Chapter-10

## Boot process

1.0 How Does an Intel Processor Boot

When we switch on a computer, it goes through a series of steps before it is able to load the operating system. In this post we will see how a typical x86 processor boots. This is a very complex and involved process. We will only present a basic overall structure. Also what path is actually taken by the processor to reach a state where it can load an OS, is dependent on boot firmware. We will follow example of coreboot, an open source boot firmware.

1.1 Before Power is Applied

Let us start with BIOS chip, also known as boot ROM. BIOS chip is a piece of silicon on the motherboard of a computer and it can store bytes. It has two characteristics which are of interest to us. First, it (or a part of it) is memory mapped into the CPU’s address space, which means that the CPU can access it in the same way it would access RAM. In particular, the CPU can point its instruction pointer to executed code inside BIOS chip. Second, the bytes that BIOS chip stores, represent the very first instructions that are executed by the CPU. BIOS chop also contains other pieces of code and data. A typical BIOS contains flash descriptor (a contents table for BIOS chip), BIOS region (the first instructions to be executed), Intel ME (Intel Management Engine) and GbE (gigabit ethernet). As you can see, BIOS chip is shared between serveral components of the system and not exclusive to CPU.

1.2 When power is applied

Modern Intel chips come with what is called Intel Management Engine. As soon as power is available – through battery or from mains – Intel ME comes on. It does its own initialisations which requires it to read BIOS’s flash descriptor to find where Intel ME region is and then from Intel ME region of BIOS, read in code and config data. Next when we press power button on the computer, the CPU comes on. On a multiprocessor system, there is always a designated processor, called Bootstrap Processor (BSP), which comes on. In either case, the processor always comes on in what is called 16-bit Real Mode with insruction pointer pointing to address 0xffff.fff0, the reset vector.

EDIT: (thanks to [burfog](https://news.ycombinator.com/user?id=burfog) for indicating that this needs explaination)

You might be wondering how could a 16-bit system address 0xffff.fff0 which is clearly beyond 0xffff, the max 16-bit value? In 16-bit mode,  physical address is calculated by left shifting code segment (CS) selector register by 4 bits and then adding instruction pointer (IP) address. On reset, IP cotains value 0xfff0 and CS has value 0xf000 [1]. By the above formula the physical address should be:

CS << 4 + IP = 0x000f.0000 + 0xfff0 = 0x000f.fff0

which is still not what we expected. This is because on reset, the system is in a “special” Real Mode, where the first 12 address lines are asserted. So all addresses look like 0xfffx.xxxx. This means in our case, we need to set the most significant 12 bits in the address we derived, which results in our expected address 0xffff.fff0. These 12 address lines remain asserted until a long JMP is executed, after which they are de-asserted and normal Real Mode addressing calculations resume.

The BIOS chip is also set up in such a way that first instruction to be executed from the BIOS is at physical address 0xffff.fff0 of the processor. Hence processor is able to execute the first instruction from BIOS region of the BIOS chip. This region contains what is called boot firmware. Examples of boot firmware are UEFI implementations, coreboot and the classic BIOS.

One of the first things that the boot firmware does is switch to 32-bit mode. It is also “protected mode”, i.e. segmentation is turned on and various segments of processor’s address space can be managed with different access permissions. Boot firmware however would have just one segment, effectively turning off segmentation. This is called flat mode.

2.0 Early Initialization

It is worth noting that at this point in boot process, DRAM is not available. DRAM Initialization is one of the main objectives of boot firmware. But before it can initialize DRAM, it needs to do some preparation.

Microcode patches are like patches for CPU to function correctly. Intel keeps publishing microcode patches for different CPUs. The boot firmware applies those patches very early on in boot process. Part of the processor is what is called south bridge or I/O controller hub (ICH) or peripheral controller hub (PCH). There are some initializations to be performed for ICH also. For example, ICH may contain a watchdog timer which can go off which DRAM is being initialized. That watchdog timer must be turned off first.

Of-course all of this is being done by firmware which is code written by someone. Now most of the code we know utilizes stack. But we have mentioned that DRAM hasn’t been initialized yet so there is no memory. So how is this code written and run? Answer is that this is stack-less code. Either it is hand written x86 assembly or, as in case of coreboot, it is written in C and compiled using special compiler called ROMCC which translates C to stackless assembly instructions. This of course comes with some restrictions so ROMCC compiled code is not how we want to execute everything. We need stack as soon as possible.

