Complete Intel x86 Boot Flow - Technical Deep Dive

Table of Contents

---

1. Pre-CPU Boot: Power-On to CSME Initialization

Power-On Sequence

When you press the power button or apply power to the board:

CSME (Converged Security Management Engine) Initialization happens FIRST, even before the main CPU:

- CSME is a separate i486-based processor embedded in the Platform Controller Hub (PCH)
- It has its own 128KB ROM and ~1.5MB SRAM
- The CSME ROM is hardware-protected with fuses burned by Intel during production
- CSME boots from its ROM at the reset vector in 16-bit real mode, then switches to protected mode
- CSME reads the IFWI (Integrated Firmware Image) flash descriptor to locate its firmware region in SPI flash
- CSME loads and runs its own firmware (customized Minix3 microkernel-based OS)

What CSME Does:

1. Initializes its own hardware (SRAM, page tables, IOMMU called "A-Unit")
2. Reads and enforces hardware security policies from fused values (Boot Guard Profile)
3. Verifies the authenticity of microcode and firmware components
4. Initializes certain PCH components (basic power management, some controllers)
5. Releases the main CPU from RESET state when ready

Connection to IFWI Image: The IFWI image in SPI flash contains multiple regions:

- **Flash Descriptor**: Map of all regions
- **ME Region**: CSME firmware
- **BIOS Region**: Contains ACM, microcode, bootloader stages (Stage 1A/1B/2), FSP binaries
- **GbE Region**: Gigabit Ethernet firmware (if applicable)
- **Platform Data**: OEM-specific data

The CSME reads this descriptor to know where each component lives in flash.

---

2. CPU Boot and Root of Trust Establishment

CPU Reset Vector and Initial State

After CSME releases the main CPU from reset, the CPU (specifically the Bootstrap Processor - BSP in multi-processor systems) starts:

Initial CPU State:

- CPU starts in 16-bit real mode
- Instruction Pointer (IP) = 0xFFF0, Code Segment (CS) = 0xF000
- Due to segmented addressing magic, this maps to physical address **0xFFFFFFF0** (16 bytes below 4GB)
- This is the **reset vector** location

What's at the Reset Vector: The reset vector is NOT in RAM - it's mapped to SPI flash (NOR flash is memory-mapped). At 0xFFFFFFF0, there's typically just a FAR JMP instruction (needs to fit in 16 bytes) that jumps to the actual reset vector code.

VTF (Volume Top File)

- **VTF** is a special file in the firmware image placed at the top of the flash address space
- It contains the actual reset vector code that must be at 0xFFFFFFF0
- In Slim Bootloader, VTF is identified by GUID `1BA0062E-C779-4582-8566-336AE8F78F09`
- VTF is part of Stage 1A firmware volume

Boot Guard and ACM (Root of Trust)

Microcode Loading FIRST: Before anything else, the CPU loads microcode updates:

- CPU reads the Firmware Interface Table (FIT) from flash
- FIT contains pointers to microcode patches
- CPU verifies microcode signature against Intel's hardcoded key
- CPU loads the latest applicable microcode patch
- This happens in protected mode with Cache-as-RAM, then CPU switches BACK to real mode for compatibility

Authenticated Code Module (ACM) Execution: If Boot Guard is enabled (modern Intel platforms):

1. After microcode loading, CPU microcode reads the ACM from the FIT
2. ACM signature is verified using Intel's public key (hardcoded in CPU)
3. ACM executes in Cache-as-RAM (CAR) mode - CPU cache acts as writeable memory
4. ACM is Intel-signed, trusted code that performs platform security initialization

ACM Responsibilities:

- Reads the OEM Public Key Hash from Field Programmable Fuses (FPF) on the motherboard
- Verifies the Boot Guard Key Manifest using this OEM key hash
- Key Manifest contains public keys for verifying firmware components
- ACM verifies the Initial Boot Block (IBB) - which includes Stage 1A
- ACM can operate in two modes:
  - **Verified Boot**: Halts if verification fails
  - **Measured Boot**: Records measurements in TPM (Trusted Platform Module), sets PCR0 to specific values, extends with hash of IBB

Root of Trust Chain:

```
CSME ROM (Intel-fused)
  → CSME Firmware (verified by CSME ROM)
  → Microcode (verified by CPU with Intel key)
  → ACM (verified by CPU with Intel key)
  → Stage 1A/IBB (verified by ACM with OEM key from FPF)
  → Stage 1B (verified by Stage 1A)
  → Stage 2 (verified by Stage 1B)
  → Payload (verified by Stage 2)
  → OS (verified by Payload/Secure Boot)
```

