Booting an Intel System Architecture

Conference Paper · November 2015

CITATIONS
0

READS
7,395

1 author:

Nikola Zlatanov
Applied Materials
44 PUBLICATIONS **10** CITATIONS

SEE PROFILE

Booting an Intel System Architecture

Nikola Zlatanov*

When external power is first applied to a platform, the hardware platform must carry out a number of tasks before the processor can be brought out of reset. The first task is for the power supply to be allowed to settle down to its nominal state; once the primary power supply settles, there are usually a number of derived voltage levels needed on the platform. For example, on the Intel architecture reference platform the input supply consists of a 12-volt source, but the platform and processor require a number of different voltage rails such as 1.5 V, 3.3 V, 5 V, and 12 V. The platform and processor also require that the voltages are provided in a particular sequence. This process is known as *power sequencing*. The power is sequenced by controlling analog switches (typically field effect transistors). The sequence is driven by often driven by a complex program logic device (CPLD). The platform clocks are also derived from a small number of input clock and oscillator sources. The devices use phase locked loop circuitry to generate the derived clocks used for the platform. These clocks also take time to converge. When all these steps have occurred, the power sequencing CPLD de-asserts the reset line to the processor. Figure 1 shows an overview of the platform blocks described. Depending on integration of silicon features, some of this logic may be on chip and controlled by microcontroller firmware, which starts prior to the main processor.

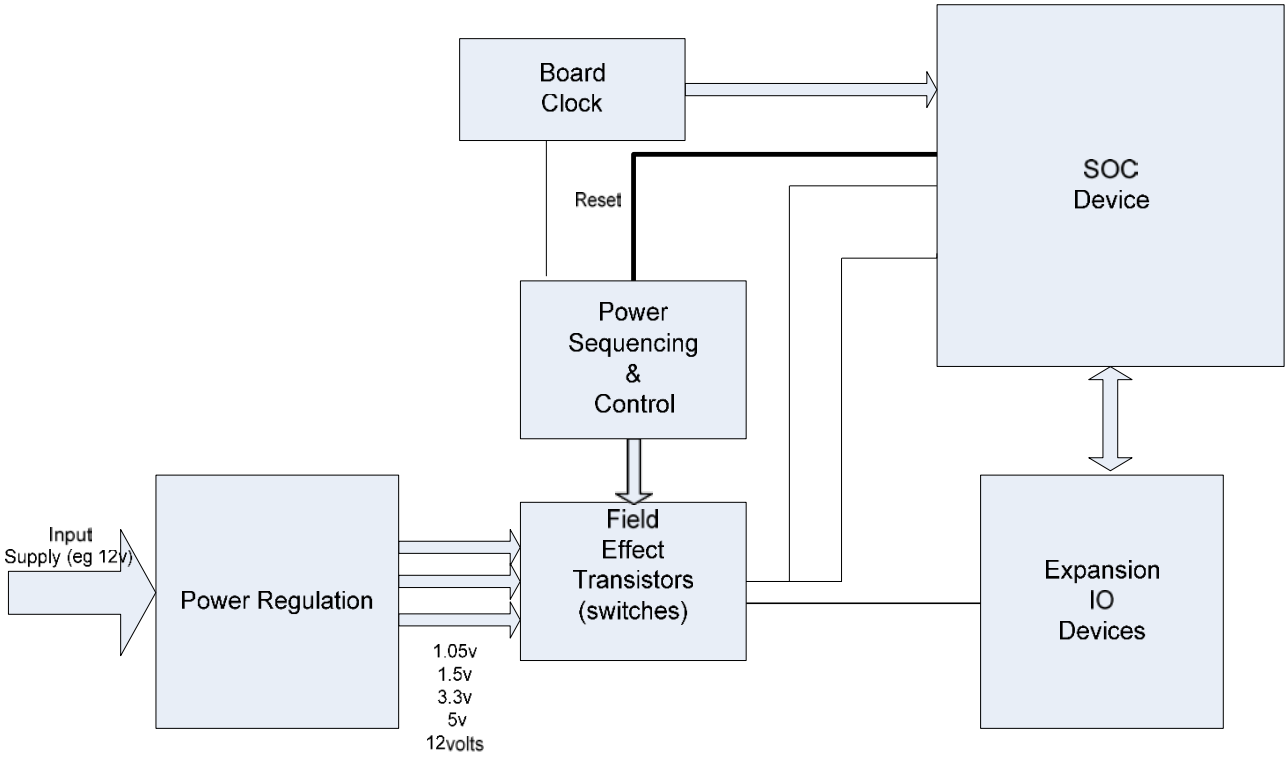

Figure 1 Power Sequencing Overview

Non-Host–based Subsystem Startup

As part of the Intel architecture, a variety of subsystems may begin prior to the main host system starting.

The Intel Manageability Engine (ME), available on some mainstream, desktop, and server derived chips, is one such component. While the main system firmware does not initialize these devices, there is likely to be some level of interactions that must be taken into account in the settings of the firmware, or in the descriptors of the flash component for the ME to start up and enable the clocks correctly. The main system firmware also has the potential to make calls and be called from the ME.

Another example is the micro engines, which are part of telecommunications segment components in the embedded market segments. These micro engines have their own firmware that starts independently of the system BIOS, but the host system BIOS has to make allowances for this in the ACPI memory map to allow for proper interaction between host drivers and the micro engine subsystem.

Starting at the Host Reset Vector

Once the processor reset line has been de-asserted, the processor begins fetching instructions. The location of these initial processor instructions are known as the *reset vector*. The reset vector may contain instructions or a pointer to the actual staring instruction sequence in the 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 MMU (if it exists) has not yet been enabled. For Intel architecture the first fetching instructions start from 0xFFF, FFF0. Only 16 bytes are left to the top of memory, so these 16 bytes must contain a far jump to the remainder of the initialization code.

This code is always written in assembly at this point as there (to date) is no software stack or cache as RAM available at this time.

Because the processor cache is not enabled by default, it is not uncommon to flush cache in this step with a WBINV instruction. The WBINV instruction is not needed on newer processors, but it doesn't hurt anything.

Mode Selection

IA-32 supports three operating modes and one quasi-operating mode:

• *Protected mode* is considered the native operating mode of the processor. It provides a rich set of architectural features, flexibility, high performance and backward compatibility to the existing software base.

• *Real-address mode* or "real mode" is operating mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to switch to protected or system management mode). Whenever a reset or a power on happens, the system transitions back to Real-address mode.

• *System management mode (SMM)* is a standard architectural feature in all IA-32 processors, beginning with the Intel386 SL processor. This mode provides an operating system or executive with a transparent mechanism for implementing power management and OEM differentiation features. SMM is entered through activation of an external system interrupt pin (SMI#), which generates a system management interrupt (SMI). In SMM, the processor switches to a separate address space while saving the context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the SMI.

• Normally the system firmware creates an SMI handler, which may periodically take over the system from the host OS. There are normally legitimate workarounds that are executed in the SMI handler and handling and logging-off errors that may happen at the system level. As this presents a potential security issue, there is also a lock bit that resists tampering with this mechanism.