So, the next step is setting up what is called cache-as-RAM (CAR). Boot firmware basically sets up CPU caches so that they can be temporarily used as RAM. This way the firmware can run code which is not stackless, but still restricted in terms of stack size and general amount of memory available.

3.0 Memory Initialization and Intel FSP

On Intel systems, memory initialization is performed using a blob called Intel Firmware Support Package (FSP). This is supplied by Intel in binary form. Intel FSP does a lot of heavy lifting when it comes to bootstrapping Intel processors and is not just limited to memory init. It is basically a three stage API. The way boot firmware interacts with FSP is set up some parameters and a return address, and jump into an FSP stage. The FSP stage would execute taking into account the parameters and then use the return address to jump back into boot firmware. This continues across these three FSP stages and in that order:

* **TempRamInit():** This performs some init for RAM and hand control back to boot firmware. Boot firmware can kick off some actions and then go on to next stage. This is because the next step performs chipset and memory initialization which may take some time. For example memory training is a time consuming operation. So this is an opportunity for boot firmware to kick off other initializations, like spinning up hard drive, which can take time to stabilize.
* **FspInitEntry():** This is where actual DRAM is achieved. This also performs other silicon init, like PCH and CPU itself. After this finishes, it passes control back to boot firmware. However, since this time, the memory has been initialized, the passing back of control and data is different from TempRamInit stage. After this stage, firmware does most of the rest of initializations – described in the next section ‘After Memory Init’ – before passing control to the next stage of FSP.
* **NotifyPhase():** This is where boot firmware would pass control back to FSP and set params which would tell FSP what sort of actions it needs to take before winding down. The types of things that FSP can do here are platform dependent but they include things like post PCI enumeration.

3.1 After Memory Init

Once DRAM is ready, it breathes a new life into boot process. First that the firmware does is copy itself into DRAM. This is done with help of “memory aliasing”, which means that reads and writes to addresses below 1MB are routed to and from DRAM. Then, firmware sets up the stack and transfer control to DRAM.

Next, some platform specific inits are done, such as GPIO configuration and re-enabling the watchdog timer in ICH which was disabled before memory init, paving the way for interrupts enabling. Local Advanced Programmable Interrupt Controller (LAPIC) sites inside each processor, i.e. it is local to each CPU in a multiprocessor system. LAPIC determines how each interrupt is delivered to that particular CPU. I/O APIC (IOxAPIC) lives inside ICH and there is one IOxAPIC for all processors. There can also be a Programmable Interrupt Controller (PIC) which is for use in Real Mode as is Interrupt Vector Table which contains 256 interrupt vectors – pointers to handlers for corresponding interrupts. Interrupt Descriptor Table on the other hand, is used to hold interrupt vectors when in Protected Mode.

Firmware then sets up various timers depending upon platform and the firmware. Programmable Interrupt Timer (PIT) is the system timer and sits on IRQ0. It lives inside ICH. High Precision Event Time (HPET) also sits inside ICH but boot firmware may not initialise it, letting the OS to set it up if needed. There is also a clock, the Real Time Clock (RTC) which too resides in ICH. There are other timers too, particularly LAPIC timer which is inside each CPU. Next, the firmware sets up memory caching. This basically means setting up different cache characteristics – write-back, uncached etc – for different ranges of memory.

3.2 Other Processors, I/O Devices and PCI

Finally, it is time to bring up other processors as all the work so far was being handled by the bootstrap processor. To find out about other application processors (AP) on the same package, BSP runs CPUID instruction. Then using its LAPIC, BSP sends an interrupt called SIPI, to each AP. Each SIPI points to the physical address at which the receiving AP should start executing. It is worth noting that each AP comes up in Real Mode, therefore the SIPI address must be less than 1MB, the maximum addressable in Real Mode. Usually soon after initialisation, each AP executes HLT instruction and gets into halt state, waiting for further instructions from BSP. However, just before OS gains control, APs are supposed to be in “waiting-for-SIPI” state. BSP achieves this by sending a couple of inter-processor interrupts to each AP.