Where Security Keys Are Stored:

- **Intel CPU**: Hardcoded Intel public key for microcode and ACM verification
- **Field Programmable Fuses (FPF)**: OEM Public Key Hash (SHA-256), burned during manufacturing, permanent
- **ACM**: Contains logic to verify using keys, not the keys themselves
- **Key Manifest**: Public keys for verifying BIOS components (in flash, verified by ACM)
- **Boot Policy Manifest**: Defines which firmware components to verify (in flash)

ACM Address:

- ACM location is defined in the Firmware Interface Table (FIT)
- FIT is at a fixed location pointer at 0xFFFFFFC0 (40h below 4GB)
- The firmware build process (stitching) ensures ACM is placed correctly and referenced in FIT

Reset Vector Code Execution

After ACM completes (or if Boot Guard is disabled), control transfers to the reset vector code in VTF:

Reset Vector Code Tasks:

1. Jumps to Stage 1A entry point (first DWORD in Stage 1A firmware volume)
2. This is `_ModuleEntryPoint` defined in `SecEntry.nasm`

---

3. Stage 1A: Reset Vector to Temporary RAM

Real Mode to Protected Mode Transition

Why GDT is Needed: You're right that there's a bit in CR0 (bit 0, the PE bit) that switches from real mode to protected mode. BUT:

- In protected mode, segment registers (CS, DS, SS, ES, etc.) no longer hold segment addresses
- Instead, they hold **segment selectors** - indices into the Global Descriptor Table (GDT)
- The GDT defines segment base addresses, limits, and access rights
- Without a valid GDT, the CPU cannot resolve memory addresses in protected mode

GDT Setup:

```assembly
; 1. Create GDT with minimum 3 entries:
;    - Null descriptor (required, all zeros)
;    - Code segment descriptor (executable, readable)
;    - Data segment descriptor (readable, writable)

; 2. Load GDT:
lgdt [gdt_descriptor]  ; Load GDT register with GDT base and limit

; 3. Set PE bit in CR0:
mov eax, cr0
or eax, 0x1           ; Set Protection Enable bit
mov cr0, eax

; 4. Far jump to flush pipeline and load CS:
jmp CODE_SEG:protected_mode_entry

; 5. Load data segment registers:
protected_mode_entry:
mov ax, DATA_SEG
mov ds, ax
mov ss, ax
mov es, ax
```

The GDT provides the segment descriptors that define:

- **Base**: Where the segment starts in linear address space
- **Limit**: Segment size
- **Access Rights**: Privilege level (ring 0-3), executable/writable/readable
- **Flags**: 16/32-bit mode, granularity (byte vs 4KB pages)

For flat memory model (modern approach), code and data segments both start at base 0x0 with limit 0xFFFFF (4GB with 4KB granularity), effectively disabling segmentation.

FSP-T and Temporary RAM Initialization

Pre-Board Initialization: Before calling FSP-T, Stage 1A may:

- Enable debug UART (platform-specific, often through PCH GPIO configuration)
- This allows early debug output via serial port
- Typically involves writing to PCH GPIO registers to configure pins as UART TX/RX

Calling FSP-T:

```c
// Stage 1A prepares parameters
FSPT_UPD tempRamInitParams;
// Configure CAR region, microcode address, code cache region
tempRamInitParams.MicrocodeRegionBase = <microcode address>;
tempRamInitParams.MicrocodeRegionLength = <size>;
tempRamInitParams.CodeRegionBase = <code to cache>;
tempRamInitParams.CodeRegionLength = <code size>;

// Call FSP TempRamInit
EFI_STATUS status = TempRamInit(&tempRamInitParams);
```

What FSP-T (TempRamInit) Does:

1. **Sets up Cache-as-RAM (CAR)**:
   - Configures CPU cache to act as temporary writeable memory
   - Uses MTRRs (Memory Type Range Registers) to mark cache region as writeback
   - Disables cache eviction (CD/NW bits in CR0)
   - Typical CAR size: 256KB-512KB
2. **Returns CAR address range**:
   - ECX = Start of available temporary memory
   - EDX = End of available temporary memory + 1
   - FSP reserves some space at the end for its own use (PcdFspReservedBufferSize)
3. **Code Caching**:
   - Enables caching for the flash code region to speed up execution

CAR Memory Layout:

```
[CAR Start]
├── Bootloader Stack
├── Bootloader Data
├── FSP Global Data (stored at address in HPET register PcdGlobalDataPointerAddress)
└── [CAR End]
```

LdrGlobal Structure Creation

What is LdrGlobal: `LOADER_GLOBAL_DATA` is the central data structure that tracks bootloader state across all stages.