• Real-time OS vendors often recommend disabling this feature because it has a potential of subverting the nature of the OS environment. If this happens, then the additional work of the SMI handler would either need to be incorporated into the RTOS for that platform or else the potential exists of missing something important in the way of error response or workarounds. If the SMI handler can work with the RTOS development, there are some advantages to the feature.

• *Virtual-8086 mode* is a quasi-operating mode supported by the processor in protected mode.

This mode allows the processor execute 8086 software in a protected, multitasking environment.

Intel® 64 architecture supports all operating modes of IA-32 architecture and IA-32e modes:

• *IA-32e mode*—in IA-32e mode, the processor supports two sub-modes: compatibility mode and 64-bit mode.
• 64-bit mode provides 64-bit linear addressing and support for physical address space larger than 64 GB.
• Compatibility mode allows most legacy protected-mode applications to run unchanged.

Figure 2 shows how the processor moves between operating modes.

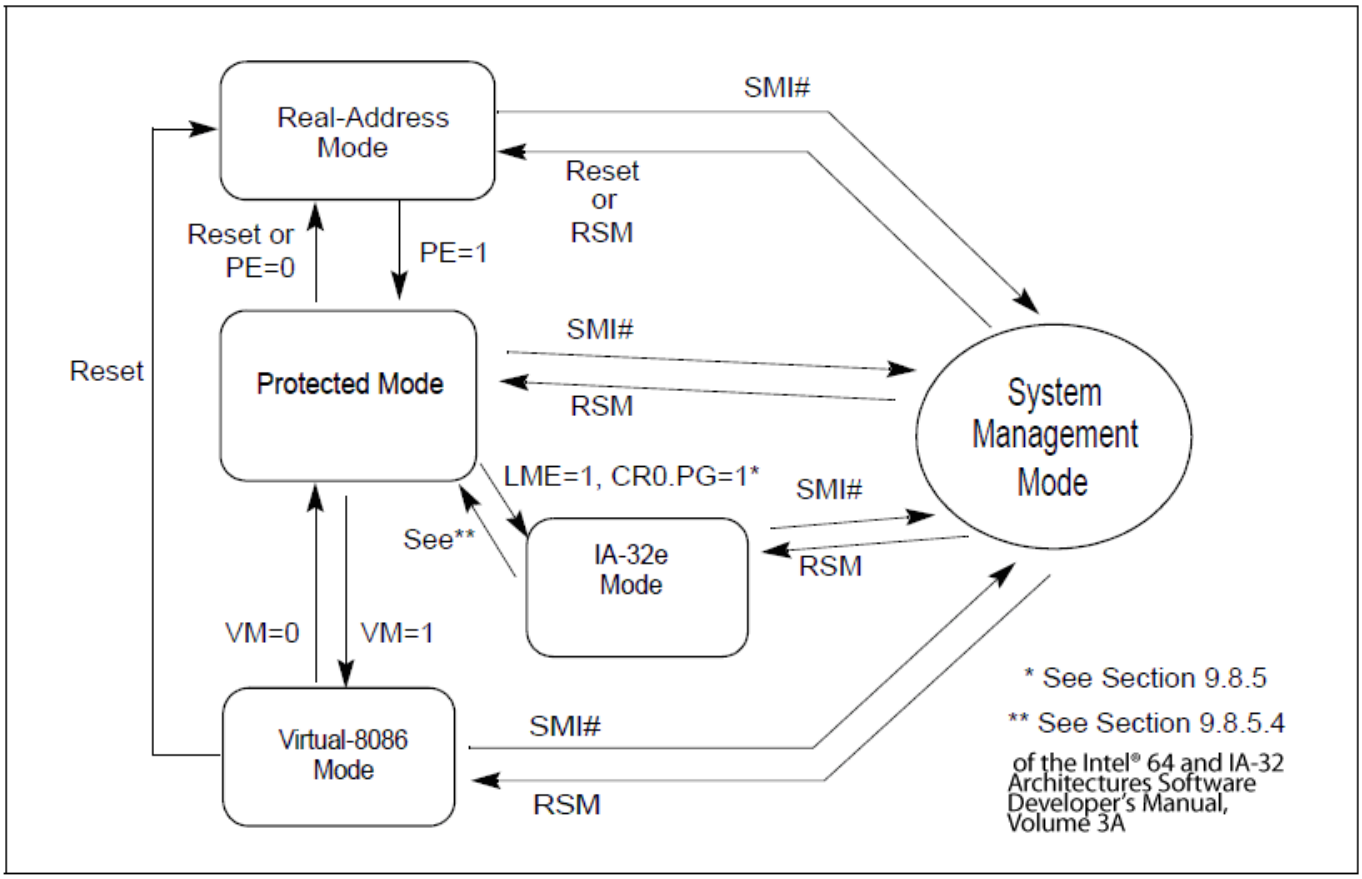

Figure 2 Switching Processor Operating Modes

Refer to the *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A* section titled "Mode Switching" for more details.

When the processor is first powered on, it will be in a special mode similar to real mode, but with the top 12 address lines being asserted high. This aliasing allows the boot code to be accessed directly from NVRAM (physical address 0xFFFxxxxx).

Upon execution of the first long jump, these 12 address lines will be driven according to instructions by firmware. If one of the protected modes is not entered before the first long jump, the processor will enter real mode, with only 1 MB of addressability. In order for real mode to work without memory, the chipset needs to be able to alias a range of memory below 1 MB to an equivalent range just below 4 GB to continue to access NVRAM. Certain chipsets do not have this aliasing and may require a switch to into a normal operating mode before performing the first long jump. The processor also invalidates the internal caches and translation lookaside buffers (TLBs).

The processor continues to boot in real mode today, for no particular technical requirement. While some speculate it is to ensure that the platform can boot legacy code (example DOS OS) written many years ago, it is more an issue of introducing and needing to validate the change into a broad ecosystem of players and developers, and backward compatibility issues it would create in test and manufacturing environments and other natural upgrade hurdles that will continue to keep the boot mode and Intel reset vector "real."

The first power-on mode is actually a special subset of the real mode. The top 12 address lines are held high, thus allowing aliasing, where the processor can execute code from the nonvolatile storage (such as flash) located within lowest one megabyte as if from the top of memory. Normal operation of the firmware (such as BIOS) is to switch modes to flat protected mode as early in the boot sequence as is possible. Once the processor is running in protected mode, it is usually not necessary to switch back to real mode unless executing a legacy option ROM, which makes certain legacy software interrupt calls. The flat mode runs 32-bit code and the physical address are mapped on to one with the logical addresses (paging off). The Interrupt Descriptor Table is used for interrupt handling. This is the recommended mode for all BIOS/boot loaders to operate in. The segmented protected mode is not used for the initialization code as part of the BIOS sequence.

Intel produces BIOS specifications or BIOS writer's guides that go into some details about chip-specific and technology-specific initialization sequences. These documents hold fine-grain details of every bit that needs to be set, but not necessarily the high level sequence in which to set them. The following flows for early initialization and advanced initialization outline that from a hardware architecture point of view.