Next come I/O devices like Embedded Controller (EC) and Super I/O, and after that PCI init. PCI init basically boils down to:

1. enumerating all PCI devices
2. allocating resources to each PCI device

This discussion here applies to PCIe also. PCI is a hierarchical bus system where for each bus, leaf is either a PCI device or a PCI bridge leading to another PCI bus. CPU communicates with PCI by reading and writing PCI registers. The resources needed by PCI devices are range inside memory address space, range inside I/O address space and IRQ assignment. CPU finds out about address ranges and their types (memory-mapped or I/O) by writing to and reading from Base Address Registers (BARs) of PCI devices. IRQs are usually set up based how the board is designed.

During PCI enumeration, firmware also reads Option ROM register. If that register is not empty then it contains address of Option ROM. This is ROM chip that is physically situated on the PCI device. For example the network card may contain Option ROM which holds iPXE firmware. When an Option ROM is encountered then it is read into DRAM and executed.

4.0 Handing Control to OS loader

Before handing over control to next stage loader which is usually an OS loader like GRUB2 or LILO, the firmware sets up some information inside memory which is later to be consumed by the OS. This information is things like Advanced Configuration and Power Interface (ACPI) tables and memory map itself. Memory map tells the OS what address ranges have been set up for what purposes. The regions can be gerenal memory for OS use, ACPI related address ranges, reserved (i.e. not to be used by OS), IOAPIC (to be used by IOAPIC), LAPIC (to be used by LAPICs). Boot firmware also sets up handlers for System Management Mode (SMM) interrupts. SMM is an operating mode of Intel CPUs, just like Real, Protected and Long (64-bit) modes. A CPU enters SMM mode upon receipt of an SMM interrupt which can be triggered by a number of things like chip’s temperature reaching a certain level. Before handing control to OS loader, the firmware also locks down some registers and CPU capability, so that it can’t be changed afterwards by the OS.

Actual transfer of control to the OS loader usually takes form of a JMP to that part of memory. An OS loader like GRUB2 will perform actions based on its config and ultimately pass controle to an operating system like Linux. For Linux, this will usually be a bzImage (big zImage, not bz compression). It is worth noting that the OS, like Linux would enumerate PCI devices again and may have other overlap with some of the final initialisations done by boot firmware. Linux usually picks up the system in 32-bit mode with paging turned off and performs its own initialisations which include setting up page tables, enabling paging and switching to long mode, i.e. 64-bit.

\* How Does an Intel Processor Boot?  
When we switch on a computer, it goes through a series of steps before it is able to load the operating system; in this post, we will see how a typical x86 processor boots. This is a very complex and involved process, but we will only present a basic overall structure. Also, what path is actually taken by the processor to reach a state where it can load an OS, is dependent on boot firmware. We will follow example of coreboot, an open source boot firmware.

\* Before Power is Applied  
Let us start with the BIOS chip, also known as boot ROM. The BIOS chip is a piece of silicon on the motherboard and it can store bytes. It has two characteristics which are of interest to us. First, it (or a part of it) is memory mapped into the CPU’s address space, which means that the CPU can access it in the same way it would access RAM. In particular, the CPU can point its instruction pointer to executed code inside BIOS chip. Secondly, the bytes that the BIOS chip stores, represents the very first instructions that are executed by the CPU. The BIOS chip also contains other pieces of code and data. A typical BIOS contains a flash descriptor (a contents table for BIOS chip), BIOS region (the first instructions to be executed), Intel ME (Intel Management Engine) and GbE (gigabit ethernet). As you can see, the BIOS chip is shared between serveral components of the system and is not exclusive to CPU.

\* When power is applied  
Modern Intel chips come with what is called Intel Management Engine. As soon as power is available, Intel ME comes on. It does its own initialisations which requires it to read the BIOS’s flash descriptor to find where Intel ME region is and then from Intel ME region of BIOS, read the code and config data. Next, when we press power button on the computer, the CPU comes on. On a multiprocessor system, there is always a designated processor, called Bootstrap Processor (BSP), which comes on. In either case, the processor always comes on in what is called 16-bit Real Mode with the insruction pointer pointing to address 0xffff.fff0, the reset vector.

