Driver Execution Environment (DXE) and SMM : Technical Overview

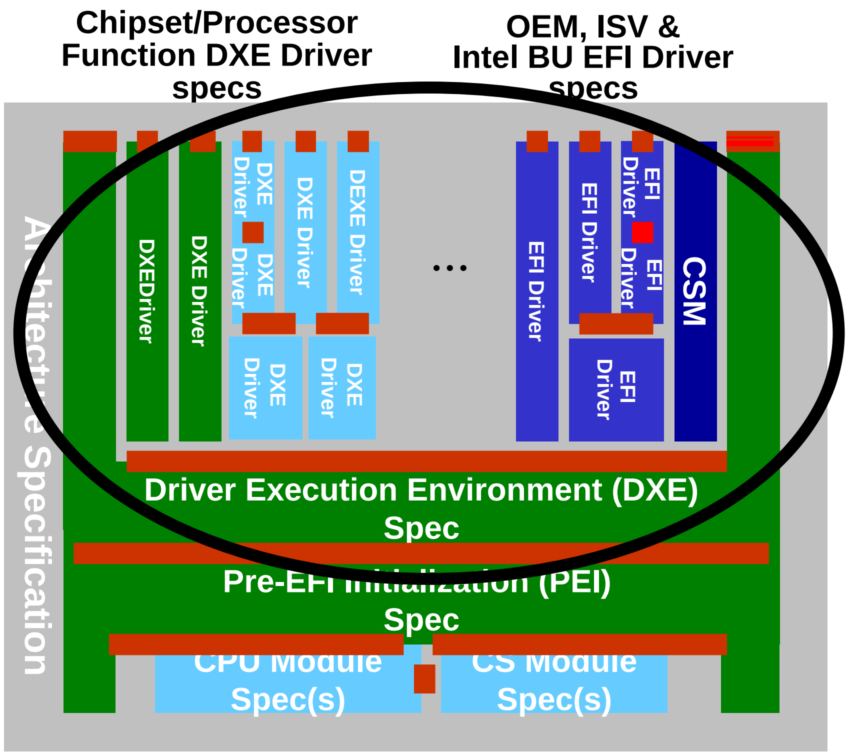
Intel Corporation
Software and
Services Group

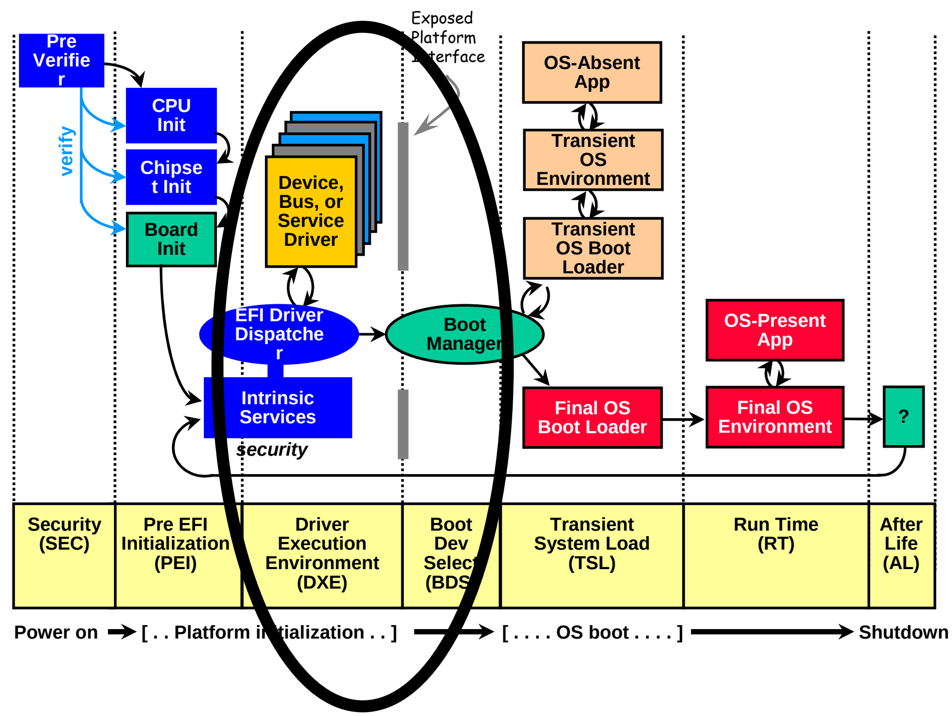
Agenda

- DXE Foundation
  - Overview
  - Data Structures
  - Events
  - Core
  - Architectural Protocols (API)
  - Dispatcher
  - Drivers
- SMM Overview

Platform Initialization (PI) Spec Vol 2 and Vol 4

DXE Foundation





Most features during DXE phase implemented as DXE drivers

See PI Specification Vol 2 DXE CIS

Why DXE

- No DXE
  - BIOS features coded in proprietary fashion
    - ODM has to port features from IBV to IBV
    - Third parties can not provide value added pre OS features
  - Single source file controls boot flow
- With DXE
  - Works like OS, large group of companies can write drivers
  - Modular systems enabled
    - FLASH on a plug in module to support module

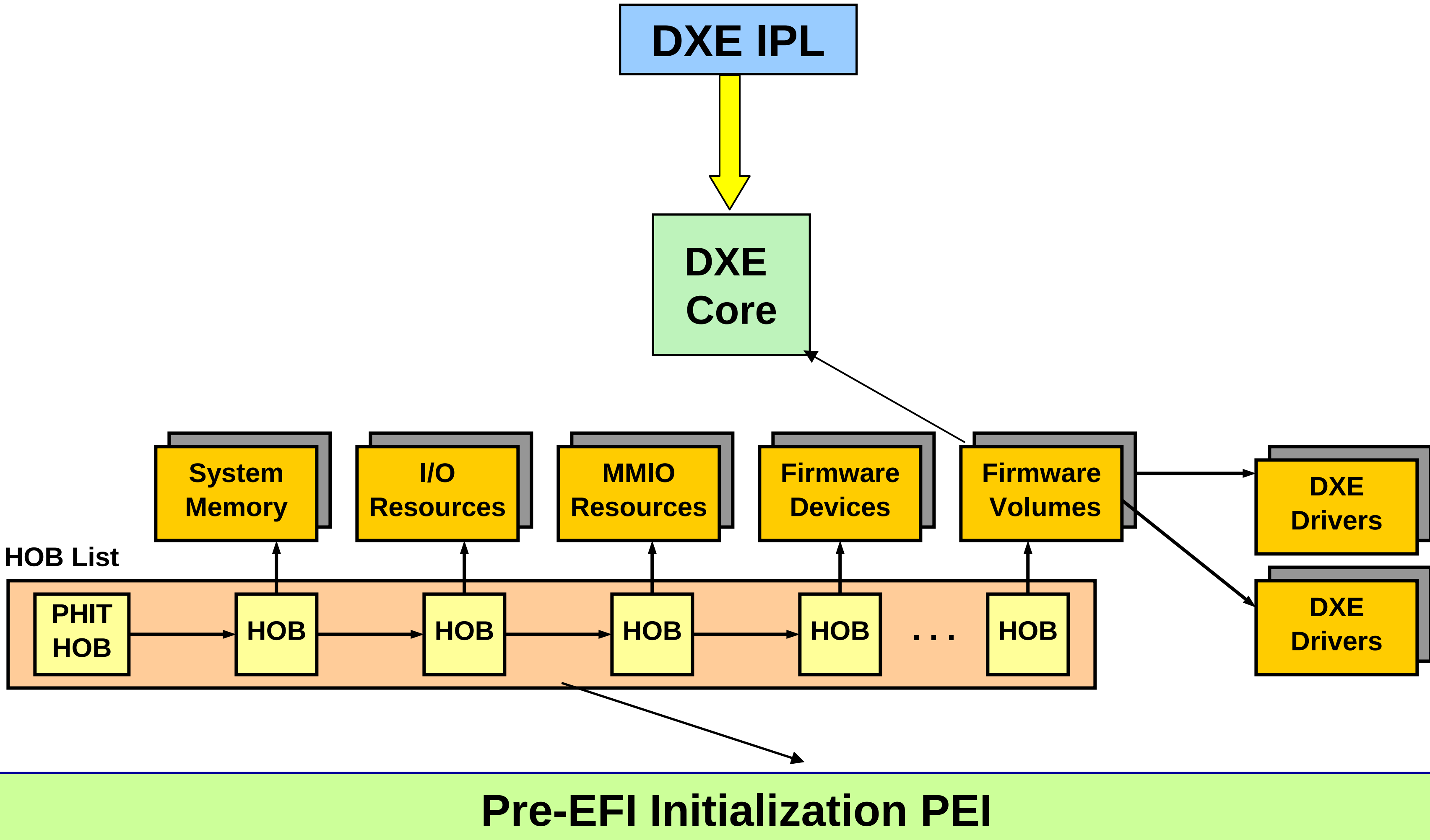
Properties of DXE Foundation

- Depends only on HOB list
  - State initialization passed in from PEI
- No hard coded addresses in DXE
  - Foundation code can be loaded anywhere
- No hardware specifics in DXE Foundation
  - Access to hardware abstracted by a set of architectural protocols (APs)
  - APs implemented as drivers
  - Only DXE Foundation may call APs
  - APs encapsulate CPU, chipset, board specifics