Early Initialization

The early phase of the BIOS/bootloader will do the minimum to get the memory and processor cores initialized.

In a system BIOS constructed using the Unified Extensible Firmware Interface (UEFI) 2.0 framework, the Security (SEC) and the pre-EFI initialization (PEI) phases are normally synonymous with "early initialization." It doesn't matter if legacy or UEFI BIOS is used; from a hardware point of view, the early initialization sequence is the same for a given system.

Single-Threaded Operation

In a multi-core system, the bootstrap processor is the CPU core/thread that is chosen to boot the normally single-threaded system firmware. At RESET, all of the processors race for a semaphore flag bit in the chipset. The first finds it clear and in the process of reading it sets the flag; the other processors find the flag set and enter a WAIT for SIPI or halt state. The first processor initializes main memory and the application processors (APs) and continues with the rest of boot. A multiprocessor (MP) system does not truly enter MP operation

6 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 DXE phase of the UEFI solutions to allow for this. In order to have broad adoption, some obvious benefits would need to arise.

Simple Device Initialization

The early initialization phase readies the bootstrap processor (BSP) and I/O peripherals' base address registers needed to configure the memory controller.

The device-specific portion of an Intel architecture memory map is highly configurable. Most devices are seen and accessed via a logical PCI bus hierarchy, although a small number may be memory-mapped devices that have part-specific access mechanisms. Device control registers are mapped to a predefined I/O or MMIO space and can be set up before the memory map is configured. This allows the early initial firmware to configure the memory map of the device needed to set up DRAM. Before DRAM can be configured, the firmware must establish the exact configuring of DRAM that is on the board. SOC devices based on other processor architectures typically provide a static address map for all internal peripherals, with external devices connected via a bus interface. The bus-based devices are mapped to a memory range within in the SOC address space. These SOC devices usually provide a configurable chip select register set to specify the base address and size of the memory range enabled by the chip select. SOCs based on Intel architecture primarily use the logical PCI infrastructure for internal and external devices.

The location of the device in the host memory address space is defined by the PCI base address register (BAR) for each of the devices. The device initialization typically enables all the BAR registers for the devices required as part of the system boot path. BIOS will assign all devices in the system a PCI

base address by writing the appropriate BAR registers during PCI enumeration. Long before full PCI enumeration, the BIOS must enable PCIe BAR and the PCH RCBA BAR for memory, I/O and MMIO interactions during the early phase of boot.

Depending on the chipset, there are prefetchers that can be enabled at this point to speed up the data transfer from the flash device. There may also be DMI link settings that have to be tuned to optimal performance.

CPU Initialization

This consists of simple configuring of processor and machine registers, loading a microcode update, and enabling the Local APIC.

Microcode Update. Microcode is a hardware layer of instructions involved in the implementation of the machine-defined architecture. It is most prevalent in CISC-based processors. Microcode is developed by the CPU7 vendor and incorporated into an internal CPU ROM during manufacture. Since the infamous "Pentium flaw," Intel processor architecture allows that microcode to be updated in the field either through a BIOS update or via an OS update of "configuration data."

Today an Intel processor must have the latest microcode update to be considered a warranted CPU. Intel provides microcode updates that are written to the writable microcode store in the CPU. The updates are encrypted and signed by Intel such that only the processor that the microcode update was designed for can authenticate and load that update. On socketed systems, the BIOS may have to carry many flavors of microcode update depending on the number of processor steppings supported. It is important to load the microcode updates early in the boot sequence to limit the exposure of the system to any known errata in the silicon. It is important to note that the microcode update may need to be reapplied to the CPU after certain reset events in the boot sequence.

The BSP microcode update must be loaded before No-Eviction mode is entered.

Local APIC. The Local APIC must be enabled to handle any interrupts that occur early in post, before enabling protected mode. For more information on the Intel APIC Architecture and Local APICs, please see the *Intel 64 and IA-32 Intel Architecture Software Developer's Manual, Volume 3A: System Programming Guide, Chapter 8*.

Switch to Protected Mode. Before the processor can be switched to protected mode, the software initialization code must load a minimum number of protected mode data structures and code modules into memory to support reliable operation of the processor in protected mode. These data structures include the following:
An IDT
A GDT
A TSS
(Optional) An LDT
If paging is to be used, at least one page directory and one page table
A code segment that contains the code to be executed when the processor switches to protected mode
One or more code modules that contain the necessary interrupt and exception handlers

Initialization code must also initialize the following system registers before the processor can be switched to protected mode:
The GDTR
Optionally the IDTR. This register can also be initialized immediately after switching to protected mode, prior to enabling interrupts.
Control registers CR1 through CR4
The memory type range registers (MTRRs)

With these data structures, code modules, and system registers initialized, the processor can be switched to protected mode by 8 loading control register CR0 with a value that sets the PE flag (bit 0). From this point onward, it is likely that the system will not enter 16b Real mode again, Legacy Option ROMs and Legacy OS/BIOS interface notwithstanding, until the next hardware reset is experienced.

More details about protected mode and real mode switching can be found in the *Intel 64 and IA-32 Intel Architecture Software Developer's Manual, Volume 3A: System Programming Guide, Chapter 9*.

Cache as RAM and No Eviction Mode. Since no DRAM is available, the code initially operates in a stackless environment. Most modern processors have an internal cache that can be configured as RAM (Cache as RAM or CAR) to provide a software stack. Developers must write extremely tight code when using CAR because an eviction would be unacceptable to the system at this point in the boot sequence; there is no memory to maintain coherency with at this time. There is a special mode for processors to operate in cache as RAM called "no evict mode" (NEM) where a cache line miss in the processor will not cause an eviction. Developing code with an available software stack is much easier, and initialization code often performs the minimal setup to use a stack even prior to DRAM initialization.

Processor Speed Correction. The processor may boot into a slower mode than it is capable of performing for various reasons. It may be considered less risky to run in a slower mode, or it may be done to save additional power. It will be necessary for the BIOS to initialize the speed step technology of the processor and may need to force the speed to something appropriate for a faster boot. This additional optimization is optional; the OS will likely have the drivers to deal with this parameter when it loads.

Memory Configuration

The initialization of the memory controller varies slightly depending on the DRAM technology and the capabilities of the memory controller itself. The information on the DRAM controller is proprietary for SOC devices, and in such cases the initialization memory reference code (MRC) is typically supplied by the SOC vendor. Developers have to contact Intel to request access to the low level information required. Without being given the MRC, developers can understand that for a given DRAM technology, they must follow a JEDEC initialization sequence. It is likely a given that the code will be single point of entry and single point of exit code that has multiple boot paths contained within it. It will be 32-bit protected mode code. The settings for various bit fields like buffer strengths and loading for a given number of banks of memory is something that is chipset-specific. The dynamic nature of the memory tests that are run (to ensure proper timings have been applied for a given memory configuration) is an additional complexity that would prove difficult to replicate without a proper code from the memory controller vendor. Workarounds for errata and interactions with other subsystems such as the Manageability Engine are not something that can be re-invented, 9 but can be reverse-engineered. The latter is of course heavily frowned upon.