You might be wondering how could a 16-bit system address 0xffff.fff0 which is clearly beyond 0xffff, the max 16-bit value? In 16-bit mode, physical addresses are calculated by left shifting the code segment (CS) selector register by 4 bits and then adding instruction pointer (IP) address. On reset, IP contains value 0xfff0 and CS has value 0xf000. By the above formula the physical address should be:

CS << 4 + IP = 0x000f.0000 + 0xfff0 = 0x000f.fff0

which is still not what we expected. This is because on reset, the system is in a “special” Real Mode, where the first 12 address lines are asserted. So all addresses look like 0xfffx.xxxx. This means in our case, we need to set the most significant 12 bits in the address we derived, which results in our expected address 0xffff.fff0. These 12 address lines remain asserted until a long JMP is executed, after which they are de-asserted and normal Real Mode addressing calculations resume.

The BIOS chip is also set up in such a way that the first instruction to be executed from the BIOS is at physical address 0xffff.fff0 of the processor. Hence the processor is able to execute the first instruction from the BIOS region of the BIOS chip; this region contains what is called boot firmware. Examples of boot firmware are UEFI implementations, coreboot and the classic BIOS.

One of the first things that the boot firmware does is switch to 32-bit mode. It is also “protected mode”, i.e. segmentation is turned on and various segments of processor’s address space can be managed with different access permissions. Boot firmware however would have just one segment, effectively turning off segmentation. This is called flat mode.

\* Early Initialisations  
It is worth noting that at this point in the boot process, DRAM is not available. DRAM initialisation is one of the main objectives of boot firmware; but before it can initialise DRAM, it needs to do some preparation.

Microcode patches are like patches for the CPU to function correctly; Intel keeps publishing microcode patches for different CPUs. The boot firmware applies those patches very early on in boot process. Part of the processor is what is called south bridge or I/O controller hub (ICH) or peripheral controller hub (PCH). There are some initialisations to be performed for ICH also. For example, ICH may contain a watchdog timer which can go off which DRAM is being initialised. That watchdog timer must be turned off first.

Of course all of this is being done by firmware which is code written by someone, but most of the code we know utilizes the stack. However, we have mentioned that DRAM hasn’t been initialised yet so there is no memory. So how is this code written and run? Answer is that this is stackless code. Either it is hand written x86 assembly or, as in case of coreboot, it is written in C and compiled using special compiler called ROMCC which translates C to stackless assembly instructions. This of course comes with some restrictions so ROMCC compiled code is not how we want to execute everything. We need the stack as soon as possible.

So, the next step is setting up what is called cache-as-RAM (CAR). Boot firmware basically sets up CPU caches so that they can be temporarily used as RAM. This way the firmware can run code which is not stackless, but still restricted in terms of stack size and general amount of memory available.

\* Memory Initialisation and Intel FSP  
On Intel systems, memory initialisation is performed using a blob called Intel Firmware Support Package (FSP). This is supplied by Intel in binary form. Intel FSP does a lot of heavy lifting when it comes to bootstrapping Intel processors and is not just limited to memory init. It is basically a three stage API. The way boot firmware interacts with FSP is to set up some parameters and a return address, and jump into an FSP stage. The FSP stage would execute, taking into account the parameters and then use the return address to jump back into boot firmware. This continues across these three FSP stages and in that order:

TempRamInit(): This performs some init for RAM and hands control back to boot firmware. Boot firmware can kick off some actions and then go on to next stage. This is because the next step performs chipset and memory initialisation which may take some time. For example memory training is a time consuming operation. So this is an opportunity for boot firmware to kick off other initialisations, like spinning up hard drive, which can take time to stabilise.  
FspInitEntry(): This is where actual DRAM is achieved. This also performs other silicon init, like PCH and CPU itself. After this finishes, it passes control back to boot firmware. However, since this time, the memory has been initialised, the passing back of control and data is different from TempRamInit stage. After this stage, firmware does most of the rest of initialisations – described in the next section ‘After Memory Init’ – before passing control to the next stage of FSP.  
NotifyPhase(): This is where boot firmware would pass control back to FSP and set params which would tell FSP what sort of actions it needs to take before winding down. The types of things that FSP can do here are platform dependent but they include things like post PCI enumeration.  
After Memory Init  
Once DRAM is ready, it breathes a new life into boot process. First, the firmware copies itself into DRAM. This is done with help of “memory aliasing”, which means that reads and writes to addresses below 1MB are routed to and from DRAM. Then, firmware sets up the stack and transfer control to DRAM.