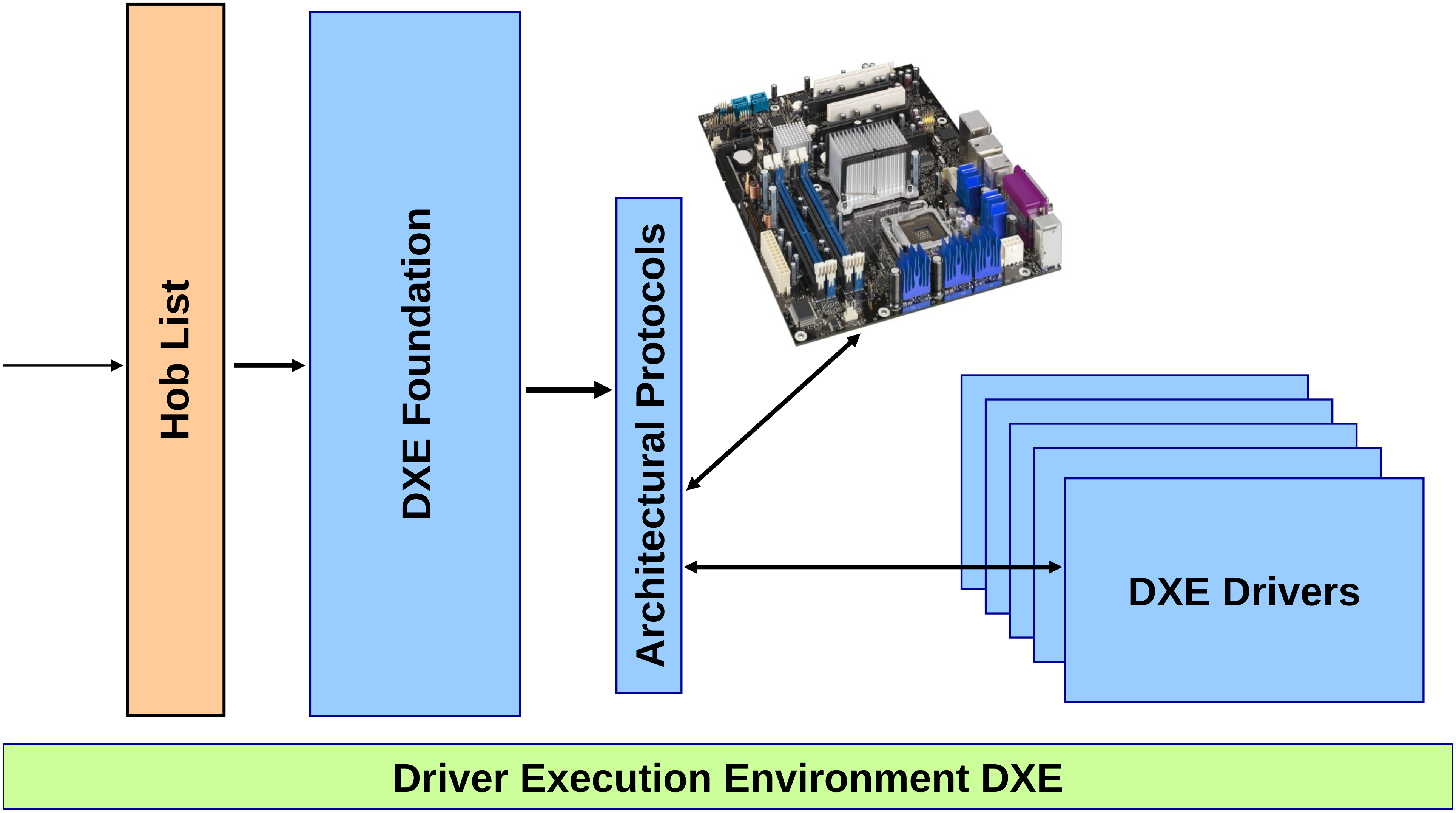
Introducing DXE Components

- DXE Core – The main DXE executable binary responsible for dispatching drivers and provide basic services.
- DXE driver – code loaded by the core to do various initializations, produce protocols and other services.
- DXE Dispatcher – The part of the DXE core that searches for and executes the drivers in the correct order.
- DXE Architectural Protocols – Produced by DXE drivers to abstract DXE from hardware.
- EFI System Table – Contains pointers to all the EFI service tables, configuration tables, handle database, and console device.

PEI to DXE Entry



PEI to DXE Entry



Source code

Where the PEI Transition Code is located

- Location in open source tree:
  - EDK I   \Sample\Universal\DxeIpl\Pei\DxeLoad.c
  - EDK II   \MdeModulePkg\Core\DxeIplPeim\DxeLoad.c
- **Call**: DxeLoadCore (inside the call DxeIpl->Entry())
  - EDK I - SwitchStacks Function call
  - EDK II - HandOffToDxeCore Function call

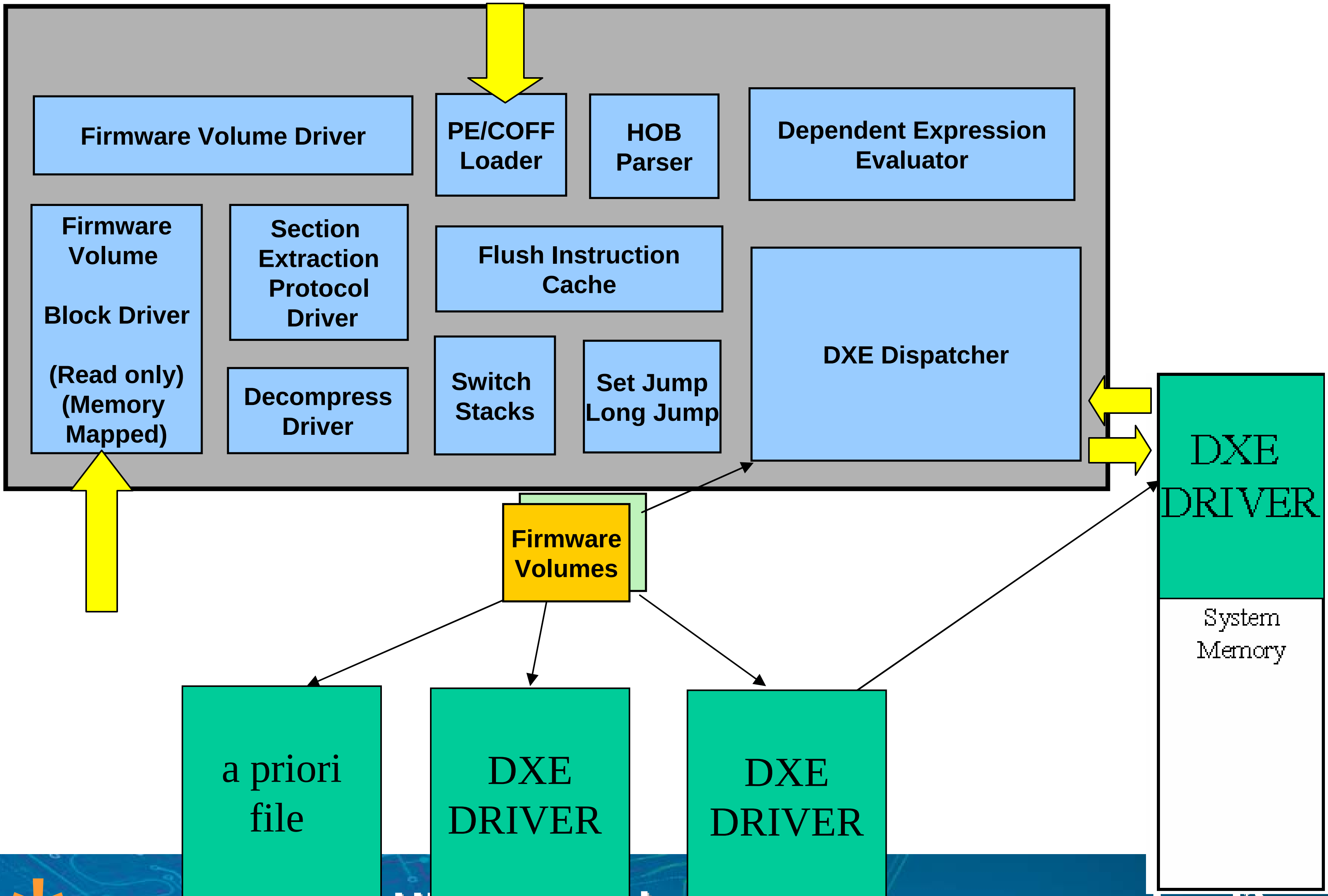
```
{ //  ------ EDK I   --------------------
 SwitchStacks (
   (VOID *) (UINTN) DxeCoreEntryPoint,
   (UINTN) (HobList.Raw),
   (VOID *) (UINTN) TopOfStack,
   (VOID *) (UINTN) BspStore
   );
}
```

```
{ //  ------ EDK II   --------------------
  // Transfer control to the DXE Core
  // The handoff state is simply a pointer to
  //     the HOB list

  HandOffToDxeCore (DxeCoreEntryPoint, HobList);

}
```

