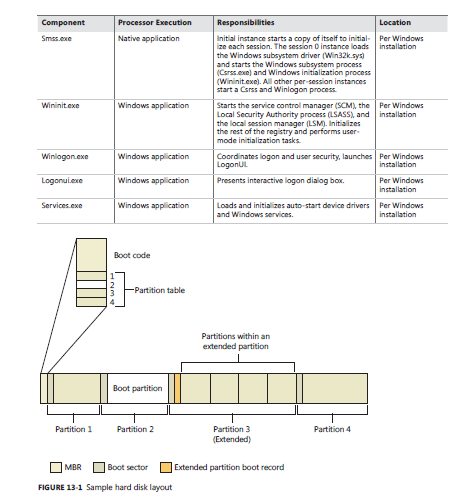
**Boot Process:**

* 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)
* 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 pre-boot 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 Pre – Boot:**

* 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.



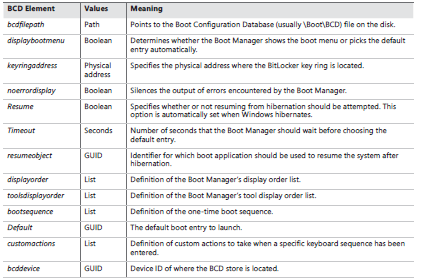
****

* 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.
* 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.
* 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.
* 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 pre-existing Windows installations.
* If the BCD already exists, the Setup program simply adds new entries relevant to the new installation.

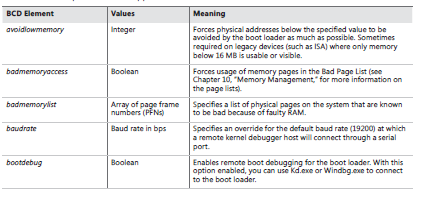
**BIOS BOOT SECTOR AND BOOTMGR:**

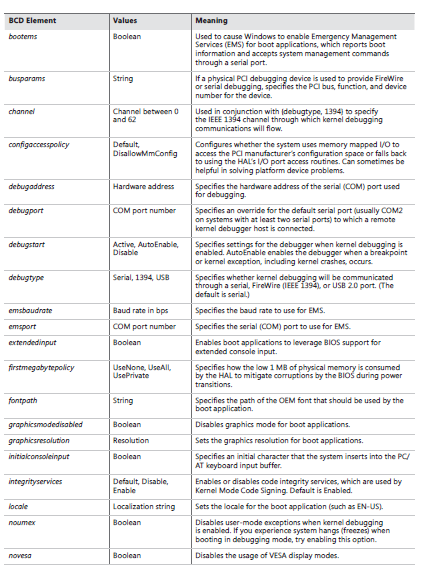
* Setup must know the partition format before it writes a boot sector because the contents of the boot sector vary depending on the format.
* For a partition that is in ntfs format, windows writes ntfs capable code. The role of the boot-sector code is to give windows information about the structure and 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.
* 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.
* 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.

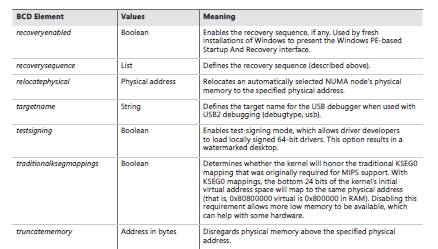
**BCD Options for the Windows Boot Manager (Bootmgr):**