Next, some platform specific inits are done, such as GPIO configuration and re-enabling the watchdog timer in ICH which was disabled before memory init, paving the way for interrupts enabling. Local Advanced Programmable Interrupt Controller (LAPIC) sits inside each processor, i.e. it is local to each CPU in a multiprocessor system. LAPIC determines how each interrupt is delivered to that particular CPU core. I/O APIC (IOxAPIC) lives inside ICH and there is one IOxAPIC for all processors. There can also be a Programmable Interrupt Controller (PIC) which is for use in Real Mode as is Interrupt Vector Table which contains 256 interrupt vectors – pointers to handlers for corresponding interrupts. Interrupt Descriptor Tables, on the other hand, are used to hold interrupt vectors when in Protected Mode.

Firmware then sets up various timers depending upon platform and the firmware. Programmable Interrupt Timer (PIT) is the system timer and sits on IRQ0. It lives inside ICH. High Precision Event Time (HPET) also sits inside ICH but boot firmware may not initialise it, leaving the OS to set it up if needed. There is also a clock, the Real Time Clock (RTC) which too resides in ICH. There are other timers as well, particularly LAPIC timer which is inside each CPU. Next, the firmware sets up memory caching. This basically means setting up different cache characteristics – write-back, uncached etc – for different ranges of memory.

\* Other Processors, I/O Devices and PCI  
Finally, it is time to bring up other processors as all the work so far was being handled by the bootstrap processor. To find out about other application processors (AP) on the same package, BSP runs CPUID instruction. Then using its LAPIC, BSP sends an interrupt called SIPI, to each AP. Each SIPI points to the physical address at which the receiving AP should start executing. It is worth noting that each AP comes up in Real Mode, therefore the SIPI address must be less than 1MB, the maximum addressable in Real Mode. Soon after initialisation, each AP executes HLT instruction and gets into halt state, waiting for further instructions from BSP. However, just before OS gains control, APs are supposed to be in a “waiting-for-SIPI” state. BSP achieves this by sending a couple of inter-processor interrupts to each AP.

Next comes I/O devices like Embedded Controller (EC) and Super I/O, and after that PCI init. PCI init basically boils down to:

1. enumerating all PCI devices  
2. allocating resources to each PCI device

This discussion here applies to PCIe also. PCI is a hierarchical bus system where for each bus, leaf is either a PCI device or a PCI bridge leading to another PCI bus. CPU communicates with PCI by reading and writing PCI registers. The resources needed by PCI devices range inside memory address space, range inside I/O address space and IRQ assignment. The CPU finds out address ranges and their types (memory-mapped or I/O) by writing to and reading from Base Address Registers (BARs) of PCI devices. IRQs are usually set up based how the board is designed.

During PCI enumeration, firmware also reads Option ROM register. If that register is not empty then it contains address of Option ROM; this is a ROM chip that is physically situated on the PCI device. For example, the network card may contain Option ROM which holds iPXE firmware. When an Option ROM is encountered then it is read into DRAM and executed.

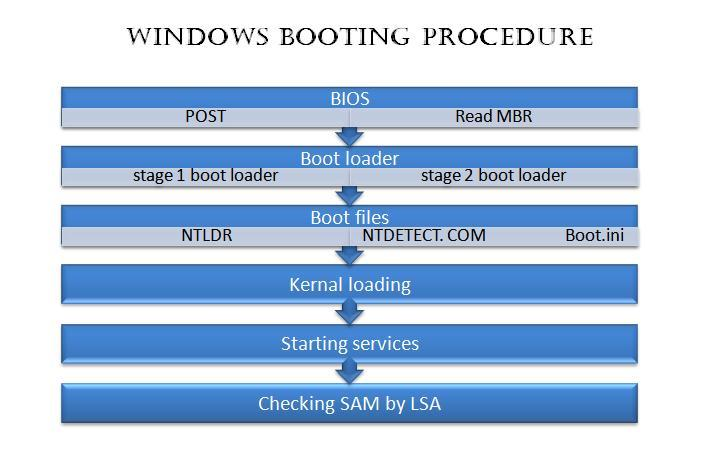
\* Handing Control to OS loader  
Before handing over control to the next stage loader, which is usually an OS loader like GRUB2 or LILO, the firmware sets up some information inside memory which is later to be consumed by the OS. This information is things like Advanced Configuration and Power Interface (ACPI) tables and memory map itself. Memory map tells the OS what address ranges have been set up for what purposes. The regions can be gerenal memory for OS use, ACPI related address ranges, reserved (i.e. not to be used by OS), IOAPIC (to be used by IOAPIC), or LAPIC (to be used by LAPICs). Boot firmware also sets up handlers for System Management Mode (SMM) interrupts. SMM is an operating mode of Intel CPUs, just like Real, Protected and Long (64-bit) modes. A CPU enters SMM mode upon receipt of an SMM interrupt which can be triggered by a number of things like the chip’s temperature reaching a certain level. Before handing control to the OS loader, the firmware also locks down some registers and CPU capability, so that it can’t be changed afterwards by the OS.