LdrGlobal Contents (added by each stage):

- **Stage 1A adds:**
  - Stack information (base, current SP)
  - Available temporary memory range (CAR)
  - Debug verbosity level
  - Performance timestamps
  - HOB list pointer (initially for FSP-T HOBs)

Why Store in IDTR:

- IDTR (Interrupt Descriptor Table Register) normally points to the IDT
- But interrupts are DISABLED during early boot
- IDTR is a convenient place to store a pointer that persists across function calls
- Format: `idtr.limit = size`, `idtr.base = address of LdrGlobal`
- Stage 1B retrieves it: `sidt [idtr_location]` to get LdrGlobal pointer

Stage 1A Verification of Stage 1B

Hash-Based Verification:

1. Stage 1A contains expected hash (SHA-256/SHA-384) of Stage 1B in a manifest
2. Stage 1A reads Stage 1B from flash
3. Stage 1A calculates hash of the read Stage 1B binary
4. Compares calculated hash with expected hash
5. **If Verified Boot mode**: Boot halts if mismatch
6. **If Measured Boot mode**: Hash is extended into TPM, boot continues

Where are Verification Results Stored:

- PCH registers can store trust chain measurements
- TPM (Trusted Platform Module) Platform Configuration Registers (PCRs) store measurements
- Boot policy decisions stored in CPU registers or memory (LdrGlobal)

Stage 1A to Stage 1B Transition

Data Passed from Stage 1A to Stage 1B:

```c
typedef struct {
  VOID  *LdrGlobal;      // Pointer to LOADER_GLOBAL_DATA
  VOID  *Stage1BBase;    // Where Stage 1B is loaded
  // ... other parameters
} STAGE1A_PARAM;
```

Stage 1B is loaded into CAR and CPU jumps to its entry point (`_ModuleEntryPoint`).

---

4. Stage 1B: Memory Initialization

Configuration Database

What is Config Data:

- Platform-specific configuration parameters stored in YAML files
- Compiled into binary Configuration Database (CfgData)
- Contains:
  - Board-specific settings (GPIO configurations, device enables/disables)
  - Memory configuration (SODIMM SPD data, training parameters)
  - Platform features (USB ports, SATA, PCIe lanes)
  - Power management settings

Early Platform Init: Before memory initialization:

- Initialize critical hardware: debug UART, EC (Embedded Controller), GPIOs
- Configure PCH straps and early silicon settings
- Prepare for memory training

Memory Initialization with FSP-M

Building UPD (Updatable Product Data):

```c
// Stage 1B creates FSPM_UPD from config data
FSPM_UPD *MemoryInitParams;

// Example parameters from config data:
MemoryInitParams->RankMask = 0x03;          // 2 DIMMs populated
MemoryInitParams->DqPinsInterleaved = TRUE;
MemoryInitParams->MemorySizeLimit = 8192;   // Max 8GB
MemoryInitParams->EccSupport = FALSE;
// ... many more DRAM timing, training parameters
```

Why Config Data is Needed: FSP-M is silicon-specific (works for Coffeelake, Alderlake, etc.), but:

- Doesn't know your specific board layout
- Doesn't know how many DIMMs you have
- Doesn't know PCB routing (affects signal timing)

Config data provides this board-specific information.

FSP-M Execution:

```c
EFI_STATUS status = FspMemoryInit(MemoryInitParams, &HobListPtr);
```

What FSP-M Does:

1. **Memory Training**:
   - Initializes memory controller in CPU/SOC
   - Discovers actual installed memory (reads SPD from DIMMs via SMBus/I2C)
   - Performs memory training: Write/Read leveling, DQ/DQS training
   - Configures memory timings for stability/performance
2. **Creates Memory Map**:
   - Determines usable vs reserved memory regions
   - Allocates space for firmware, SMM, graphics stolen memory
3. **Builds HOBs**:
   - **Memory Resource Descriptor HOBs**: Describe physical memory ranges
   - Example: "Physical memory from 0x0 to 0x3FFFFFFF is available"
   - Example: "Physical memory from 0x80000000 to 0x80FFFFFF is reserved for bootloader"

HOB Example (Memory from Config vs Actual):

- Config says: 2x 4GB DIMMs (total 8GB)
- FSP-M discovers: Only 1x 4GB DIMM works (other slot defective)
- HOB describes: 4GB actual working memory, not 8GB

FSP-M vs Memory Training:

- Memory training is PART of FSP-M
- FSP-M is the complete memory initialization framework
- Training is the process of finding optimal signal timing

Migrating from CAR to DRAM

After FSP-M Returns:

1. **Stage 1B updates LdrGlobal**:
    - Memory information (TOLUM - Top of Low Usable Memory, TOUUM - Top of Upper Usable Memory)
    - HOB list pointer from FSP-M
2. **Creates LdrGlobal in DRAM**:
    - Allocates space in the bootloader-reserved DRAM region
    - Copies LdrGlobal from CAR stack to DRAM
3. **Stack Migration**:
    - Sets up new stack in DRAM
    - Switches stack pointer (ESP) from CAR to DRAM
4. **Calls TempRamExit**:

```c
FspTempRamExit(); // Tears down CAR, returns cache to normal mode
```

5. **Loads Stage 2**:
    - Reads Stage 2 from flash
    - Verifies Stage 2 hash (same process as 1A→1B)
    - Decompresses Stage 2 (may be LZ4/LZMA compressed)
    - Jumps to Stage 2 entry point

Data Passed to Stage 2:

```c
typedef struct {
  VOID  *HobList;       // Pointer to HOB list
  VOID  *LdrGlobal;     // Pointer to LOADER_GLOBAL_DATA (now in DRAM)
  // ... other parameters
} STAGE2_PARAM;
```

---

5. Stage 2: Silicon and Platform Initialization

Unshadowing

What is Unshadowing: In legacy BIOS, "shadowing" meant copying firmware from slow flash to faster RAM. In modern systems, "unshadowing" might refer to:

- Moving from execute-in-place (XIP) mode to executing from DRAM
- Stage 2 is already loaded and decompressed in DRAM
- No longer executing directly from memory-mapped flash

Pre-Silicon Init

Platform-specific initialization before FSP-S:

- Additional GPIO configuration
- EC (Embedded Controller) initialization
- Early device power-up sequences

FSP-S and Platform Initialization

Saving MRC (Memory Reference Code) Boot Parameters:

- MRC is part of FSP-M
- After first boot with memory training, training results are saved
- Stored as Non-Volatile Storage (NVS) data in flash or CMOS
- On subsequent boots: Fast boot path using saved parameters
- HOB: `FSP_NON_VOLATILE_STORAGE_HOB` contains this data

Creating UPDs for FSP-S:

```c
FSPS_UPD *SiliconInitParams;

// CPU features
SiliconInitParams->EnableVirtualization = TRUE;
SiliconInitParams->EnableHyperThreading = TRUE;

// PCH configuration
SiliconInitParams->PchSataEnable = TRUE;
SiliconInitParams->PchUsb30Port[0] = TRUE;
SiliconInitParams->PchPciePortEnable[0] = TRUE;
// ... many more silicon features
```

Who Initialized PCH and When:

- **CSME** (before CPU boot): Basic PCH initialization, power sequencing
- **FSP-M**: Memory controller in PCH (if applicable), some early PCH setup
- **FSP-S**: Full PCH initialization:
  - USB controllers
  - SATA controllers
  - PCIe root ports
  - HDA (audio) codec
  - Interrupt routing (IOAPIC configuration)
  - LPC/eSPI configuration

PCH Pin Initialization:

- PCH handles GPIO muxing for peripherals
- Reduces CPU pin count - peripherals connect to PCH, PCH connects to CPU via DMI (Direct Media Interface)
- FSP-S configures these based on UPD parameters from config data

FSP-S HOBs: FSP-S builds additional HOBs:

- Graphics GOP (Graphics Output Protocol) information
- SMM reserved memory regions
- Silicon-specific data structures

Post-Silicon Board Init

- Final platform-specific initialization
- Initialize remaining devices not handled by FSP

Multi-Processor Initialization

What Happens:

- BSP (Bootstrap Processor) wakes up APs (Application Processors)
- APs load microcode patches
- APs initialize their local APICs (Advanced Programmable Interrupt Controllers)
- APs enter waiting state for OS

PCI Enumeration

What is PCI Enumeration: Discovery and configuration of all PCI/PCIe devices:

1. **Scan**: Walk PCI bus tree, discover devices (read Vendor ID/Device ID)
2. **Assign Resources**:
   - Memory-mapped I/O (MMIO) addresses
   - I/O port addresses
   - Interrupt lines
3. **Configure Devices**:
   - Write base address registers (BARs)
   - Enable memory/IO access
   - Set up bridges for proper routing

Example:

```
Bus 0:
├── Device 0: CPU (Host Bridge)
├── Device 2: GPU → Assign MMIO 0xC0000000-0xCFFFFFFF
└── Device 1C: PCIe Root Port
    └── Bus 1:
        └── Device 0: NVMe SSD → Assign MMIO 0xD0000000-0xD0003FFF
```