DXE Phase Flow



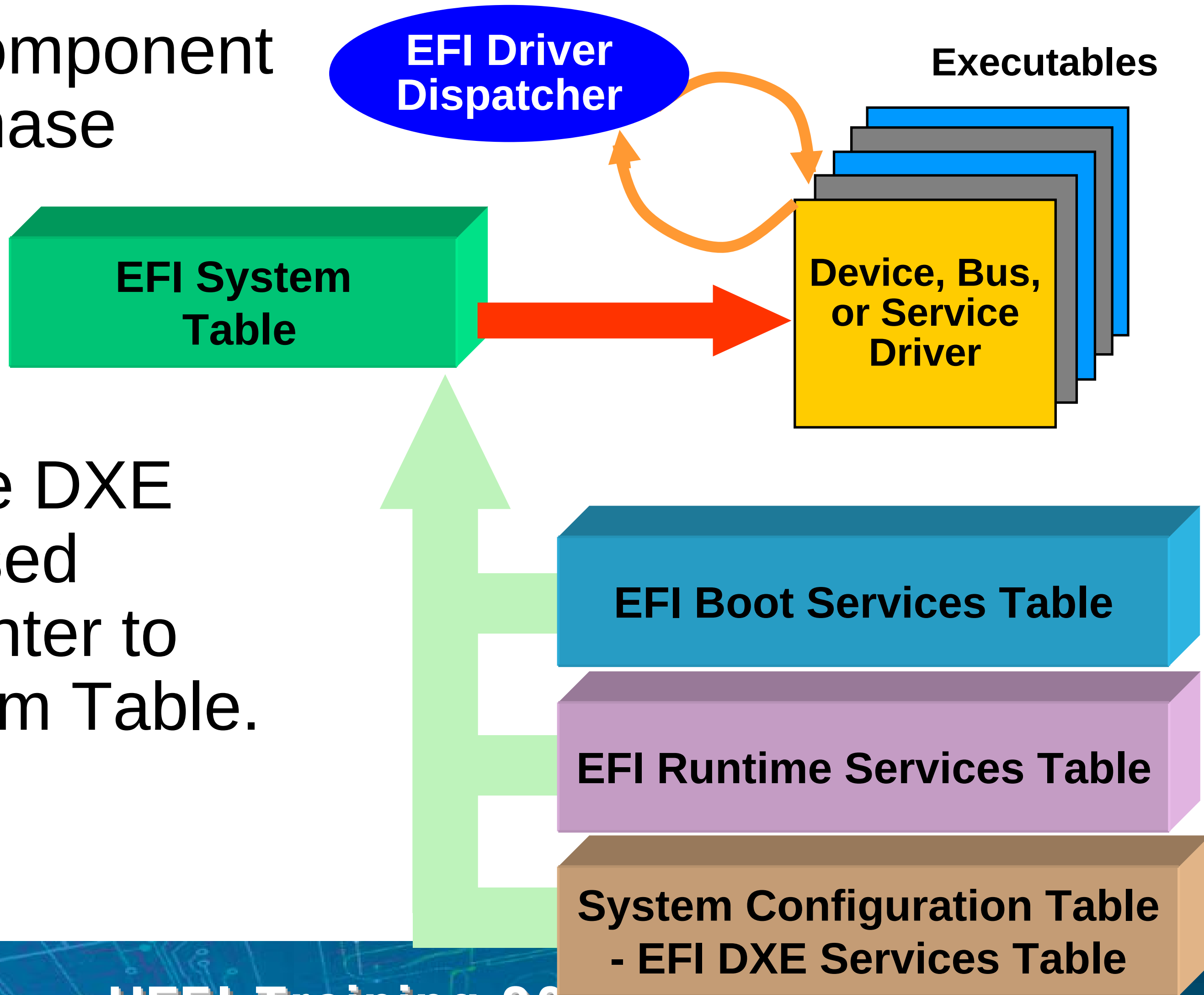
Agenda

- DXE Foundation
  - Overview
  - Data Structures
  - Events
  - Core
  - Architectural Protocols (API)
  - Dispatcher
  - Drivers
- SMM Overview

Platform Initialization (PI) Spec Vol 2 and Vol 4

EFI System Table

Passed to every executable component in the DXE phase



All services in the DXE phase accessed through a pointer to the EFI System Table.

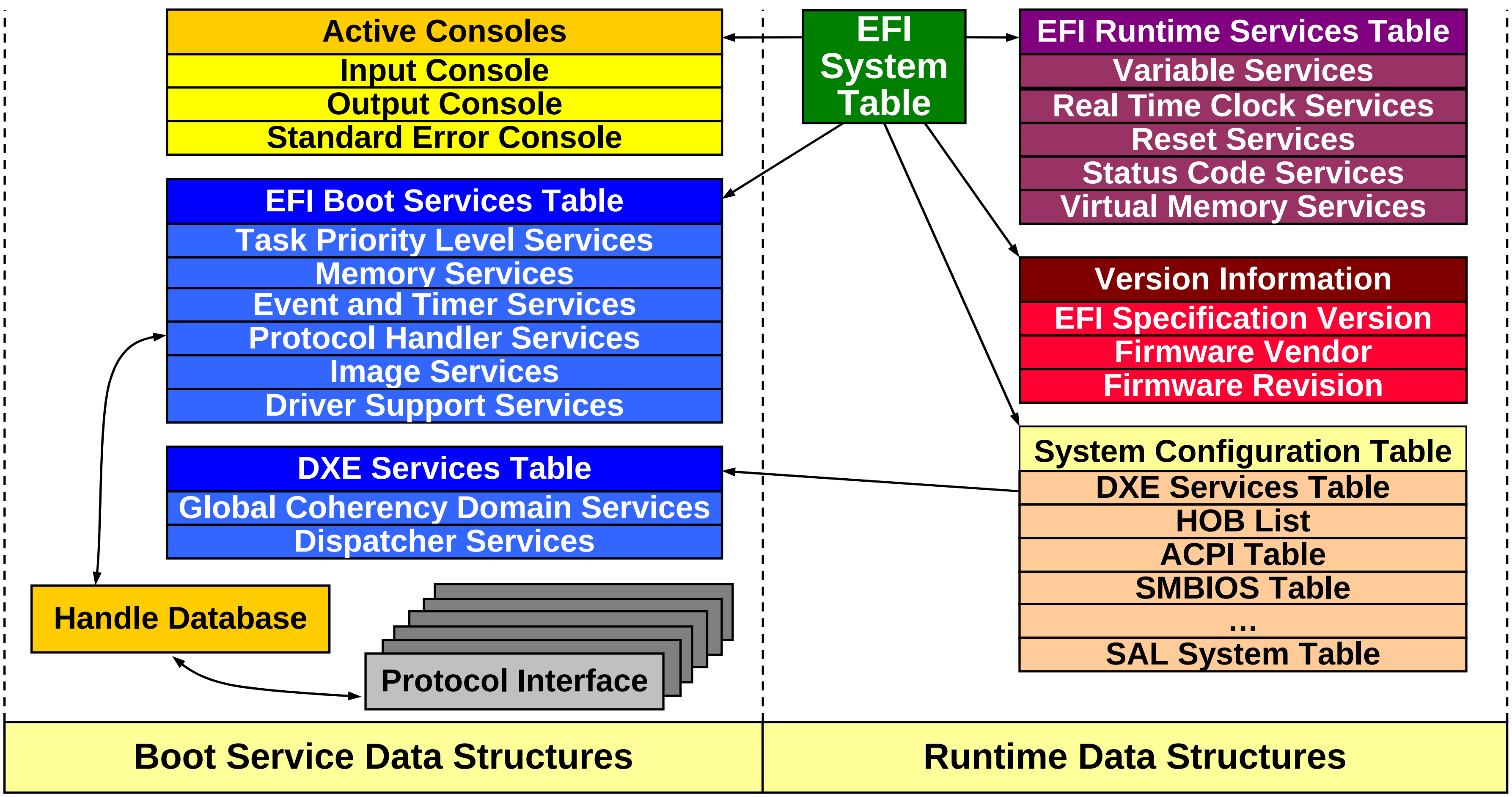
See § 4 UEFI 2.2 Spec.

System Boot Services?

- System services are interfaces that all UEFI compliant systems offer.
- Boot Services are a subset of these that are available only before ExitBootServices() is called.
- Runtime Services are the other subset and they are available both before and after ExitBootServices() is called.

See § 6 UEFI 2.2 Spec.