Actual transfer of control to the OS loader usually takes form of a JMP to that part of memory. An OS loader like GRUB2 will perform actions based on its config and ultimately pass control to an operating system like Linux. For Linux, this will usually be a bzImage (big zImage, not bz compression). It is worth noting that an OS, like Linux, would enumerate PCI devices again and may have other overlap with some of the final initialisations done by boot firmware. Linux usually picks up the system in 32-bit mode with paging turned off and performs its own initialisations which include setting up page tables, enabling paging and switching to long mode, i.e. 64-bit.

5.0 Why Booting is required?

* Hardware doesn’t know where the operating system resides and how to load it.
* Need a special program to do this job – **Bootstrap** loader.
  + E.g. BIOS – Boot Input Output System.
* Bootstrap loader locates the kernel, loads it into main memory and starts its execution.
* In some systems, a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

5.1 Boot Process Steps



**Fig-1: boot flow**

* Execute code from a well-known location.
* i.e. ROM-BIOS Chip.
* Execute first-stage boot loader from MBR.
* Execute second-stage boot loader.
* Load the kernel.
* Load the first user space program.

6.0 POST (Power-On Self-Test)

* One of the first processes that a computer undergoes when booting.
* POST tests the computer to ensure that it is working as it is supposed to.
* POST can detect some errors with the processor, motherboard, RAM and other memory, as well as the video card.
* Most BIOS chips use a system of beep codes to indicate the POST status to the user and each BIOS chipset uses a different code.
* The IBM PC BIOS code standard, for example, uses one short beep to indicate a successful POST and two short beeps to indicate a POST error.

## Stage 1: BIOS

* BIOS - Basic Input/output System.
* Boot firmware designed to run at start up.
* POST (Power-On Self-Test)
  + Identifies, tests, and initializes system devices
* Run-time services
  + Initial configuration
  + Selects which device to boot from
* Loads the MBR (Master Boot Record) to RAM.

## Stage 2: MBR (Master Boot Record)

* After the POST the BIOS wants to boot up an operating system, which
* must be found somewhere: hard drives, CD-ROM drives, floppy disks, etc.
* The actual order in which the BIOS seeks a boot device is called Boot sequence and is user configurable.
* If there is no suitable boot device the BIOS halts with a complaint like
* “Non-System Disk or Disk Error.”
* The master boot record is always located at cylinder 0, head 0, and
* sector 0, the first sector on the disk
* The BIOS now reads the first 512-byte sector (sector zero) of the hard
* disk. This is called the Master Boot Record.

## Stage 3: VBR (Volume Boot Record)

* Once the BIOS transfers control to the start of the MBR that was loaded
* into memory, the MBR code scans through its partition table and loads
* the Volume Boot Record for that partition.
* The Volume Boot Record is the first sector of a partition, as opposed to
* the first sector for the whole disk.
* The first block of the VBR identifies the partition type and size and
* contains an Instruction Program Loader that contains code to load additional blocks that comprise the second stage boot loader.
* On Windows NT-derived systems (e.g., Windows 2008, Windows 2012,
* Windows 7, Windows 8), the IPL loads a program called NTLDR, which then loads the operating system.