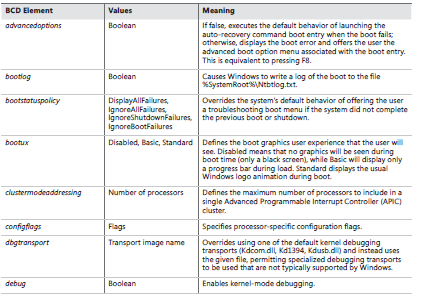
**BCD Options for Boot Applications:**

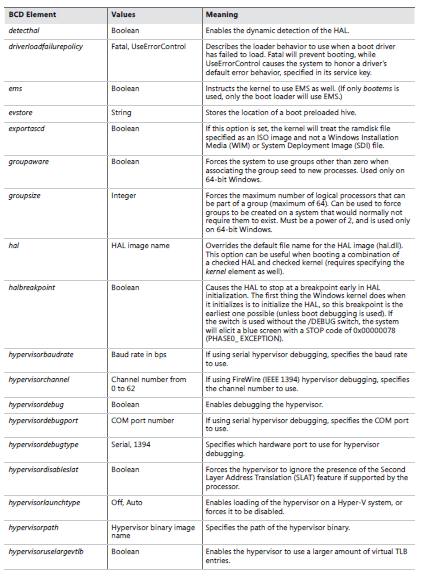


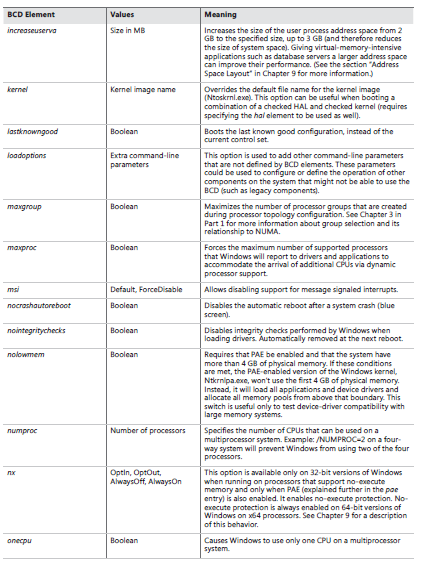


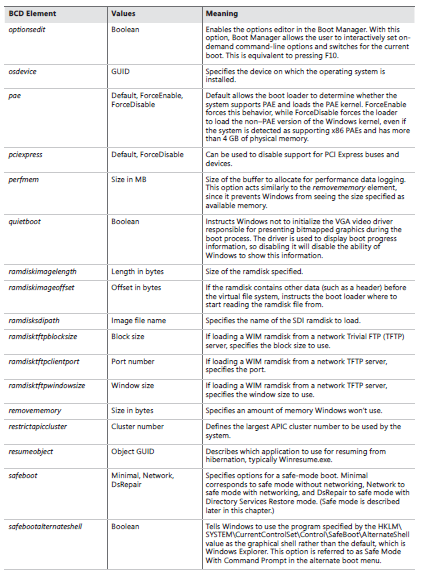


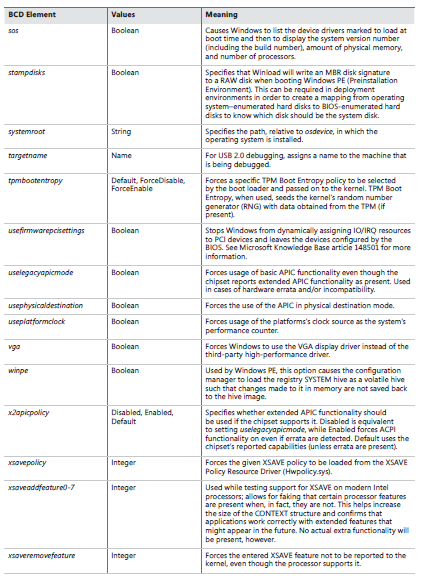
**BCD Options for the Windows Boot Loader WINLOAD:**

****

****

****

****

****

* 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
* 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:

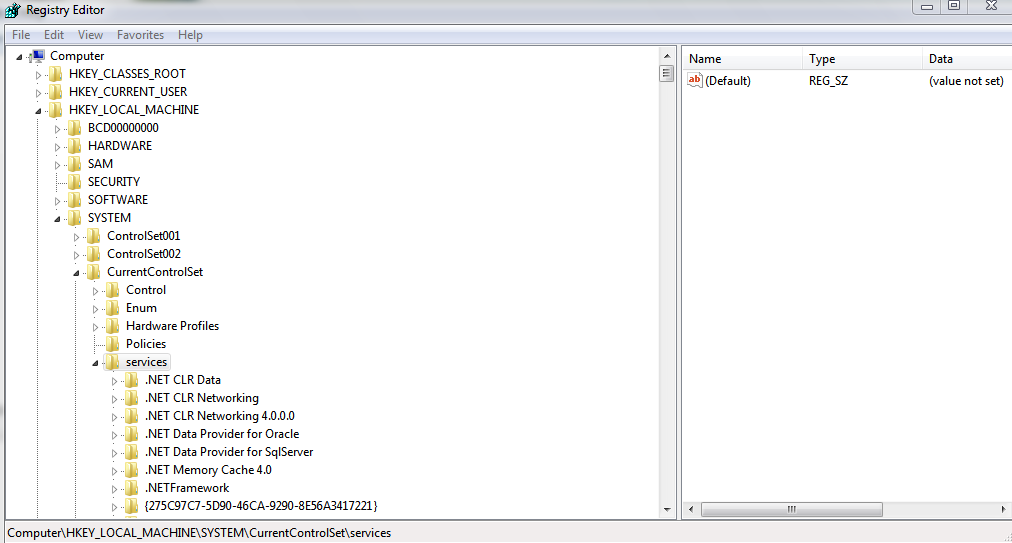
1. Loads the appropriate kernel and HAL images (Ntoskrnl.exe and Hal.dll by default) as well as any of their dependencies. If Winload fails to load either of these files, it prints the message “Windows could not start because the following file was missing or corrupt”, followed by the name of the file.

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 l\_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.

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 underHKLM\SYSTEM\CurrentControlSet\Services. For example, Services has a subkey named fvevol for the BitLocker driver.



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.

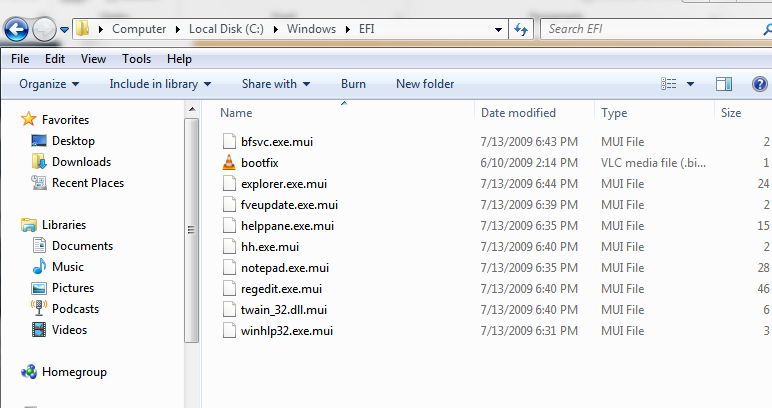
* For steps 1 and 8, Winload also implements part of the Kernel Mode Code Signing (KMCS) infrastructure.
* Additionally, the system will crash if the signature of the early boot files is incorrect.
* This action is the end of Winload’s role in the boot process.
* At this point, Winload calls the main function in Ntoskrnl.exe (*KiSystemStartup*) to perform the rest of the system initialization.

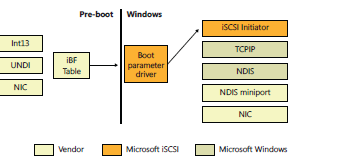
**UEFI Boot Process:**

* 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:

1. Network adapters
2. Video adapters
3. Keyboards
4. Disk controllers
5. Storage devices

* 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

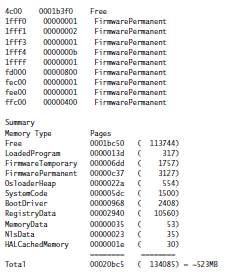




**Initializing Kernel and Executive Sub - systems:**

* 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.

**Loader Parameter Block:**



****

* 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.

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.

**Phase1:**

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 pre-emption.

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 start-up 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

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.

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. he older LPC objects are set as aliases.

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.