DXE Core Data Structures



Agenda

- DXE Foundation
  – Overview
  – Data Structures
  – Events
  – Core
  – Architectural Protocols (API)
  – Dispatcher
  – Drivers
- SMM Overview

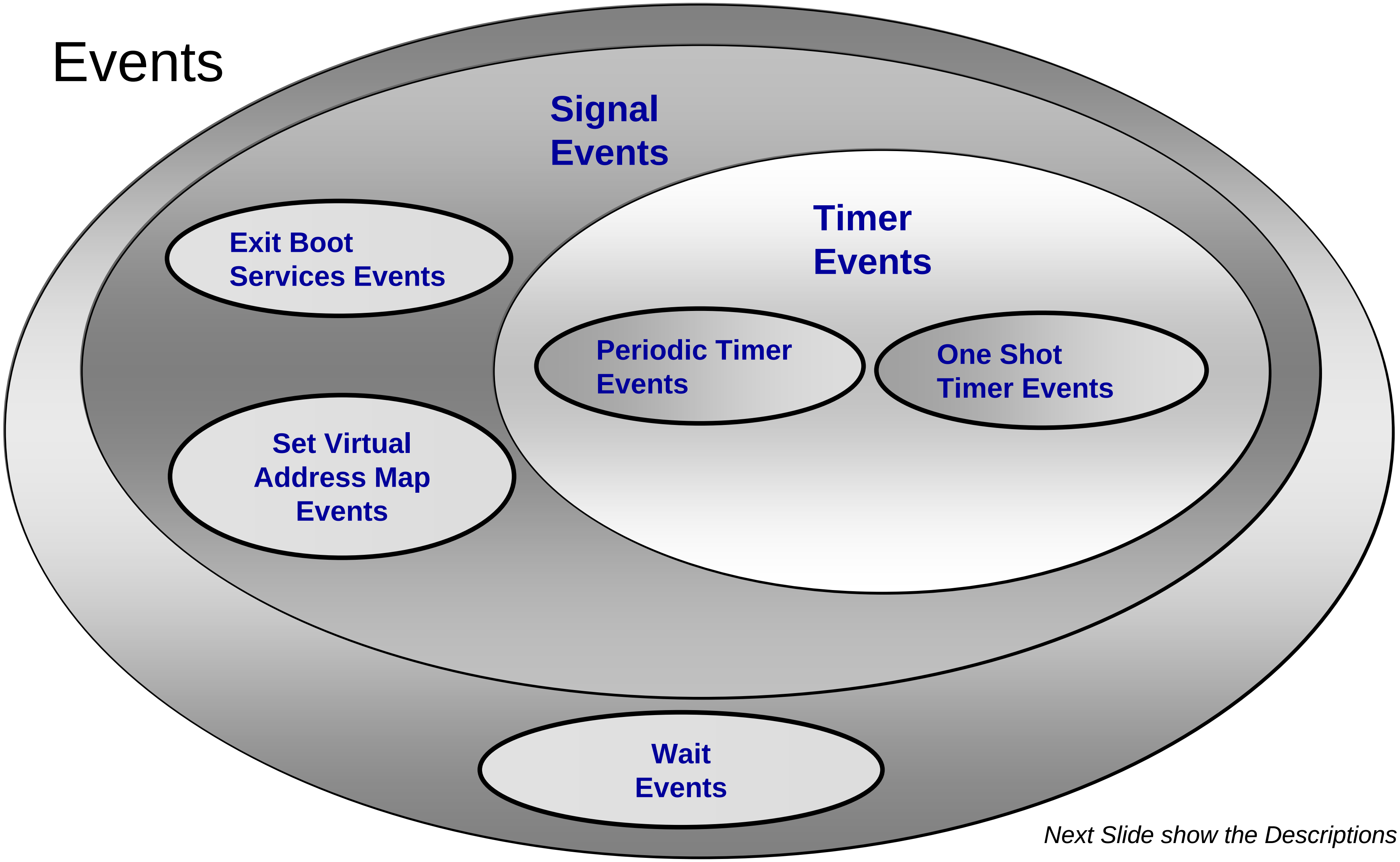
Platform Initialization (PI) Spec Vol 2 and Vol 4

Event Definition

- A part of Boot Services
- A messaging method
  - Returns control to a specific function
  - When the event is Signaled
    - After a specified time lapse
      - Useful for polling (i.e. Device Drivers)
    - When SignalEvent() is called with the event handle
      - Useful for controlling order of events
  - Note: special event for ExitBootServices()

See § 6.1 UEFI Spec 2.2 for Events as Boot Services

Event Types and Relationships



Next Slide show the Descriptions

Description of Event Types

Type of Events	Description
Wait event	notification function is executed whenever the event is checked or waited upon
Signal event	notification function is scheduled for execution when the event goes from the waiting state to the signaled state.
Exit Boot Services event	A special type of signal event that is moved from the waiting state to the signaled state when the EFI Boot Service **`ExitBootServices()`** is called.
Set Virtual Address Map event	special type of signal event that is moved from the waiting state to the signaled state when the EFI Runtime Service **`SetVirtualAddressMap()`** is called.
Timer event	type of signal event that is moved from the waiting state to the signaled state when at least a specified amount of time has elapsed.
Periodic timer event	type of timer event that is moved from the waiting state to the signaled state at a specified frequency.
One-shot timer event	type of timer event that is moved from the waiting state to the signaled state after the specified timer period has elapsed.

Description of previous Diagram

See backup for more info. on Events

Agenda

- DXE Foundation
  - Overview
  - Data Structures
  - Events
  - Core
  - Architectural Protocols (API)
  - Dispatcher
  - Drivers
- SMM Overview

Platform Initialization (PI) Spec Vol 2 and Vol 4

DXE Main

- Responsible for Initializing DXE Core
- Consumes HOB List
- Builds EFI System Table
- Builds EFI Boot Services Table
- Builds EFI Runtime Services Table
- Builds DXE Services Table
- Makes Memory-Only Boot Services Available
- Hands off control to the DXE Dispatcher
  - Requires access to Firmware Volumes
  - Requires LoadImage(), StartImage(), Exit()
  - May require decompression service

Source code

Where the DXE Main Code is located

- Location in open source tree:
  - EDK I   \Foundation\Core\Dxe\DxeMain\DxeMain.c
  - EDK II   \MdeModulePkg\Core\Dxe\DxeMain\DxeMain.c
- **Call**: DxeMain

```
VOID
EFIAPI
DxeMain (
  IN  VOID *HobStart  // Pointer to the beginning of the HOB List from PEI
  )
{
  // Initialize     ...        ...        ...        ...
  // Initialize Memory Services
  CoreInitializeMemoryServices (&HobStart, &MemoryBaseAddress, &MemoryLength);
   ...        ...        ...        ...
  // Allocate the EFI System Table and EFI Runtime Service Table from EfiRuntimeServicesData
  // Use the templates to initialize the contents of the EFI System Table and EFI Runtime Services Table
  //
  // Invoke the DXE Dispatcher
    CoreDispatcher ();
   ...        ...        ...        ...
  // Display Architectural protocols that were not loaded if this is DEBUG build
   DEBUG_CODE (     CoreDisplayMissingArchProtocols ();   )
  //
   gBds->Entry (gBds); // Transfer control to the BDS Architectural Protocol
  }
```

DXE Core Initialization

- Initialize EFI Boot Services Table
  - Global Variable in DXE Core
  - All services return EFI_NOT_AVAILABLE_YET
- Initialize DXE Services Table
  - Global Variable in DXE Core
  - All services return EFI_NOT_AVAILABLE_YET
- Initialize Memory-Only TPL[1] Services
  - RaiseTPL(), RestoreTPL()
- Initialize Memory Services (Parses HOB List)
  - AllocatePages(), AllocatePool(), FreePages(), FreePool()
- Allocate EFI System Table
  - Allocated from *EfiRuntimeServicesData*
- Allocate EFI Runtime Service Table
  - Allocated from *EfiRuntimeServicesData*
  - All services return EFI_NOT_AVAILABLE_YET
- Initialize GCD Services (Parses HOB List)
- Initialize Driver Support Services
- Initialize Event Services
- Initialize Event Services
- Initialize Protocol Services
- Initialize Miscellaneous Services
- Add HOB List to System Configuration Table
  - Part of EFI System Table
  - Provides other component access to HOB List
- Initialize Image Services
  - Creates Image Handle for DXE Core itself
- Create event for each Architectural Protocol
  - Informs DXE Core when AP is installed
  - Used to complete EFI Boot Services
  - Used to complete EFI Runtime Services
- Initialize Firmware Volume Drivers
  - Provides file access to FVs discovered by PEI
- Initialize File Decompression (Optional)
- **Hand control to DXE Dispatcher**

[1] Task Priority Level (TPL)

DXE Foundation Theory of Operation

- First goal: Initialize Platform
  - Initialize chipset and platform
- Loads drivers to construct environment that can support boot manager and OS boot
- Dependencies provide driver ordering
  - Grammar-based description of drivers' requirements
  - Including patch or override operations e.g. with "before/after" dependencies
- EFI drivers with no dependency started last
  - Compatibility for EFI 1.10 drivers, IHV cards etc.
- Dispatch completes as fast as practical
  - Required hardware init performed by driver on call to entry point
  - EFI driver entry points just register protocol
  - Defer initialization of boot devices until we know which are needed
- When all required drivers are loaded go to boot manager to attempt to boot