### Bootloaders

* Program that allows the user to select
* which OS to boot, called from MBC
  + NTLDR (NT-Loader) comes with Windows
  + LILO, GRUB, LOADLIN from the Linux world
  + BootX from Apple

6.1 Windows Boot Sequence

* BIOS Power On Self Test
* Firmware checks hardware configuration
* Check media in boot sequence.
* For floppy drives, load the boot sector.
* For hard drives
  + Load the Master Boot Record.
  + Find the active partition.
  + Load boot sector.
* Load system hive(a place in which busily occupied)
* Load some drivers and services
* Initialize kernel and drivers that were loaded
* Load and initialize the rest of the drivers
* Create hardware list in registry using NTDETECT information
* Load and run Auto-check to check file systems
* Run the loaded boot code.
* Load and run NTLDR
* Load and run NTDETECT.COM
* Read BOOT.INI
* Ask which OS to start (if there is an option)
* Load and run Windows Kernel
* Load Hardware Adaptation Layer
* Set up paging
* Load services defined in the registry
* Win32 starts the logon process
* Run applications specified in the registry for this user.
* Run applications in startup directory.

6.2 Boot Errors

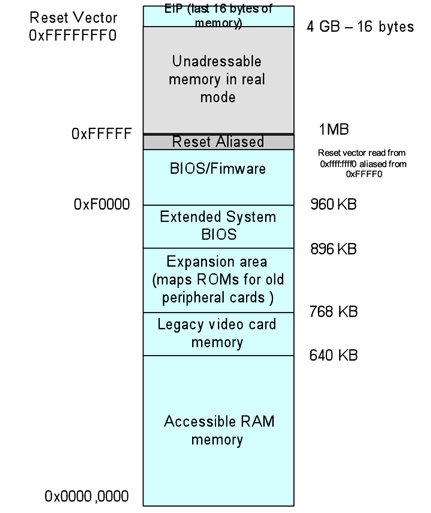
* If the active partition does not contain the boot components (or if you left a data diskette in the floppy drive), you will see
  + **BOOT: Couldn’t find NTLDR**
  + **Please insert another disk**
* If the boot.ini file points to the partition and directory that does not contain a copy of NT, you may
* see the messages:
  + **Windows could not start because the following file is missing or corrupt:**
* **\<winnt root>\system32\ntoskrnl.exe Please re-install a copy of the above file.**

7.0 Processor Initialization

* Two pins, RESET and INIT, are used to reset the processor in different manners.
* A “cold” or “power on” RESET refers to the assertion of RESET while power is initially being applied to the processor.
* A “warm” RESET refers to the assertion of RESET or INIT while VCC and CLK remain within specified operating limits.
* CPU can't simply jump up and start fetching code from flash memory. When external power is first applied, the hardware platform must carry out a number of tasks before the processor can be brought out of its reset state.
* Once the processor reset line has been de-asserted, the processor begins fetching instructions. The location of these instructions is known as the reset vector. The reset vector may contain instructions or a pointer to the starting instructions in flash memory. The location of the vector is architecture-specific and usually in a fixed location, depending on the processor.
* The initial address must be a physical address, as the Memory Management Unit (MMU), if it exists, has not yet been enabled. The first fetching instructions start at 0xFFF, FFF0.
* processor cache is not enabled by default.
* **A multiprocessor system does not truly enter multiprocessing operation until the OS takes over.**
* **While it is possible to do a limited amount of parallel processing during the UEFI boot phase, such as during memory initialization with multiple socket designs, any true multithreading activity would require changes to be made to the Driver Execution Environment (DXE) phase of the UEFI. Without obvious benefits, such changes are unlikely to be broadly adopted.**

7.1 Memory Initialization

* The initialization of the memory controller varies slightly depending on the DRAM technology and the capabilities of the memory controller itself.
* There is a very wide range of DRAM configuration parameters, including number of ranks, eight-bit or 16-bit addresses, overall memory size and constellation, soldered down or add-in module configurations, page-closing policy, and power management.
* If the platform supports add-in-modules for memory, it may use any of a number of standard form factors. The small-outline Dual In-Line Memory Module (DIMM) is often found in embedded systems. The DIMMs provide a serial EPROM that contains DRAM configuration information known as Serial Presence Detect (SPD) data. The firmware reads the SDP data and subsequently configures the device. The serial EPROM is connected via the System Management Bus (SMBus). This means the device must be available in the early initialization phase so the software can establish the memory devices on-board. It is also possible for memory-down motherboards to incorporate SPD EEPROMs to allow for multiple and updatable memory configurations that can be handled efficiently by a single BIOS algorithm. A hard-coded table in one of the MRC files could be used to implement an EEPROM-less design.