39. The transaction manager sets up the Windows software trace preprocessor (WPP) and ETW and initializes with WMI.

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 usermode 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.

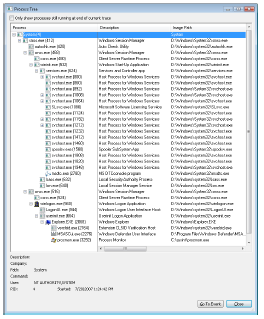
48. The security reference monitor is called to create the Command Server Thread that communicates with LSASS.

49. The Session Manager (Smss) process 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.

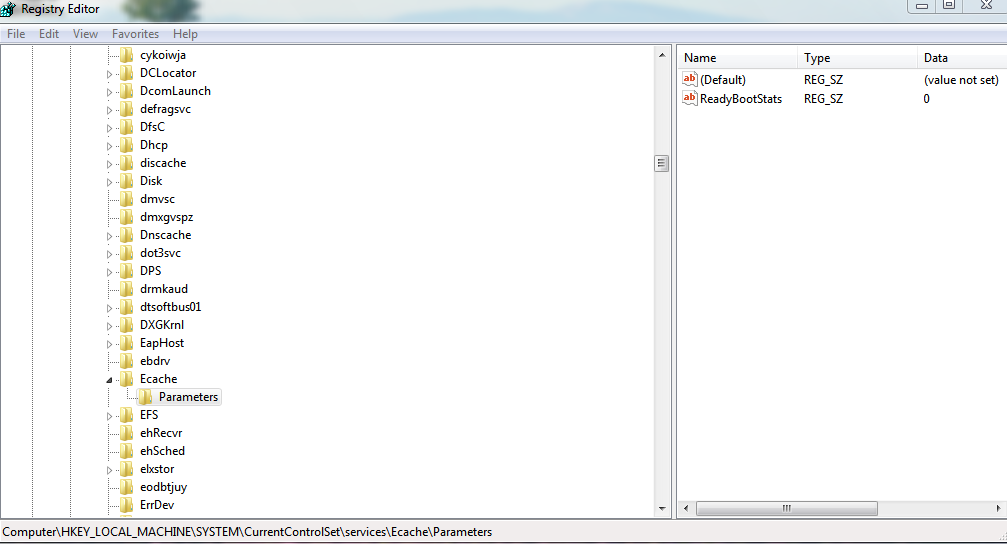
51. All the memory used up by the loader parameter block and all its references is now freed.

**Process Tree during Logon:**

****

**Ready Boot:**

* 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.
* 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, (MY COMPUTER)

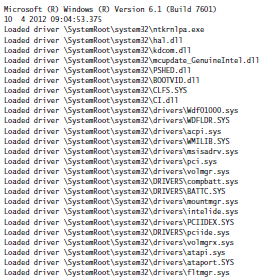


**Safe Mode:**

* When Windows boots, you press the F8 key to enter a special boot menu that contains the safemode 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.

**Partial Contents Of a Sample Boot Log:**

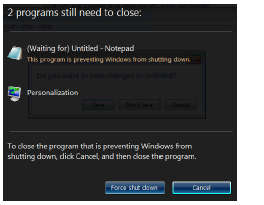




**Shutdown:**

* 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.
* Remote shutdown is also done but is excluded in windows 7 with networking bases. Internally, the execution can be stopped.
* 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.
* 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 hungprogram 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.

**Hung Program Screen:**



**References:**

* <http://blogs.windows.com/windows/archive/b/windows7/archive/2009/10/22/windows-7-arrives-today-with-new-offers-new-pcs-and-more.aspx>
* Mike Nash (October 14, 2008). ["Why 7?"](http://windowsteamblog.com/windows/archive/b/windowsvista/archive/2008/10/14/why-7.aspx). *The Windows Blog*. Microsoft. RetrievedFebruary 1, 2011.
* ["NtShutdownSystem"](http://undocumented.ntinternals.net/source/usermode/undocumented%20functions/hardware/ntshutdownsystem.html). 2001-02-25. Retrieved 2015-04-11
* Personal Research