Dependency-based flow of control leads to more "just works" scenarios

DXE Control Flow

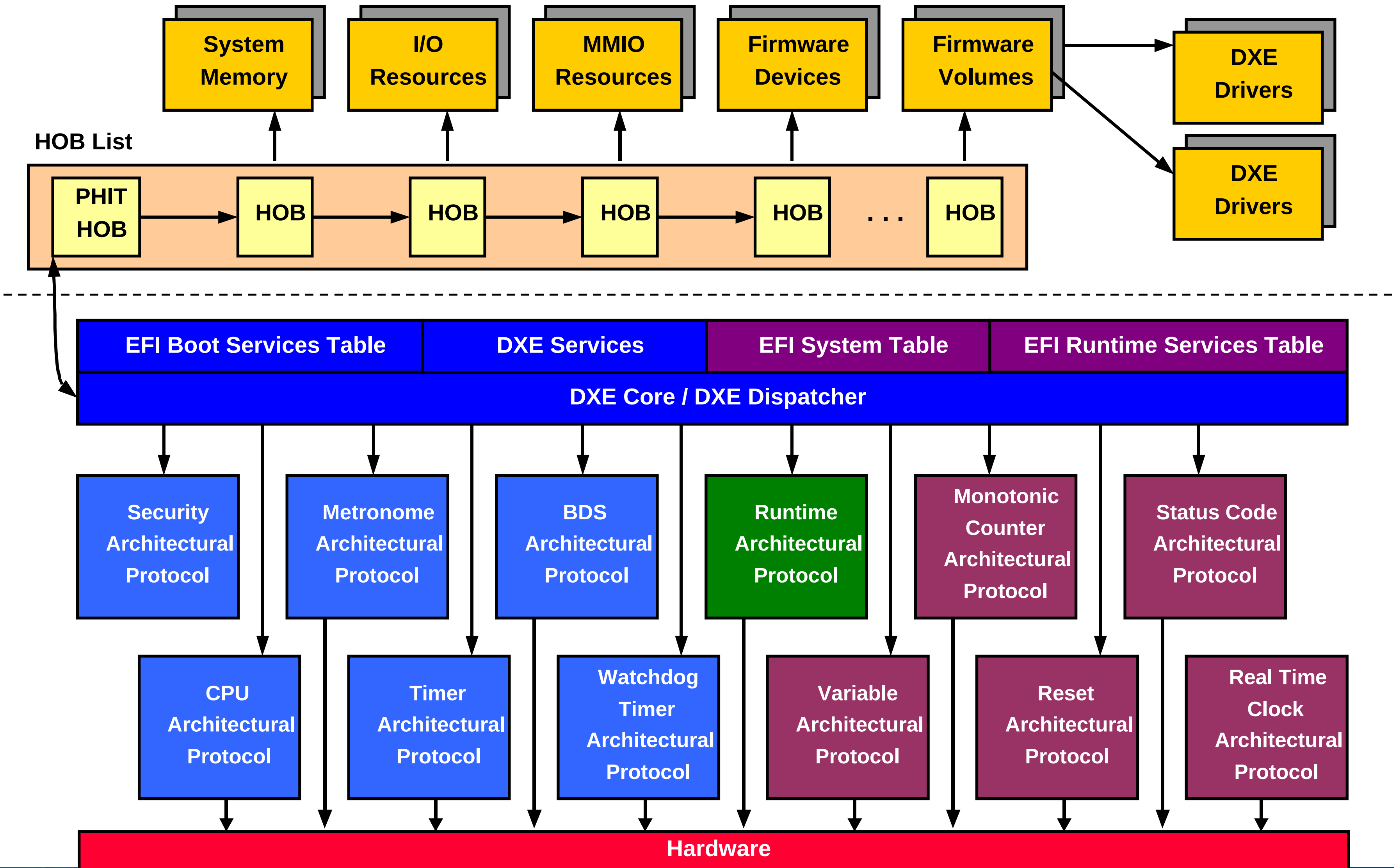
- Single threaded environment
- One software interrupt: Timer tick
  - Only means of asynchronous control transfer
- Implies devices are all polled
  - Timer tick allows event and callback when needed, e.g. servicing NIC for TCP/IP
- Avoids need to abstract interrupt controller
  - Typically CPU architecture specific
  - Hard to model with “good” s/w abstractions
- Experience says interrupts not needed
  - But timer flexibility required, e.g. power management, PPP stack serial port flow control
  - Use UEFI Events instead

Agenda

- DXE Foundation
  - Overview
  - Data Structures
  - Events
  - Core
  - Architectural Protocols (API)
  - Dispatcher
  - Drivers
- SMM Overview

Platform Initialization (PI) Spec Vol 2 and Vol 4

DXE Core Block Diagram



DXE – Architectural Protocols

- What are Architectural Protocols?
  - Typically functions that isolate platform specific hardware (e.g. real-time clock)
  - Provide support for boot services and runtime services
  - Low level protocols that support DXE APIs (e.g. boot and runtime services
  - Directly called by DXE core

See § 12 PI 1.1 Vol. 2 Spec

DXE – Architectural Protocols

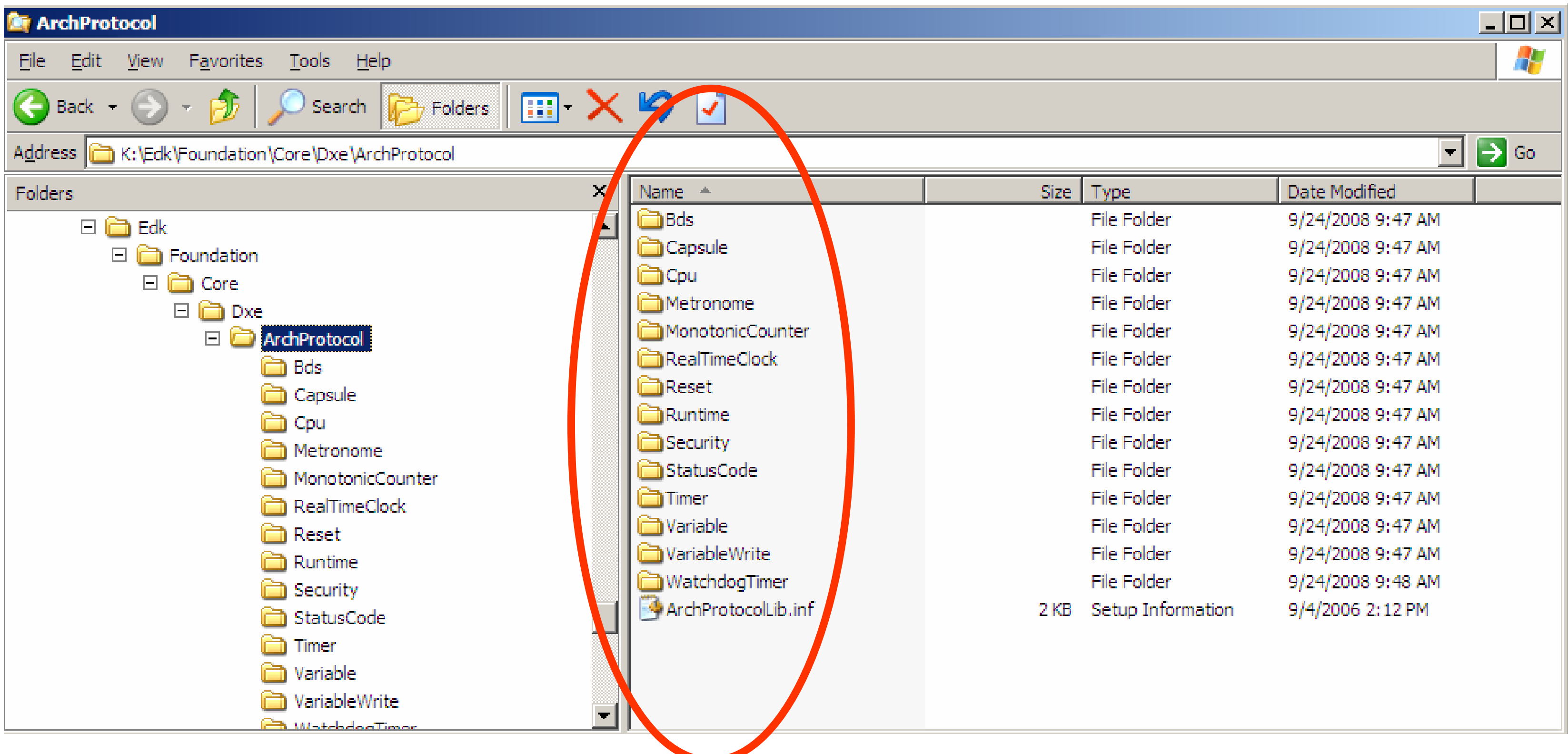
- Some APs have dependencies on others
  - Timer requires interrupts (CPU) and IO access
  - Watchdog timer requires timer and IO access
  - Reset requires CPU and IO access
- Dependencies can be satisfied by using one or more of the following methods to control load order
  - Dependency grammar to have DXE load in the correct order
  - RegisterProtocolNotify() to be notified when required AP gets loaded
  - Apriori list – file in the file system containing list of filename GUIDs

Source code

Where the DXE Architectural Protocols is located

Location in open source tree:

- EDK I \Foundation\Core\Dxe\ArchProtocol
- EDK II \EdkCompatibilityPkg\Foundation\Core\Dxe\ArchProtocol