ACPI Tables

Types of ACPI Tables:

1. **RSDP (Root System Description Pointer)**:
   - Located in BIOS data area or EFI system table
   - Points to RSDT/XSDT
2. **XSDT (Extended System Description Table)**:
   - Points to all other ACPI tables
   - 64-bit version of RSDT
3. **FADT (Fixed ACPI Description Table)**:
   - Hardware register addresses (PM timer, power buttons)
   - Power management profiles
   - **Points to DSDT**
4. **DSDT (Differentiated System Description Table)**:
   - Contains AML (ACPI Machine Language) code
   - Describes devices, power methods, GPIO, thermal zones
   - Example: `_S3` object describes S3 suspend sequence
   - Example: `_PRT` describes PCI interrupt routing
5. **SSDT (Secondary System Description Table)**:
   - Additional AML code (multiple SSDTs possible)
   - CPU P-states/C-states, dynamic devices
6. **MADT (Multiple APIC Description Table)**:
   - Lists all processors and their APIC IDs
   - I/O APIC information
   - Interrupt source overrides
7. **MCFG (PCI Express Memory Mapped Configuration)**:
   - PCIe MMCONFIG base address
8. **HPET (High Precision Event Timer)**, **BGRT (Boot Graphics Resource Table)**, etc.

Who Builds ACPI Tables:

- **Static Tables** (compiled from ASL source):
  - DSDT, some SSDTs
  - Built during firmware build process using ACPI compiler (iasl)
  - Stored in firmware image
- **Dynamic Tables** (built at runtime):
  - MADT (needs actual CPU topology)
  - SRAT (System Resource Affinity Table - NUMA)
  - Some SSDTs (CPU power management)
  - Stage 2 or UEFI DXE builds these using HOB data

ACPI Table Dependencies:

- **FSP builds HOBs → Stage 2 uses HOBs to build ACPI tables**
- Example: FSP-M HOB contains memory map → ACPI E820 table
- Example: FSP-S HOB contains graphics info → ACPI BGRTtable
- HOBs are firmware's internal format
- ACPI tables are OS's interface format

Initialization Order:

1. FSP-M/S create HOBs (firmware-internal data)
2. PCI enumeration assigns resources
3. Stage 2 reads HOBs and enumeration results
4. Stage 2 builds ACPI tables (OS-visible data)

FSP-S Notify Phases

Why Notify FSP-S About PCIe Enumeration:

```c
// After PCI enumeration completes
NotifyPhase(EnumInitPhaseAfterPciEnumeration);
```

- FSP-S may need to perform silicon-specific configuration based on discovered devices
- Lock down certain registers
- Configure silicon features that depend on PCI configuration

Notify Phases:

1. **After PCI Enumeration**: Silicon config based on devices found
2. **Ready to Boot**: Final lockdown, security measures, prepare for OS

---

6. Payload and OS Loading

Platform ID Matching

Where Platform ID Matching Occurs:

- **Hardware Platform ID**: Stored in GPIO straps, EEPROM, or PCH registers
- **Software Platform ID**: Embedded in firmware image or config data

Matching Process:

```c
// Example in Stage 2 or payload
uint8_t hwPlatformId = ReadPlatformIdFromHardware();  // Read GPIOs or straps
uint8_t swPlatformId = GetFirmwarePlatformId();       // From config data

if (hwPlatformId != swPlatformId) {
    if (hwPlatformId == 0) {
        // Default/generic platform, use default config
        LoadDefaultConfiguration();
    } else {
        // Mismatch - may halt or use failsafe config
        ERROR("Platform ID mismatch!");
    }
}
```

Where DLT (Device List Table) Fits:

- Platform ID determines which device configuration to use
- Different platform IDs may enable/disable different devices
- Typically checked before major initialization, possibly in Stage 2

Payload Execution

What is Payload: Highly customizable final stage before OS. Common payloads:

1. **OS Loader** (like SBL's OS Loader):
    - Loads and boots kernel (bzImage/zImage for Linux)
    - Minimal, fast, optimized for specific OS
2. **UEFI Payload**:
    - Full UEFI environment
    - Supports Windows, multiple boot options
    - UEFI services for OS loader
3. **U-Boot**:
    - Flexible boot loader, network boot, scripts
4. **Pre-OS Binary**:
    - Manufacturing/diagnostics tools
    - Firmware update tools
    - Example: Memory test utilities

HOB List Passing:

Why Pass HOBs When LdrGlobal Has Them:

- **LdrGlobal** is bootloader-internal structure
- **HOBs** are standard firmware interface (UEFI PI spec)
- Payload may not understand LdrGlobal format
- HOBs provide standardized way to:
    - Describe memory map
    - Pass performance data
    - Provide graphics framebuffer info
    - Pass ACPI table locations

How HOBs Are Passed:

```c
// Stage 2 to Payload (32-bit):
void PayloadEntry(void *HobList, void *PayloadBase) {
   // HobList pointer in EDX or as parameter
}

// 64-bit payload:
void PayloadEntry(void *HobList) {
   // HobList pointer in RCX register
}
```

HOBs are a linked list in memory:

```
[HOB Header] → [Next HOB] → [Next HOB] → [End HOB]
Each HOB has:
- Type (Resource, Memory Allocation, etc.)
- Length
- Data specific to type
```

Kernel Loading

OS Loader Payload Process:

1. **Parse HOBs**: Get memory map, reserved regions
2. **Locate Kernel**: Read from disk/network/filesystem
3. **Verify Kernel**: Check signature (if secure boot)
4. **Load Kernel**: Copy to appropriate memory location
5. **Setup Boot Parameters**:
    - For Linux: Create boot_params structure, command line
    - Pass E820 memory map (from HOBs)
6. **Transfer Control**: Jump to kernel entry point

FSP-S Notification:

```c
// Before booting OS
NotifyPhase(EnumInitPhaseReadyToBoot);
```

- FSP-S performs final lockdown
- May lock SMM memory
- Disable firmware interfaces

FSP-S Memory Cleanup:

- **Ready to Boot Phase**: Locks down temporary FSP-used memory
- **End of Firmware Phase**: Clears sensitive data from temporary regions

Kernel and ACPI/Device Tree

Windows Kernel:

- Requires ACPI tables
- Kernel's ACPI driver parses DSDT/SSDTs to discover devices
- ACPI _HID/_CID methods identify devices
- No device tree - ACPI only

Linux Kernel:

- **x86**: Uses ACPI (same as Windows)
- **ARM/Embedded**: Can use Device Tree (DTB - Device Tree Blob)
- **How Linux Gets ACPI**: OS loader passes ACPI RSDP pointer in boot parameters

```c
boot_params.acpi_rsdp_addr = <RSDP address from HOB>;
```

How OS Gets ACPI Tables:

1. Firmware builds ACPI tables in memory
2. RSDP location is passed via:
    - EFI System Table (UEFI)
    - Boot parameters (Linux)
    - HOB (firmware interface)
3. OS parses RSDP → XSDT → All other tables
4. OS builds device tree from DSDT/SSDTs

Linux RootFS:

- RootFS location passed via kernel command line: `root=/dev/sda1`
- Not from ACPI - ACPI only describes hardware
- Bootloader (OS Loader payload) constructs kernel command line

---

7. Key Data Structures and Their Relationships

LdrGlobal vs HOBs

LdrGlobal:

- Bootloader-private data structure
- Lives throughout boot process (Stage 1A → Stage 2)
- Contains:
  - Stack info
  - Memory ranges
  - HOB pointers
  - Performance timestamps
  - Config data pointers
  - Internal state
- **NOT passed to OS**
- Stored in IDTR during early boot, then in bootloader-reserved memory

HOBs:

- Standard interface (UEFI PI specification)
- Created by FSP, passed between firmware stages
- Accessible to payload and OS
- Contains:
  - Memory map
  - Resource allocations
  - Hardware information
  - ACPI table locations
  - Graphics framebuffer

Why Separate:

- HOBs are the "public API" for platform information
- LdrGlobal is "internal state"
- Payload needs HOBs (standard), not LdrGlobal (proprietary)

HOBs vs ACPI Tables

Dependency: ACPI tables are BUILT FROM HOBs (among other sources):

```
FSP HOBs → Stage 2/UEFI → ACPI Tables → OS
```

HOBs:

- Firmware-internal format
- Used during boot process
- Describes discovered hardware
- Not directly visible to OS

ACPI Tables:

- OS-visible format
- Used during OS runtime
- Describes hardware in ACPI-specific way (AML bytecode)
- Includes methods (_STA, _ON, _OFF) for runtime control

Example Flow:

1. FSP-M creates Memory Resource HOB: "0x0-0x7FFFFFFF = System RAM"
2. Stage 2 reads HOB
3. Stage 2 builds ACPI E820 memory map table
4. OS reads ACPI E820 table
5. OS uses E820 to manage memory

OS and LdrGlobal

How OS Knows LdrGlobal Address: It doesn't! And it doesn't need to.