There is a very wide range of DRAM configuration parameters, such as number of ranks, 8-bit or 16-bit addresses, overall memory size, and constellation, soldered down or add-in module (DIMM) configurations, page closing policy, and power management. Given that most embedded systems populate soldered down DRAM on the board, the firmware may not need to discover the configuration at boot time. These configurations are known as *memory-down*. The firmware is specifically built for the target configuration. This is the case for the Intel reference platform from the Embedded Computing Group. At current DRAM speeds, the wires between the memory controllers behave like transmission lines; the SOC may provide automatic calibration and runtime control of resistive compensation (RCOMP) and delay locked look (DLL) capabilities. These capabilities allow the memory controller to change elements such as the drive strength to ensure error-free operation over time and temperature variations.

If the platform supports add-in-modules for memory, there are a number of standardized form factors for such memory. The small outline dual in-line memory module (SODIMM) is one such form factor often found in embedded systems. The DIMMs provide a serial EPROM. The serial EPROM devices contain the DRAM configuration data. The data is known as serial presence detect data (SPD data). The firmware reads the SDP data to identify the device configuration and subsequently configures the device. The serial EPROM is connected via SMBUS, thus the device must be available in this early initialization phase, so the software can establish the memory devices on board. It is possible for memory-down motherboards to also incorporate serial presence detect EEPROMs to allow for multiple and updatable memory configurations to be handled efficiently by a single BIOS algorithm. It is also possible to provide a hard-coded table in one of the MRC files to allow for an EEPROM-less design. In order to derive that table for SPD, please see Appendix A.

Post-Memory

Once the memory controller has been initialized, a number of subsequent cleanup events take place.

Memory Testing

The memory testing is now part of the MRC, but it is possible to add more tests should the design merit it. BIOS vendors typically provide some kind of memory test on a cold boot. Writing custom firmware requires the authors to choose a balance between thoroughness and speed, as highly embedded/mobile devices require extremely fast boot times and memory testing can take up considerable time.

If testing is warranted for a design, testing the memory directly following initialization is the time to do it. The system is idle, the subsystems are not actively accessing memory, and the OS has not taken over the host side of the platform. Memory errors manifest themselves in random ways, sometimes inconsistently. Several hardware features 10 can assist in this testing both during boot and during runtime. These features have traditionally been thought of as high-end or server features, but over time they have moved into the client and embedded markets.

One of the most common is ECC. Some embedded devices use error correction codes (ECC) memory, which may need extra initialization. After power up, the state of the correction codes may not reflect the contents and all memory must be written to; writing to memory ensures that the ECC bits are valid and sets the ECC bits to the appropriate contents. For security purposes, the memory may need to be zeroed out manually by the BIOS or in some cases a memory controller may incorporate the feature into hardware to save time.

Depending on the source of the reset and security requirements, the system may not execute a memory wipe or ECC initialization. On a warm reset sequence, memory context can be maintained.

If there were any memory timing changes or other configuration changes that require a reset to take effect, this is normally the time to execute a warm reset. That warm reset would start the early initialization phase over again; registers would need to be restored that are affected.

Shadowing

From the reset vector, execution starts off executing directly from the nonvolatile flash storage (NVRAM). This operating mode is known as execute in place (XIP). The read performance of nonvolatile storage is much slower than the read performance of DRAM. The performance of the code running from flash is much lower than if it executed from RAM, so most early firmware will copy from the slower nonvolatile storage into RAM. The firmware starts to run the RAM copy of the firmware. This process is sometimes known as *shadowing*. Shadowing involves having the same contents in RAM and flash; with a change in the address decoders the RAM copy is logically in front of the flash copy and the program starts to execute from RAM. On other embedded systems, the chip selects ranges that are managed to allow the change from flash to RAM execution. Most computing systems run as little as possible in place. However, some embedded platforms constrained (in terms of RAM) execute all the application in place. This is generally an option on very small embedded devices. The Intel architecture platforms generally do no execute in place for anything but the very initial boot steps before memory has been configured. The firmware is often compressed instead of a simple copy. This allows reduction of the NVRAM requirements for the firmware. Clearly the processor cannot execute a compressed image in place. On Intel architecture platforms the copy of the firmware is usually located below 1 MB.

There is a tradeoff between the size of data to be shadowed and the act of decompression. The decompression algorithm may take much longer to load and execute than it would be for the image to remain uncompressed. Prefetchers in the processor, if enabled, may also speed up execution in place and some SOCs have internal NVRAM cache buffers to assist in pipelining the data from the flash to the processor.

Figure 3 shows the memory map at initialization in real mode. Real mode has an accessibility limit of 1MB.

Exit from No-Eviction Mode and Transfer to DRAM

Before memory was initialized, the data and code stacks were held in the processor cache. With memory now initialized that special and temporary caching mode must be exited and the cache flushed. The stack will be transferred to a new location in system main memory and cache reconfigured as part of AP initialization.
The stack must be set up before jumping into the shadowed portion of the BIOS that now is 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 SS:SP must be set with the appropriate values. If protected mode is used, which is likely the case following MRC execution, then SS:ESP must be set to the correct memory location.

Transfer to DRAM

This is where the code makes the jump into memory. As mentioned before, if a memory test has not been performed up until this point, the jump could very well be to garbage. System failures indicated by a POST code between “end of memory initialization” and the first following POST code almost always indicates 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.

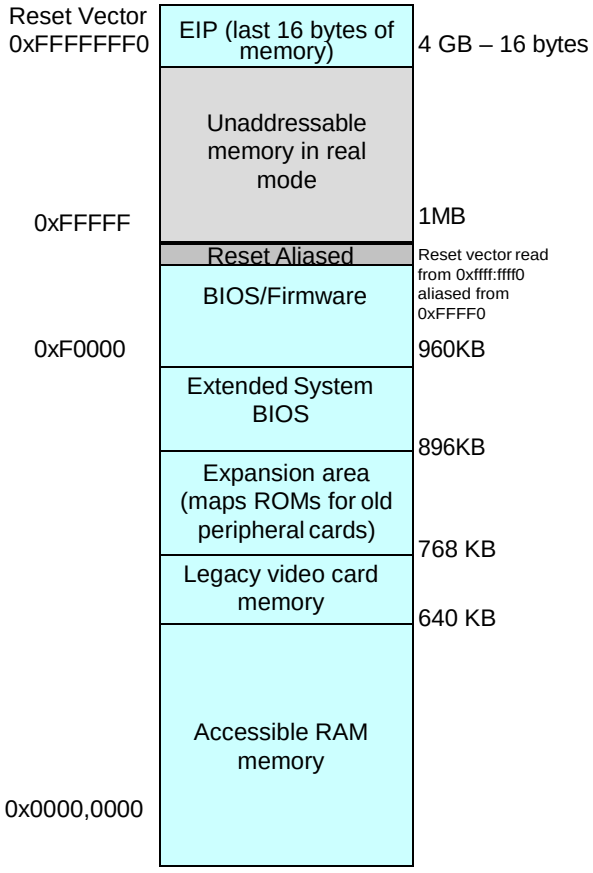

Figure 3 Intel® Architecture Memory Map at Power On

Memory Transaction Re-Direction

For Legacy option ROMs and BIOS memory ranges, Intel chipsets usually come with memory aliasing capabilities that allow reads and writes to sections of memory below 1 MB to be either routed to/from DRAM or nonvolatile storage located just under 4 GB. The registers that control this aliasing are typically referred to as programmable attribute maps (PAMs). Manipulation of these registers may be required