**fig-2:RAM initialization**

* The early initialization phase next readies the bootstrap processor and I/O peripherals' base address registers, which are needed to configure the memory controller.
* Once the memory controller has been initialized, a number of subsequent cleanup events take place, including tests to ensure that memory is operational.
* From the reset vector, execution starts directly from nonvolatile flash storage. This operating mode is known as execute-in-place. The read performance of nonvolatile storage is much slower than the read performance of DRAM. The performance of code running from flash is therefore much lower than code executed in RAM. Most firmware is therefore copied from slower nonvolatile storage into RAM. The firmware is then executed in RAM in a process known as shadowing.
* Intel Architecture systems generally do not execute-in-place for anything but the initial boot steps before memory has been configured. The firmware is often compressed, allowing reduction of nonvolatile RAM requirements. Clearly, the processor cannot execute a compressed image in place. There is a trade-off between the size of data to be shadowed and the act of decompression. Pre-fetchers in the processor, if enabled, may speed up execution-in-place. Some SOCs have internal NVRAM cache buffers to assist in pipelining the data from the flash to the processor.
* Before memory is initialized, the data and code stacks are held in the processor cache. Once memory is initialized, the system must exit that special caching mode and flush the cache. The stack will be transferred to a new location in main memory and cache reconfigured.
* The stack must be set up before jumping into the shadowed portion of the BIOS that is now in memory. A memory location must be chosen for stack space. The stack will count down so the top of the stack must be entered and enough memory must be allocated for the maximum stack.
* If the system is in real mode, then set with the appropriate values. If protected mode is used then set to the correct memory location. This is where the code makes the jump into memory. If a memory test has not been performed before this point, the jump could very well be to garbage. System failures indicated by a Power-On Self Test (POST) code between "end of memory initialization" and the first following POST code almost always indicate a catastrophic memory initialization problem. If this is a new design, then chances are this is in the hardware and requires step-by-step debug.
* For legacy option ROMs and BIOS memory ranges, Intel chipsets usually come with memory aliasing capabilities that allow access to memory below 1 MB to be routed to or from DRAM or nonvolatile storage located just under 4 GB.
* For shadowing, if PAM registers remain at default values. Shadowing can be used to improve boot speed. This will direct reads to the flash device and writes to memory. Data can then be shadowed into memory by reading and writing the same address. Once BIOS code has been shadowed into memory, the enables can be changed to read-only mode so memory reads are directed to memory. This also prevents accidental overwriting of the image in memory.
* After memory is started, the remaining processors are initialized and left in a wait-for-SIPI state. To accomplish this, the system firmware must:
  + Find microcode and copy it to memory.
  + Find the CPU code in the Serial Peripherals Interface (SPI) and copy it to memory — an important step to avoid execution-in-place for the remainder of the sequence.
  + Send start-up inter-processor interrupts to all processors.
  + Disable all NEM settings, if this has not already been done.
  + Load microcode updates on all processors.
  + Enable cache-on mode for all processors.

8.0 Option ROM

* Option ROMs (or OpROMs) are firmware run by the PC BIOS during platform initialization. They are usually stored on a plug-in card, though they can reside on the system board.
* Devices that typically require option ROMs are video cards, network adapters, and storage drivers for RAID modules. These option ROMs also typically provide firmware drivers to the PC.
* The Unified Extensible Firmware Interface (UEFI) has support for Legacy mode option ROMs.
* As per latest UEFI specification (currently at 2.3.1 Errata C – section 2.5.1.2), ISA (legacy) option ROMs are not a part of the UEFI Specification. For the purposes of this discussion, only PCI-based UEFI-compatible option ROMs will be considered.
* Option ROMs can be used when it's not be possible to embed a device's firmware in the PC firmware. When the option ROM carries the driver, the IHV can leverage that driver, and keep the driver and device in one place.

9.0 Graphics Enabling

**video BIOS**

* A video BIOS or VBIOS is the Basic Input Output System (BIOS) of a graphics card or the integrated graphics controller in a computer.
* The VBIOS provides a set of video-related functions that are used by programs to access the video hardware. An example of the video hardware is the integrated graphics controller within an Intel Chipset.
* When the computer is started, it usually displays the graphics card vendor, model, BIOS version and the amount of graphics memory.