- **For S3 Resume**: Firmware needs LdrGlobal, stores it in bootloader-reserved memory
- **OS**: Uses ACPI tables, not LdrGlobal
- **Bootloader-Reserved Memory**: Marked in E820/ACPI as "Reserved" - OS won't touch it

Memory Map Visibility:

```
Firmware View:
├── LdrGlobal data
├── HOBs
├── FSP reserved regions
├── SMM regions

OS View (from ACPI E820):
├── Usable RAM (0x0-0x7FFFFFFF)
├── Reserved (bootloader/firmware regions)
├── ACPI Tables (reclaimable after parsing)
├── ACPI NVS (for S3 resume)
```

8. S3 Resume Path

What is S3 Resume

ACPI S3 State:

- **S3 = Suspend to RAM**
- RAM remains powered (in self-refresh mode)
- CPU, chipset, most devices powered off
- Resume is much faster than cold boot

S3 Resume Flow:

1. **Wake Event**: Power button, lid open, timer, USB, network packet
2. **CSME**: Detects S3 resume (reads sleep type from RTC or PCH registers)
3. **CPU Boot**: Starts at reset vector (same as cold boot)
4. **Boot Guard**: May skip verification or fast path
5. **Stage 1A**: Detects S3 resume from sleep type register
6. **Memory Preserved**: RAM content intact, no memory training
7. **FSP-M Skip**: S3 path skips most of FSP-M (memory already initialized)
8. **Restore Context**:
    - CPU restores state from "waking vector" in ACPI FACS table
    - OS kernel resumes execution where it left off

LdrGlobal in S3:

- LdrGlobal created during initial boot contains S3-specific data
- Sleep type, NVS data pointer
- S3 resume path may reconstruct minimal LdrGlobal or use stored copy
- Most S3 data stored in ACPI NVS (Non-Volatile Storage) memory region

Why LdrGlobal Needed for S3:

- Firmware needs to know this is S3 path (different initialization)
- Track performance/debug info for S3 resume
- Store pointers to S3 waking vector

---

9. Common Questions Answered

IFWI Image and CSME

How CSME Gets Its Firmware:

1. **Manufacturing**: IFWI image is programmed to SPI flash using a programmer (DediProg, etc.)
2. **Flash Layout Tool**: Intel's iFlash tool creates IFWI with proper flash descriptor
3. **Flash Descriptor**: Contains regions map:

```
  0x00000000-0x00000FFF: Descriptor
  0x00001000-0x003FFFFF: ME/CSME Region
  0x00400000-0x00FFFFFF: BIOS Region
```

4. **CSME Boot**: CSME ROM knows to read descriptor, then boots from ME region

CSME Runs Before CPU:

- CSME is independent processor (i486-compatible)
- Has its own ROM with boot code
- Accesses SPI flash via dedicated hardware interface
- Doesn't need CPU to be running

Root of Trust Summary

Establishment:

```
Hardware Fuses (Intel + OEM)
  → CSME ROM (Intel-fused)
  → CPU Microcode (Intel-signed)
  → ACM (Intel-signed)
  → Boot Manifests (OEM-signed)
  → Firmware Stages (Verified chain)
```

Root of Trust Anchors:

- **Intel Root**: CPU's hardcoded Intel public key
- **OEM Root**: FPF-burned OEM public key hash
- **TPM**: Hardware root for measurements

Configuration Database Detail

Config Data Structure:

```
Platform Config
├── GPIO Config (pin assignments)
├── Memory Config
│   ├── SPD data
│   ├── Training parameters
│   └── Timing overrides
├── Device Config
│   ├── USB port enables
│   ├── SATA port enables
│   └── PCIe lane assignments
├── Security Config
│   ├── Secure Boot enable
│   └── Verified/Measured boot
└── Power Config
    ├── Package power limits
    └── C-state settings
```

YAML to Binary:

- Build process uses Config Editor tool
- Compiles YAML → Binary CfgData blob
- Embedded in firmware image
- Loaded and parsed by Stage 1B

Complete Boot Timeline

```
Time    Event                          Location         Memory State
=====================================================================
T+0ms   Power on                       Hardware         No memory
T+5ms   CSME ROM starts                CSME             CSME SRAM only
T+50ms  CSME firmware loaded           CSME             CSME SRAM only
T+100ms CPU released from reset        CPU              No memory
T+101ms Reset vector code              Flash (XIP)      No memory
T+102ms Microcode load                 Flash            Cache only
T+105ms ACM execution                  Cache            Cache as RAM
T+110ms Stage 1A starts                Flash (XIP)      Cache as RAM
T+115ms FSP-T (TempRamInit)            Flash (XIP)      Cache as RAM (enabled)
T+120ms Stage 1B starts                CAR              Cache as RAM
T+125ms FSP-M (MemoryInit)             CAR              Cache + DRAM training
T+400ms DRAM initialized               CAR              CAR + DRAM
T+405ms Migration to DRAM              DRAM             DRAM only
T+410ms Stage 2 starts                 DRAM             DRAM only
T+450ms FSP-S (SiliconInit)            DRAM             DRAM only
T+550ms Payload starts                 DRAM             DRAM only
T+600ms Kernel loading                 DRAM             DRAM only
T+650ms Kernel executing               DRAM             DRAM only
```