before, during, and after firmware shadowing. The control over the redirection of memory access varies from chipset to chipset. For example, some chipsets allow control over reads and writes, while others only allow control over reads.
For shadowing, if PAM registers remain at default values (all 0s), all FWH accesses to the E and F segments (E_0000–F_FFFFh) will be directed downstream toward the flash component.
This will function to boot the system, but is very slow. Shadowing, as we know, improves boot speed.
One method of shadowing the E and F segments (E_0000–F_FFFFh) of BIOS is to utilize the PAM registers. This can be done by changing the enables (HIENABLE[], LOENABLE[]) to be 10 (write only). This will direct reads to the flash device, and writes to memory. By reading and then writing the same address, the data is shadowed into memory. Once BIOS code has been shadowed into memory, the enables can be changed to '01(read only) so memory reads are directed to memory.
This also prevents accidental overwriting of the image in memory. See the example in Table 1.

PAM Registers	Register Address
PAM0	0F0000–0FFFFFh
PAM1	0C0000–0C7FFFh
PAM2	0C8000–0CFFFFh
PAM3	0D0000–0D7FFFh
PAM4	0D8000–0DFFFFh
PAM5	0E0000–0E7FFFh
PAM6	0E8000–0EFFFFh

Table 1 Address Ranges of the Programmable Attribute Map

Consult the chipset datasheet for details on the memory redirection feature controls applicable to the target platform.

Application Processor (AP) Initialization

Even in SOCs, there is the likelihood of having multiple CPU cores, which are considered BSP + AP to system initialization. While the BSP starts and initializes the system, the application processors (APs) must also be initialized with identical features enabled to the BSP. Prior to memory, the APs are left uninitialized. After memory is started, the remaining processors are initialized and left in a WAIT for SIPI state. To do this the system firmware must:

1. Find microcode and then copy it to memory
2. Find the CPU code in SPI and copy to memory—an important step to avoid XIP for the remainder of the sequence
3. Send Startup IPIs to all processors
4. Disable all NEM settings, if this has not already been done
5. Load microcode updates on all processors
6. Enable Cache On for all processors

Although partial details of these sequences are in the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, more complete details can be found in the BIOS Writer's Guide for that particular processor or on CPU reference code that may be obtained from Intel.From a UEFI perspective the AP initialization may either be part of the PEI or the DXE phase of the boot flow, or in the early or advanced initialization. At the time of this printing, there is some debate as the final location.

CPUID—Threads and Cores

Since Intel processors are packaged in various configurations, there are different terms that must be understood when considering processor initialization:

Thread. A logical processor that shares resources with another logical processor in the same physical package.

Core. A processor that coexists with another processor in the same physical package that does not share any resources with other processors.

Package. A "chip" that contains any number of cores and threads.

Threads and cores on the same package are detectable by executing the CPUID instruction.
See the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* for details on the information available with the CPUID instruction on various processor families.
Detection of additional packages must be done "blindly." If a design must accommodate more than one physical package, the BSP needs to wait a certain amount of time for all potential APs in the system to "log in." Once a timeout occurs or the maximum expected number of processors "log in," it can be assumed that there are no more processors in the system.

Startup Inter-Processor Interrupt (SIPI)

In order to wake up secondary threads or cores, the BSP sends a SIPI to each thread and core. This SIPI is sent by using the BSP's LAPIC, indicating the physical address from which the application processor (AP) should start executing. This address must be below 1 MB of memory and must be aligned on a 4-KB boundary.

AP Wakeup State

Upon receipt of the SIPI, the AP starts executing the code pointed to by the SIPI message. As opposed to the BSP, when the AP starts code execution it is in real mode. This requires that the location of the code that the AP starts executing is located below 1 MB.

Wakeup Vector Alignment

The starting execution point of the AP has another architectural restriction that is very important and is commonly forgotten. The entry point to the AP initialization code must be aligned on a 4-KB boundary. Refer to the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* section "MP Initialization Protocol Algorithm for Intel Xeon Processors."

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* section "Typical AP Initialization Sequence" illustrates what is typically done in by the APs after receiving the SIPI.

Caching Considerations

Because of the different types of processor combinations and different attributes of shared processing registers between threads, care must be taken to ensure that the caching layout of all processors in the entire system remain consistent such that there are no caching conflicts.

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* section "MTRR Considerations in MP Systems" outlines a safe mechanism for changing the cache configuration in all systems that contain more than one processor. It is recommended that this be used for any system with more than one processor present.

AP Idle State

Behavior of APs during firmware initialization is dependent on the firmware implementation, but is most commonly restricted to short durations of initialization followed by entering a halt state with a HLT instruction, awaiting direction from the BSP for another operation.

Once the firmware is ready to attempt to boot an OS, all AP processors must be placed back in their power-on state (WAIT for SIPI), which can be accomplished by the BSP sending an INIT ASSERT IPI followed by an INIT DEASSERT IPI to all APs in the system (all except self).

Advanced Initialization

The advanced device initialization follows the early initialization and basically ensures that the DRAM is initialized. This second stage is focused on device-specific initialization. In a UEFI-based BIOS solution, advanced initialization tasks are also known as Dynamic Execute Environment (DXE) and Boot Device Selection (BDS) phases. The following devices must be initialized in order to enable an embedded system. Not all are applicable to all embedded systems, but the list is prescriptive for most and is particular to an SOC based on Intel architecture.

General purpose I/O (GPIO)
Interrupt controller
Timers
Cache initialization (could also be done during early initialization)
Serial ports, Console In/Out
Clocking and overclocking
PCI bus initialization
Graphics (optional)
USB
SATA

General Purpose I/O (GPIO) Configuration

GPIOs are key to the extensibility of the platform. As the name implies, GPIOs can be configured for either input or output, but can also be configured for a native functionality. Depending on weak or strong pull-up or pull-down resistors, some of the GPIOs can also act like strapping pins, which are sampled at RESET by the chipset, and can have a second meaning during boot and runtime. GPIOs may also act like sideband signals to allow for system wakes. GPIO 27 is used for this on most mainstream platforms.

System-on-chip devices are designed to be used in a large number of configurations, the devices often having more capabilities than the device is capable of exposing on the I/O pins concurrently. That is because several functions are multiplexed to a particular I/O pin. The configuration of the pins must be set before use. The pins are configured to either provide a specific function or serve as a general purpose I/O pin. I/O pins on the device are used to control logic or behavior on the device. General purpose I/O pins can be configured as input or output pins. The status and control is provided by GPIO control registers.

The system firmware developer must work through between 64 and 256 GPIOs and their individual options with the board designer (per platform) to ensure that this feature is properly enabled.