Agenda

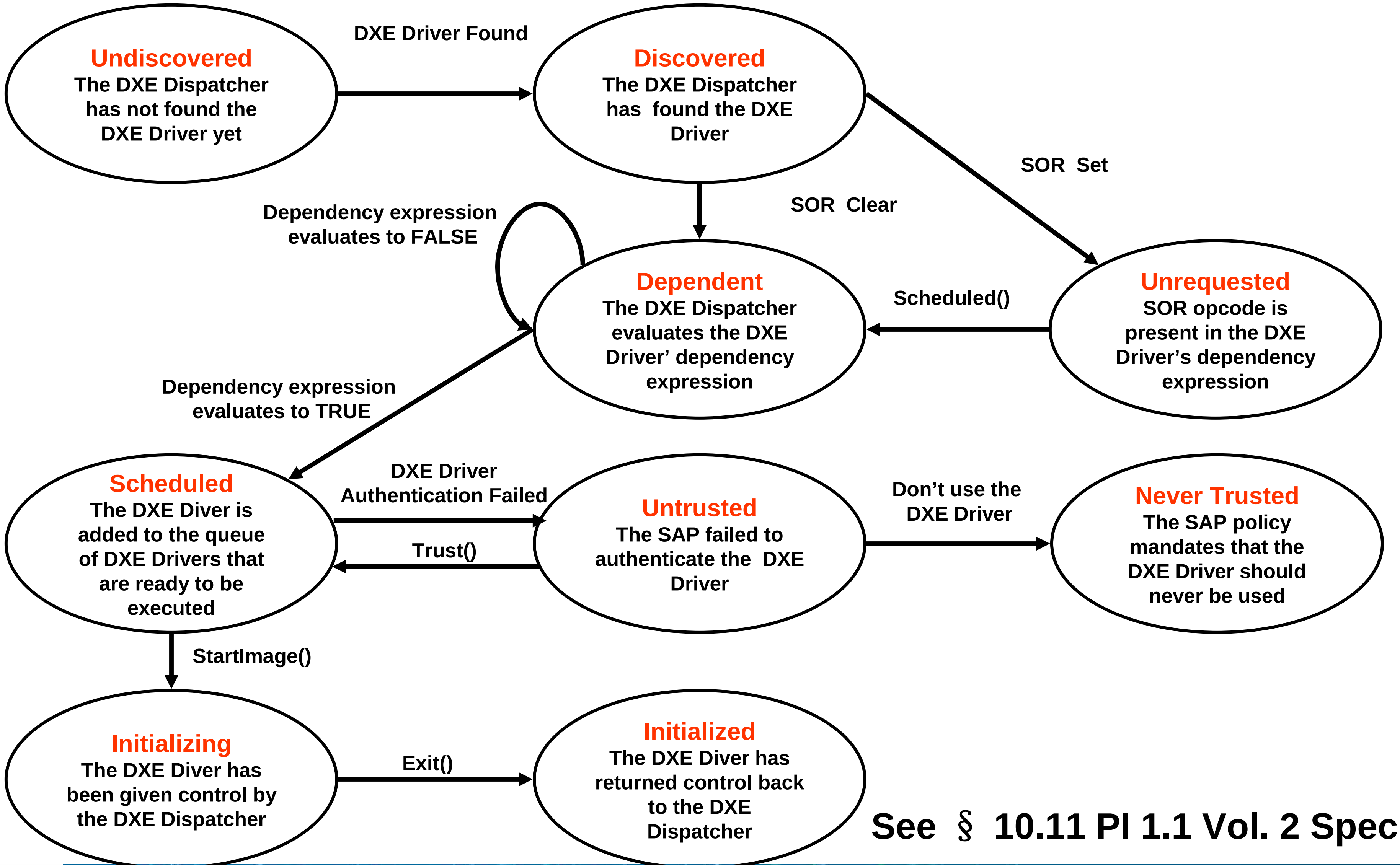
- DXE Foundation
  - Overview
  - Data Structures
  - Events
  - Core
  - Architectural Protocols (API)
  - Dispatcher
  - Drivers
- SMM Overview

Platform Initialization (PI) Spec Vol 2 and Vol 4

Dispatcher

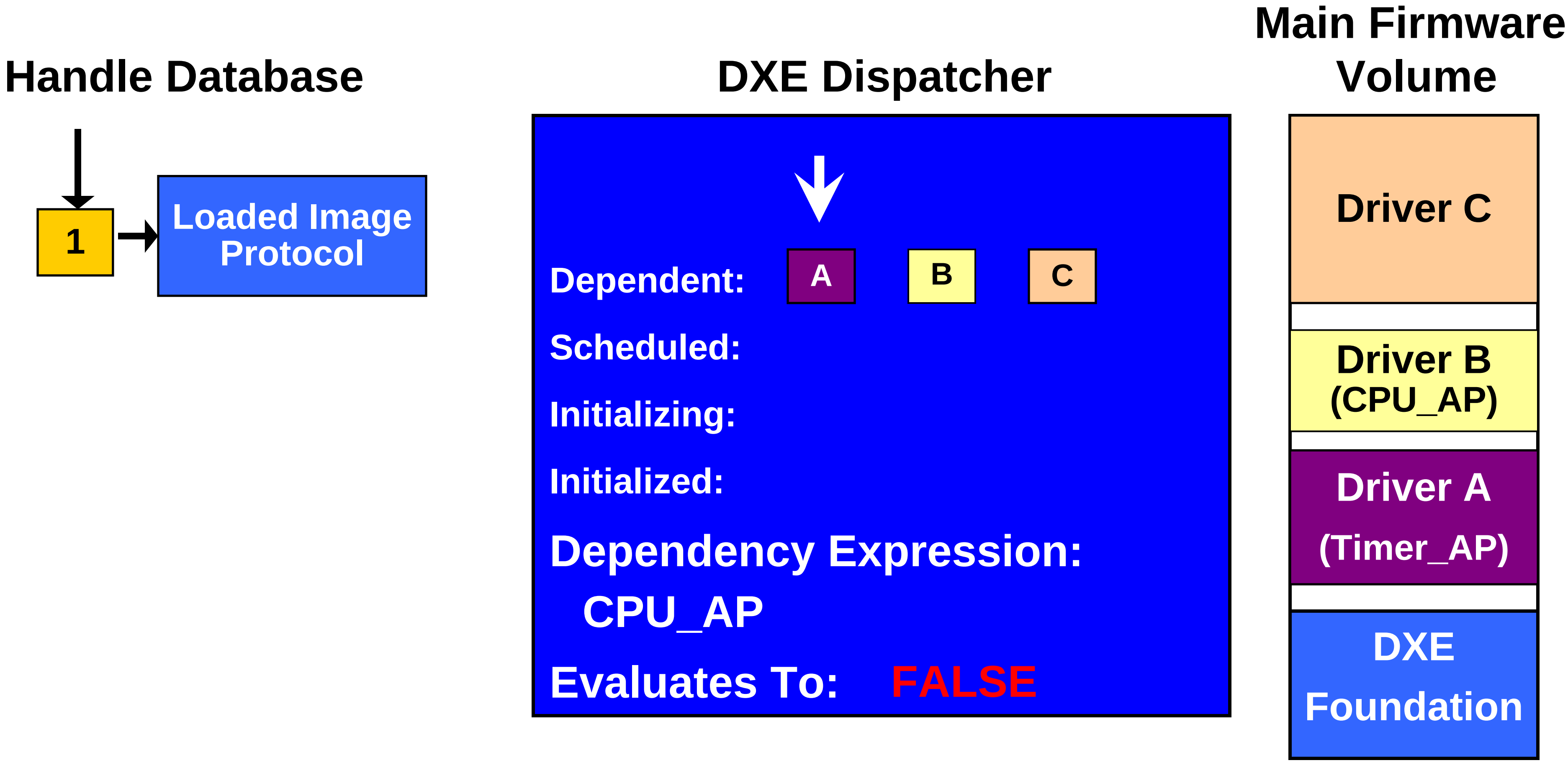
- Goals
  - Resolve Execution Ordering Among DXE Drivers
    - Drivers may be written by different organizations -Different divisions -Different companies
  - Support Known Business Issues
    - Emergency patches -“Control of Destiny” for examples: System developer - Add-in card developer - Driver developer
  - Expansion Hooks for e.g. Security
- Model
  - *A Priori* list of drivers to be run first
  - Requirements based dispatching for the rest
- Requirements
  - Protocols are the “Requirements” -Represented by their GUIDs
  - “List of Requirements”: Boolean Expressions / Before / After