Platform ID Details

Hardware Platform ID Sources:

1. **GPIO Straps**: PCH GPIO pins pulled high/low via resistors
2. **EEPROM**: I2C EEPROM on board with platform data
3. **PCH Scratch Registers**: Configured by CSME or earlier stage
4. **Embedded Controller (EC)**: EC firmware returns platform ID

Typical Usage:

```c
// Stage 2 platform detection
PlatformId hwId = GetGpioBasedPlatformId();  // Read GPIO straps
BoardConfig *config = GetBoardConfig(hwId);  // Lookup config

if (config == NULL) {
   // Unknown platform, use default
   hwId = 0;
   config = GetBoardConfig(0);
}
```

Default Platform (ID 0):

- Generic configuration
- Minimal device enables
- Safe settings that work on most boards
- Used when hardware ID doesn't match any known config

Complete Flow Summary

Pre-CPU Boot: CSME initializes from its ROM → Loads ME firmware from flash → Initializes PCH basics → Enforces Boot Guard policy → Releases CPU

CPU Early Boot: CPU reset vector (0xFFFFFFF0) → Load microcode → Execute ACM (if Boot Guard enabled) → ACM verifies OEM key → ACM measures/verifies IBB

Stage 1A (Temporary RAM Setup): Reset vector code jumps to Stage 1A → Switch to protected mode (GDT setup) → Enable debug UART → Call FSP-T to setup Cache-as-RAM → Create LdrGlobal structure → Store LdrGlobal pointer in IDTR → Verify Stage 1B hash → Load Stage 1B to CAR → Jump to Stage 1B

Stage 1B (Memory Initialization): Load Config Database from flash → Parse YAML-based configuration → Early platform init → Build FSPM_UPD from config data → Call FSP-M → FSP-M trains memory and creates memory HOBs → Update LdrGlobal with memory info → Allocate LdrGlobal in DRAM → Migrate stack from CAR to DRAM → Call TempRamExit to tear down CAR → Verify Stage 2 → Load and decompress Stage 2 → Jump to Stage 2

Stage 2 (Silicon and Platform Init): Pre-silicon init → Save MRC parameters for fast boot → Build FSPS_UPD → Call FSP-S → FSP-S initializes PCH (USB, SATA, PCIe, interrupts) → FSP-S creates additional HOBs → Post-silicon board init → Initialize additional processors (APs) → PCI enumeration (scan buses, assign resources) → Notify FSP-S about PCI enumeration → Build ACPI tables from HOBs and config → Check platform ID match → Verify payload → Prepare HOBs for payload → Load and jump to payload

Payload (OS Loader): Parse HOBs for memory map → Setup boot environment → Locate kernel image → Verify kernel signature → Load kernel to memory → Build boot parameters (ACPI RSDP pointer, E820 map, command line) → Notify FSP-S ReadyToBoot → Jump to kernel entry point

OS Kernel: Parse ACPI RSDP → Walk ACPI tables (XSDT → FADT → DSDT/SSDTs → MADT, etc.) → Initialize ACPI subsystem → Discover devices from ACPI → Setup memory management from E820 → Load drivers → Mount root filesystem → Start init process

---

Key Takeaways

1. **CSME starts before CPU** and is critical for platform security policy
2. **Root of Trust** is established through hardware fuses → Intel-signed code → OEM-signed firmware
3. **GDT is mandatory** for protected mode as it defines memory segmentation even in flat memory model
4. **LdrGlobal** is internal bootloader state, **HOBs** are the standard interface for hardware information
5. **HOBs are created by FSP**, then used by bootloader to **build ACPI tables** for the OS
6. **Config Database** provides board-specific parameters that FSP needs but doesn't have
7. **Platform ID matching** ensures firmware compatibility with hardware
8. **S3 resume** preserves RAM and skips most initialization for fast wake
9. **ACPI tables are the OS's view** of hardware, built from HOBs and config data
10. **OS never sees LdrGlobal** - it's in bootloader-reserved memory marked as reserved in E820