Interrupt Controllers

Intel architecture has several different methods of interrupt handling. The following or a combination of the following can be used to handle interrupts:

Programmable Interrupt Controller (PIC) or 8259
Local Advanced Programmable Interrupt Controller (APIC)
Input/Output Advanced Programmable Interrupt Controller (IOxAPIC)
Messaged Signaled Interrupt (MSI)

Programmable Interrupt Controller (PIC)

When the PIC is the only interrupt device enabled, it is referred to as PIC Mode. This is the simplest mode where the PIC handles all the interrupts. All APIC components are bypassed and the system operates in single-thread mode using LINT0.
The BIOS must set the IRQs per board configuration for all onboard, integrated, and add-in PCI devices. The PIC contains two cascaded 8259s with fifteen available IRQs. IRQ2 is not available since it is used to connect the 8259s. On mainstream components, there are 8 PIRQ pins supported in PCH, named PIRQ[A# :H#], that route PCI interrupts to IRQs of the 8259 PIC. PIRQ[A#:D#] routing is controlled by PIRQ Routing Registers 60h–63h (D31:F0:Reg 60- 63h). The PIRQ[E# : H#] routing is controlled by PIRQ Routing Registers 68h–6Bh (D31:F0:Reg 68 – 6Bh). See Figure 4.

The PCH also connects the 8 PIRQ[A# : H#] to 8 individual IOxAPIC input pins, as shown in Table 2.

PIRQ# Pin	Interrupt Router Register for PIC	Connected to IOxAPIC Pin
PIRQA#	D31:F0:Reg 60h	INTIN16
PIRQB#	D31:F0:Reg 61h	INTIN17
PIRQC#	D31:F0:Reg 62h	INTIN18
PIRQD#	D31:F0:Reg 63h	INTIN19
PIRQE#	D31:F0:Reg 68h	INTIN20
PIRQF#	D31:F0:Reg 69h	INTIN21
PIRQG#	D31:F0:Reg 6Ah	INTIN22
PIRQH#	D31:F0:Reg 6Bh	INTIN23

Table 2 Platform Controller Hub (PCH) PIRQ Routing Table

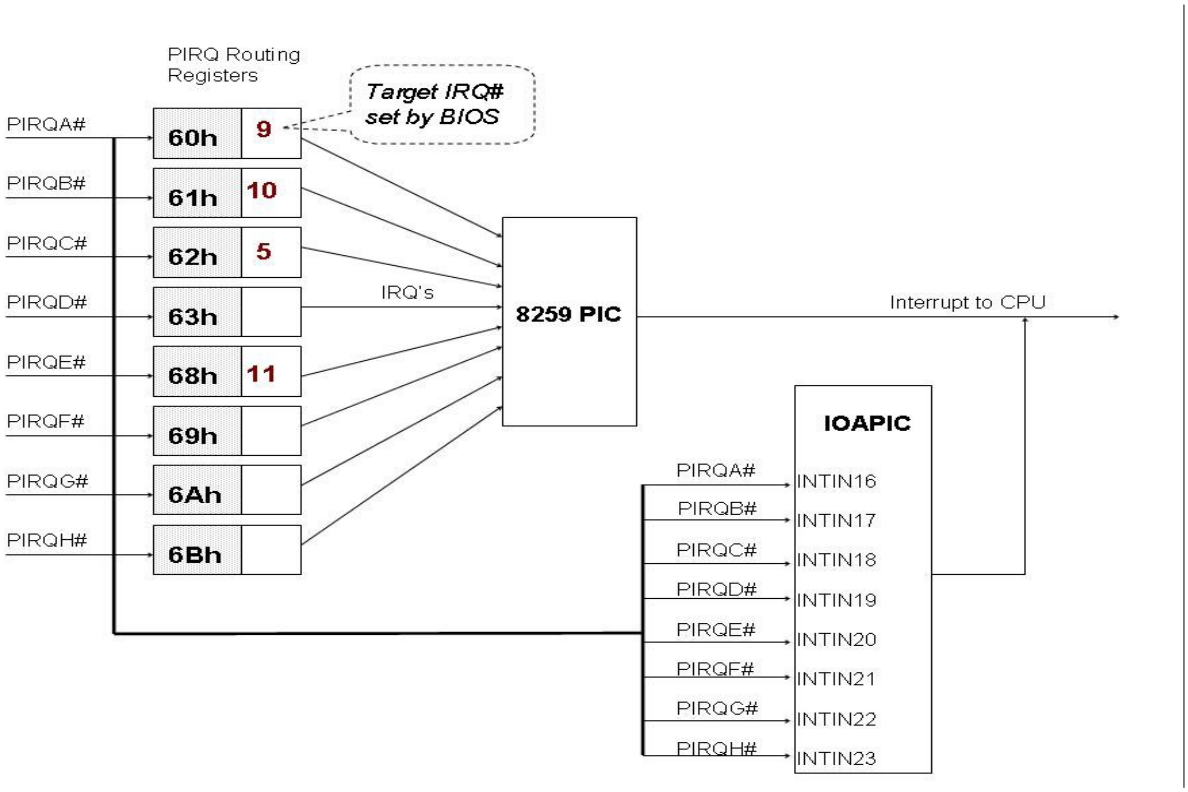

Figure 4 Platform Controller Hub (PCH) PIRQ to IRQ Router

Local Advanced Programmable Interrupt Controller (LAPIC)

The local APIC is contained inside the processor and controls the interrupt delivery to the processor. Each local APIC contains is own set of associated registers as well as a Local Vector Table (LVT). The LVT specifies the manner in which the interrupts are delivered to each processor core.

Refer to the *Intel® 64 and IA-32 Architectures Software Developer's Manual* for more information on initializing the local APIC.

I/O Advanced Programmable Interrupt Controller (IOxAPIC)

The IOxAPIC is contained in the ICH/IOH and expands the number of IRQs available to 24. Each IRQ has an associated redirection table entry that can be enabled or disabled and selects the IDT vector for the associated IRQ. This mode is only available when running in protected mode.
Refer to the Chipset BIOS Writer's Guide for more information on initializing the IOxPIC.

Message Signaled Interrupt (MSI)

The boot loader does not typically use MSI for interrupt handling.

Interrupt Vector Table (IVT)

The IVT is the Interrupt Vector Table located at memory location 0p and containing 256 interrupt vectors. The IVT is used in real mode. Each vector address is 32 bits and consists of the CS:IP for the interrupt vector. Refer to the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Volume 3A section titled "Exception and Interrupt Reference" for a list of real mode interrupts and exceptions.

Interrupt Descriptor Table (IDT)

The IDT is the Interrupt Descriptor Table and contains the exceptions and interrupts in protected mode. There are also 256 interrupt vectors, and the exceptions and interrupts are defined in the same locations as the IVT. Refer to the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Volume 3A for a detailed description of the IDT.

Exceptions

Exceptions are routines that run to handle error conditions. Examples include page fault and general protection fault. At a minimum, placeholders (dummy functions) should be used for each exception handler. Otherwise the system could exhibit unwanted behavior if an exception is encountered that isn't handled.

Real Mode Interrupt Service Routines (ISRs)

Real mode ISRs are used to communicate information between the boot loader and the OS. For example INT10h is used for video services such as changing video modes and resolution. There are some legacy programs and drivers that assume these real mode ISRs are available and directly call the INT routine.

Timers

There are a variety of timers that can be employed on today's Intel Architecture system.

Programmable Interrupt Timer (PIT)

The PIT (8254) resides in the IOH/ICH and contains the system timer also referred to as IRQ0. Refer to the chipset datasheet for more details.

High Precision Event Timer (HPET)

HPET resides in the IOH/ICH and contains three timers. Typically the boot loader does not need to do any initialization of HPET and the functionality is used only by the OS. Refer to the Chipset BIOS Writer's Guide for more details.

Real Time Clock (RTC)

The RTC resides in the IOH/ICH and contains the system time (seconds/minutes/hours/and so on). These values are contained in CMOS, which is explained later. The RTC also contains a timer that can be utilized by firmware. Refer to the appropriate chipset datasheet for more details.

System Management TCO Timer

The TCO timers reside in the IOH/ICH and contain the Watch Dog Timer (WDT). The WDT can be used to detect system hangs and will reset the system.

Note: It is important to understand that for debugging any type of firmware on Intel architecture chipsets that implement a TCO Watch Dog Timer that it should be disabled by firmware as soon as possible coming out of reset. Halting system for debug prior to disabling this Watch Dog Timer on chipsets that power on with this timer enabled will result in system resets, which doesn't allow firmware debug. The OS will re-enable the Watch Dog Timer if it so desires. Consult the chipset datasheet for details on the specific implementation of the TCO Watch Dog Timer. Refer to the Chipset BIOS Writer's Guide for more details.

Local APIC (LAPIC) Timer

The Local APIC contains a timer that can be used by firmware. Refer to the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A for a detailed description of the Local APIC timer.

Memory Caching Control

Memory regions that must have different caching behaviors applied will vary from design to design. In the absence of detailed caching requirements for a platform, the following guidelines provide a "safe" caching environment for typical systems:

1. Default Cache Rule – Uncached.
2. 00000000-0009FFFF – Write Back.
3. 000A0000-000BFFFF – Write Combined / Uncached
4. 000C0000-000FFFFF – Write Back / Write Protect
5. 00100000-TopOfMemory – Write Back.
6. TSEG – Cached on newer processors.
7. Graphics Memory – Write Combined or Uncached.
8. Hardware Memory-Mapped I/O – Uncached.

While MTRRs are programmed by the BIOS, Page Attribute Tables (PATs) are used primarily with the OS to control caching down to the page level.

Serial Ports

An RS-232 serial port or UART 16550 is initialized for either runtime or debug solutions. Unlike USB ports, which require considerable initialization and a large software stack, serial ports have a minimal register-level interface. A serial port can be enabled very early in POST to provide serial output support.

Console In/Console Out

During the DXE phase, part of the boot services includes console in and console out protocols.

Clock and Overclock Programming

Depending on the clocking solution of the platform, the BIOS may have to enable the clocking of the system. It is possible that a subsystem such as the Manageability Engine or baseboard management controller (BMC) in server platforms has this responsibility. It is also possible that beyond the basic clock programming, there may be expanded configuration options for overclocking such as:
Based on enumeration enable/disable clock output enables
Adjust clock spread settings. Enable/disable and adjust amount. Note: settings are provided as fixed register values determined from expected usages.
Under-clock CPU for adaptive clocking support. If done directly, the BIOS must perform adjustment with ramp algorithm.
Lock out clock registers prior to transitioning to host OS.

PCI Device Enumeration

Peripheral Connect Interface (PCI) device enumeration is a generic term that refers to detecting and assigning resources to PCI-compliant devices in the system. The discovery process assigns the resources needed by each device including the following:

Memory, prefetchable memory, I/O space
Memory mapped I/O (MMIO) space
IRQ assignment
Expansion ROM detection and execution

PCI device discovery applies to all the newer (non-legacy) interfaces such as PCI Express (PCIe) root ports, USB controllers, SATA controllers, audio controllers, LAN controllers, and various add-in devices. These newer interfaces all comply with the PCI specification.

Refer to the PCI Specification for more details. A list of all the applicable specifications is in the References section.

It is interesting to note that in the UEFI system, the DXE phase does not execute the majority of drivers, but it is the BDS Phase which executes most of the required drives in UEFI to allow the system to boot.

Graphics Initialization

If the platform has a head, then the video BIOS or Graphics Output Protocol UEFI driver is normally the first option ROM to be executed in the string. Once the main console out is up and running, the console in can be configured.

Input Devices

Refer to the board schematics to determine which I/O devices are in the system. Typically a system will contain one or more of the following devices.

Embedded Controller (EC)

An embedded controller is typically used in mobile or low power systems. The EC contains separate firmware that controls the power management functions for the system as well as PS/2 keyboard functionality. Refer to the specific EC datasheet for more details.

Super I/O (SIO)

An SIO typically controls the PS/2, serial, and parallel interfaces. Most systems still support some of the legacy interfaces rather than implementing a legacy-free system. Refer to the specific SIO datasheet for details on programming information.

Legacy-Free Systems

Legacy-free systems use USB as the input device. If pre-OS keyboard support is required, then the legacy keyboard interfaces must be 22 trapped. Refer to the IOH/ICH BIOS Specification for more details on legacy-free systems.

USB Initialization

The USB controller supports both EHCI and now XHCI. To enable the host controller for standard PCI resources is relatively easy. It is possible to not enable USB until the OS drivers take over and have a very well-functioning system. If pre-OS support for EHCI or XHCI is required, then the tasks associated with the USB subsystem become substantially more complex. Legacy USB requires an SMI handler be used to trap port 60/64 accesses to I/O space and convert these to the proper keyboard or mouse commands. This pre-OS USB support is required if booting to USB is preferred.

SATA Initialization

A SATA controller supports the ATA/IDE programming interface as well as the Advanced Host Controller Interface (AHCI) (not available on all SKUs). In the following discussion, the term “ATA-IDE Mode” refers to the ATA/IDE programming interface that uses standard task file I/O registers or PCI IDE Bus Master I/O block registers. The term “AHCI Mode” refers to the AHCI programming interface that uses memory-mapped register/buffer space and a command-list-based model.

A separate document, RS – Intel® I/O Controller Hub 6 (ICH6) Serial ATA Advanced Host Controller Interface (SATA-AHCI) Hardware Programming Specification (HPS) contains details on SATA software configuration and considerations.

SATA Controller Initialization

The general guidelines for initializing the SATA controller during POST and S3 resume are described in the following sections. Upon resuming from S3, System BIOS is responsible for restoring all registers that it initialized during POST.

Setting the SATA Controller Mode

The system BIOS must program the SATA controller mode prior to beginning other initialization steps. The SATA controller mode is set by programming the SATA Mode Select (SMS) field of the Port Mapping register (D31:F2:Reg 90h[7:6]). The system BIOS may never change the SATA controller mode during runtime. Please note that the availability of the following modes is dependent on the SKU of PCH in use.

If system BIOS is enabling AHCI Mode or RAID Mode, system BIOS must disable D31:F5 by setting the SAD2 bit, RCBA + 3418h[25]. System BIOS must ensure that it has not enabled memory space, I/O space, or interrupts for this device prior to disabling the device.

IDE Mode

IDE mode is selected by programming the SMS field, D31:F2:Reg 90h[7:6] to 00. In this mode the SATA controller is set up to use the ATA/IDE programming interface. In this mode the 6/4 SATA ports are controlled by two SATA functions. One function routes up to four SATA ports, D31:F2, and the other routes up to two SATA ports, D31:F5 (Desktop SKUs only). In IDE mode, the Sub Class Code, D31:F2:Reg 0Ah and D31:F5:Reg 0Ah will be set to 01h. This mode may also be referred to as compatibility mode as it does not have any special OS driver requirements.

AHCI Mode

AHCI mode is selected by programming the SMS field, D31:F2:Reg 90h[7:6], to 01h. In this mode, the SATA controller is set up to use the AHCI programming interface. The six SATA ports are controlled by a single SATA function, D31:F2. In AHCI mode the Sub Class Code, D31:F2:Reg 0Ah, will be set to 06h. This mode does require specific OS driver support.

RAID Mode

RAID mode is selected by programming the SMS field, D31:F2:Reg 90h[7:6] to 10b. In this mode, the SATA controller is set up to use the AHCI programming interface. The 6/4 SATA ports are controlled by a single SATA function, D31:F2. In RAID mode, the Sub Class Code, D31:F2:Reg 0Ah, will be set to 04h. This mode does require specific OS driver support.

In order for the RAID option ROM to access all 6/4 SATA ports, the RAID option ROM enables and uses the AHCI programming interface by setting the AE bit, ABAR + 04h[31]. One consequence is that all register settings applicable to AHCI mode set by the BIOS have to be set in RAID as well. The other consequence is that the BIOS are required to provide AHCI support to ATAPI SATA devices, which the RAID option ROM does not handle.

PCH supports stable image-compatible ID. When the alternative ID enable, D31 :F2 :Reg 9Ch is not set, the PCH SATA controller will report Device ID as 2822h for a desktop SKU.

Enable Ports

It has been observed that some SATA drives will not start spin-up until the SATA port is enabled by the controller. In order to reduce drive detection time, and hence the total boot time, system BIOS should enable the SATA port early during POST (for example, immediately after memory initialization) by setting the Port x Enable (PxE) bits of the Port Control and Status register, D31:F2:Reg 92h and D31:F5:Reg 92h (refer **Error! Reference source not found.** step **Error! Reference source not found.** requirement), to initiate spin-up of such drive(s).

Memory Map

In addition to defining the caching behavior of different regions of memory for consumption by the OS, it is also firmware's responsibility to provide a "map" of the system memory to the OS so that it knows what regions are actually available for its consumption. The most widely used mechanism for a boot loader or an OS to determine the system memory map is to use real mode interrupt service 15h, function E8h, sub-function 20h (INT15/E820), which firmware must implement.

Region Types

There are several general types of memory regions that are described by this interface:
Memory (1) – General DRAM available for OS consumption.
Reserved (2) – DRAM address not for OS consumption.
ACPI Reclaim (3) – Memory that contains all ACPI tables to which firmware does not require runtime access. See the applicable ACPI specification for details.
ACPI NVS (4) – Memory that contains all ACPI tables to which firmware requires runtime access. See the applicable ACPI specification for details.
ROM (5) – Memory that decodes to nonvolatile storage (for example, flash).
IOAPIC (6) – Memory that is decoded by IOAPICs in the system (must also be uncached).
LAPIC (7) – Memory that is decoded by local APICs in the system (must also be uncached).

Region Locations

The following regions are typically reserved in a system memory map:
00000000-0009FFFF – Memory
000A0000-000FFFFF – Reserved
00100000-???????? – Memory (The ???????? indicates that the top of memory changes based on "reserved" items listed below and any other design-based reserved regions.)
TSEG – Reserved
Graphics Stolen Memory – Reserved
FEC00000-FEC01000* – IOAPIC
FEE00000-FEE01000* – LAPIC

Loading the operating system

Following the memory map configuration, a boot device is selected from a prioritized list of potential bootable partitions. The "Load Image" command or Int 19h is used to call the OS loader, which loads OS.

References

[1] Advanced Configuration and Power Interface Specification
http://www.acpi.info/spec.htm

[2] Applicable Chipset Datasheets
http://www.intel.com/products/embedded/chipsets.htm?iid=embed_porta
l+hdprod_chipsets#s1=all&s2=Intel%AE%20QM57%20Express%20Chips et&s3=all

[3] ATA/ATAPI Command Set Specifications
http://www.t13.org/Documents/MinutesDefault.aspx?keyword=atapi

[4] Chipset BIOS Writers Guides. See your Intel account representative. If you don't have an Intel account representative click here to get help online (http://edc.intel.com/Get-Help/).

[5] Intel® 64 and IA-32 Architectures Software Developer's Manual
http://developer.intel.com/products/processor/manuals/index.htm

[6] JEDEC DRAM Specifications http://www.jedec.org/

[7] Memory Reference Code. See your Intel account representative. If you don't have an Intel account representative click here to get help online (http://edc.intel.com/Get-Help/).

[8] Multiprocessor Specification 1.4 http://www.intel.com/design/pentium/datashts/24201606.pdf

[9] PCI Express® Base Specification http://www.pcisig.com/specifications/pciexpress/base2/

[10] PCI Firmware Specification http://www.pcisig.com/specifications/conventional/pci_firmware/

[11] PCI Local Bus Specification http://www.pcisig.com/specifications/conventional/

[12] $PIR Specification http://www.microsoft.com/whdc/archive/pciirq.mspx

[13] SD Specifications Part 1 Physical Layer Simplified Specification
http://www.sdcard.org/developers/tech/sdcard/pls/

[14] SD Specifications Part 2A SD Host Controller Simplified Specification
http://www.sdcard.org/developers/tech/host_controller/simple_spec/

[15] SD Specifications Part E1 SDIO Simplified Specification
http://www.sdcard.org/developers/tech/sdio/sdio_spec/

[16] Serial ATA http://www.sata-io.org/

[17] Simple Firmware Interface Specification http://www.simplefirmware.org

[18] Universal Serial Bus Specification http://www.usb.org/developers/docs/

* Mr. Nikola Zlatanov spent over 20 years working in the Capital Semiconductor Equipment Industry. His work at Gasonics, Novellus, Lam and KLA-Tencor involved progressing electrical engineering and management roles in disruptive technologies. Nikola received his Undergraduate degree in Electrical Engineering and Computer Systems from Technical University, Sofia, Bulgaria and completed a Graduate Program in Engineering Management at Santa Clara University. He is currently consulting for Fortune 500 companies as well as Startup ventures in Silicon Valley, California.