DXE Dispatcher State Machine

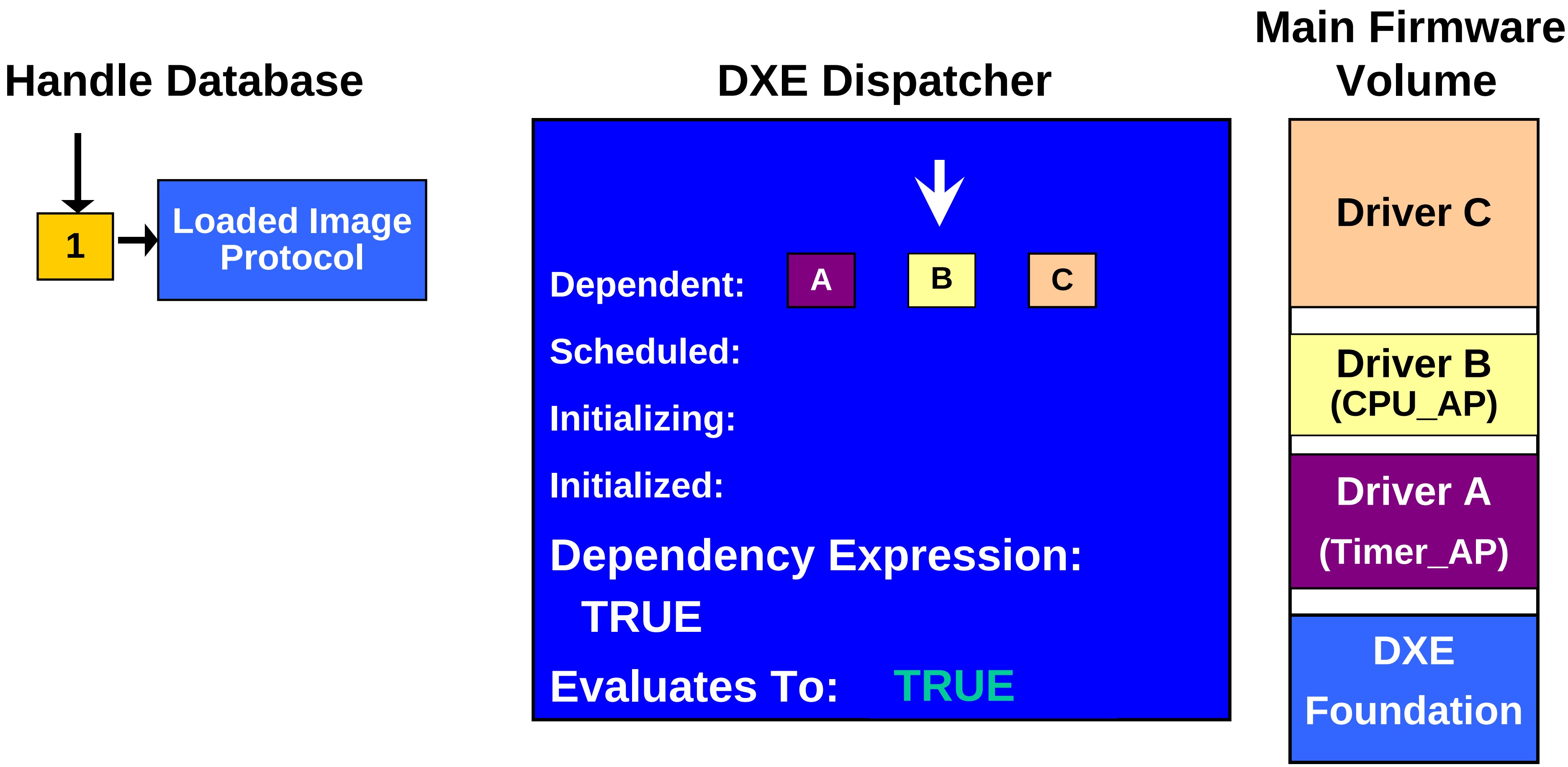


See § 10.11 PI 1.1 Vol. 2 Spec

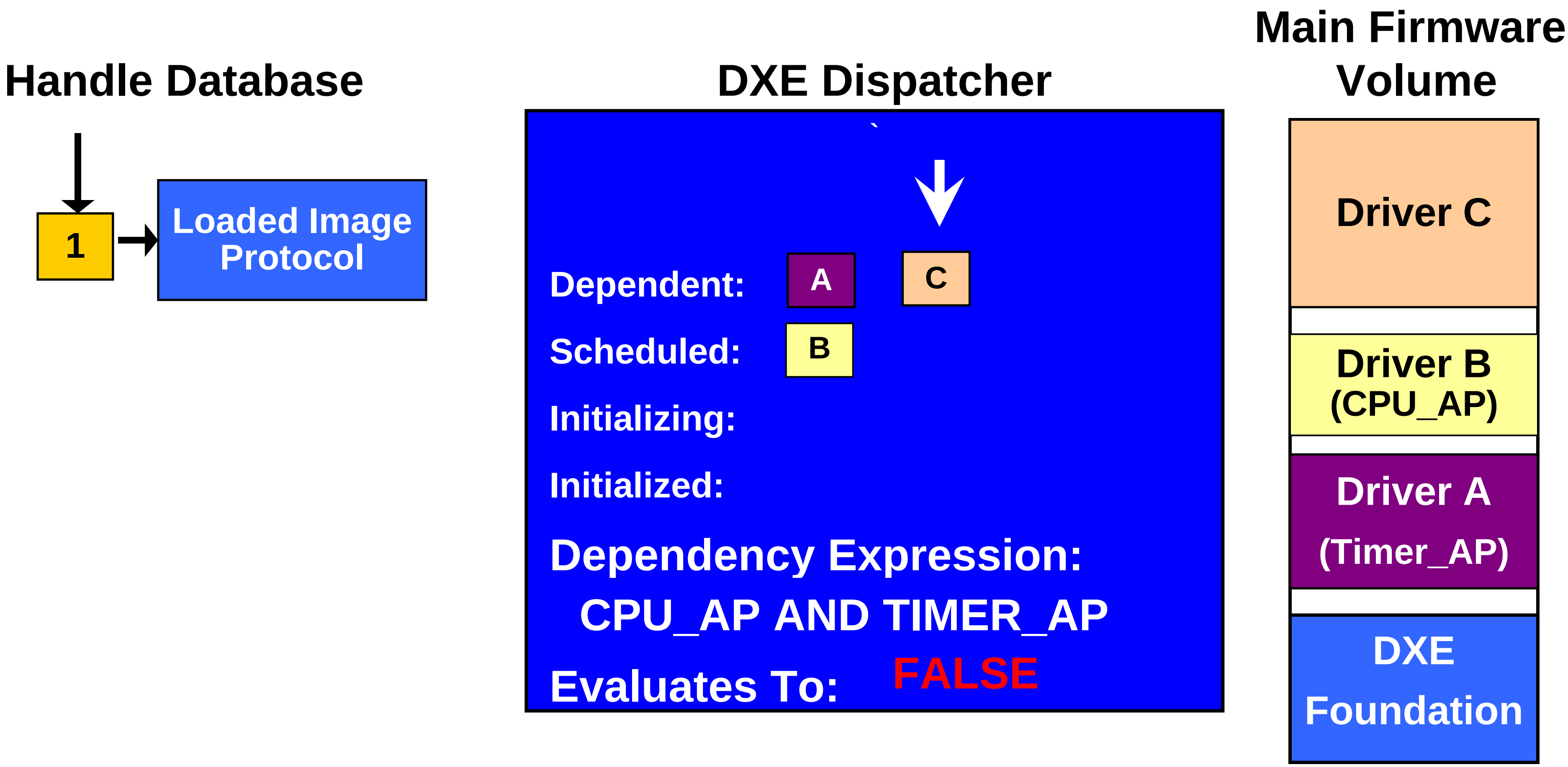
DXE Foundation Dispatcher



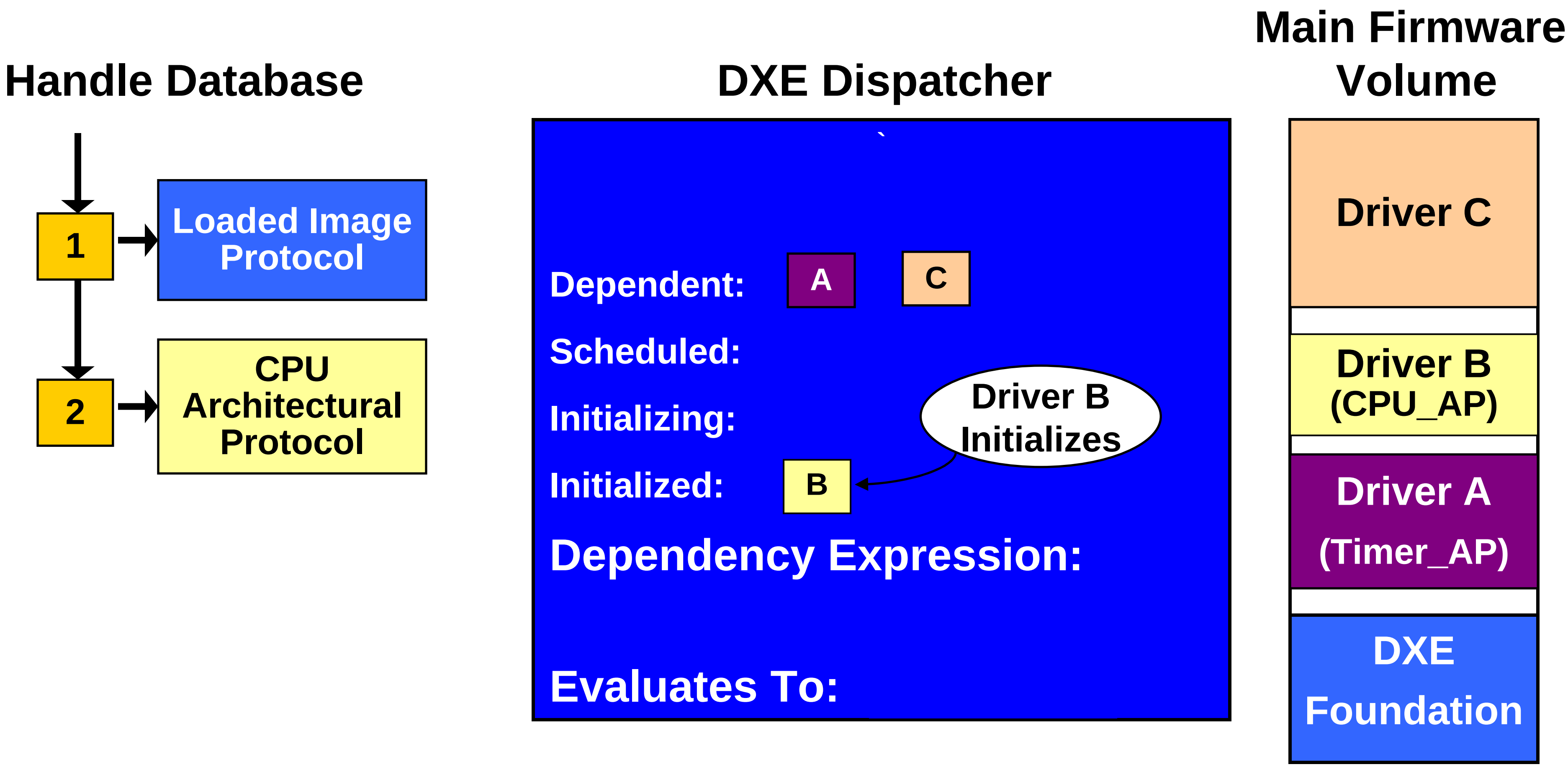
DXE Foundation Dispatcher



DXE Foundation Dispatcher



DXE Foundation Dispatcher



DXE Foundation Dispatcher

Handle Database

1 → Loaded Image Protocol

2 → CPU Architectural Protocol

DXE Dispatcher

Dependent: A C

Scheduled:

Initializing:

Initialized: B

Dependency Expression:

CPU_AP

Evaluates To: TRUE

Main Firmware Volume

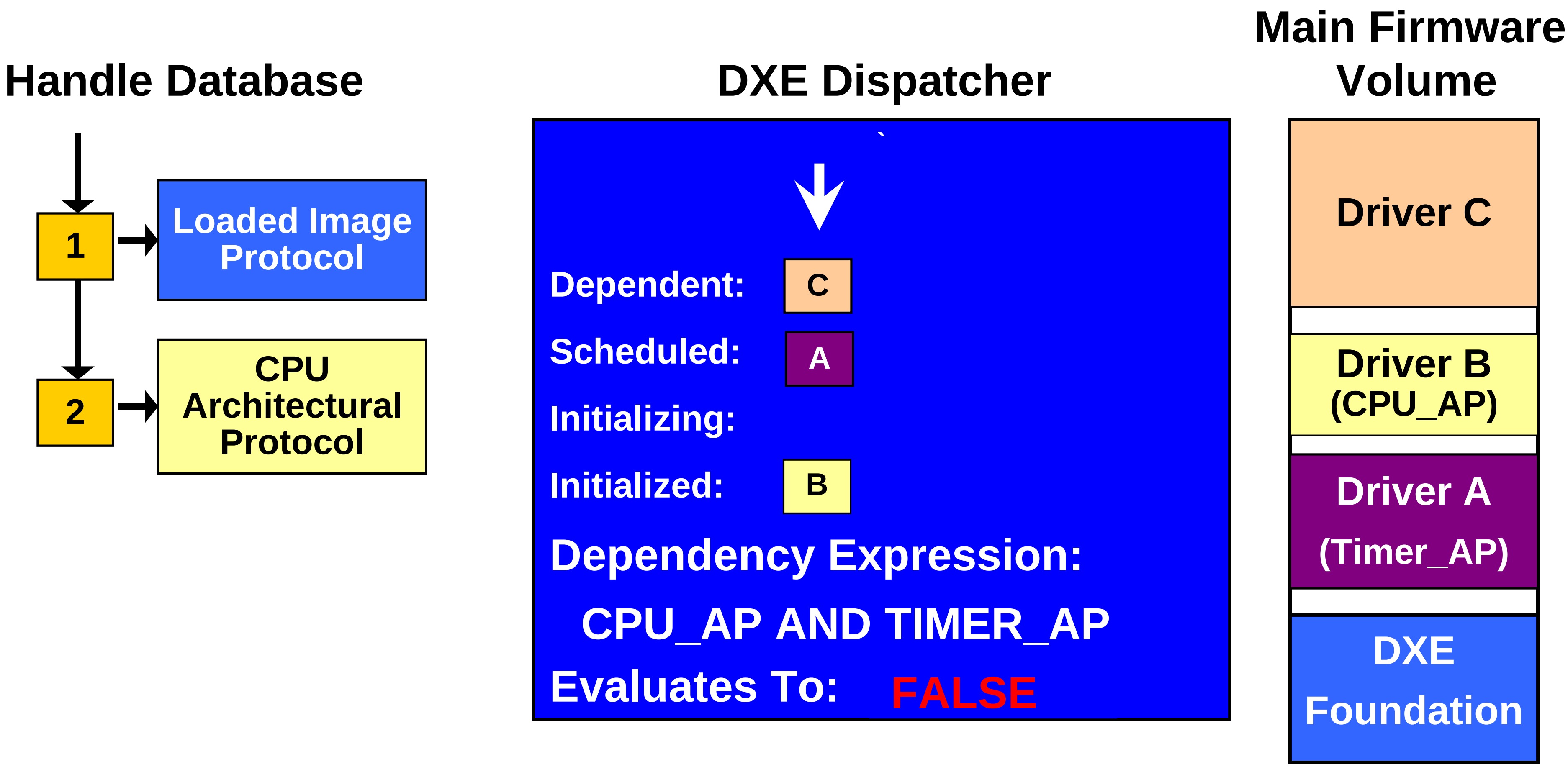
Driver C

Driver B (CPU_AP)

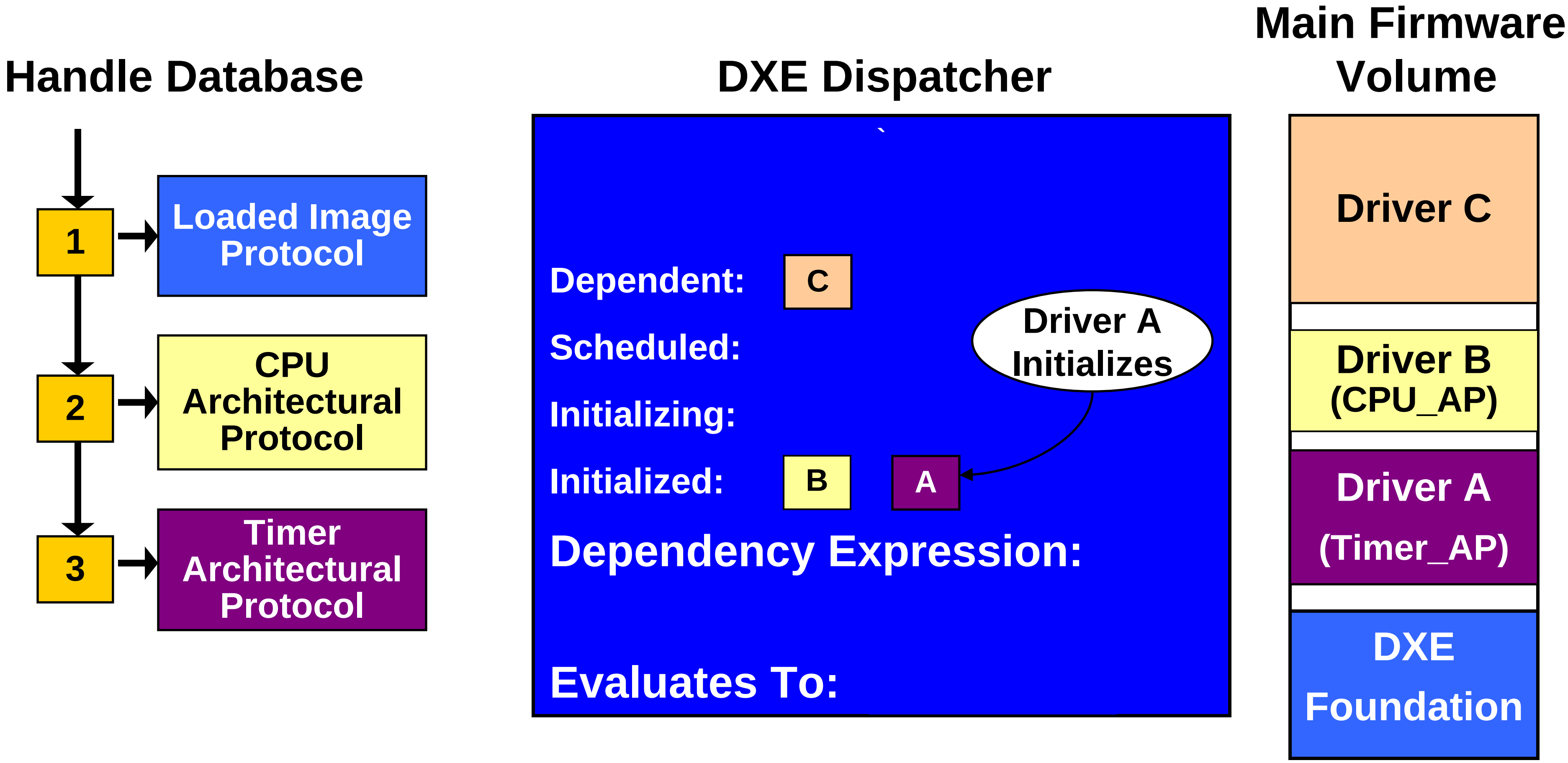
Driver A (Timer_AP)

DXE Foundation

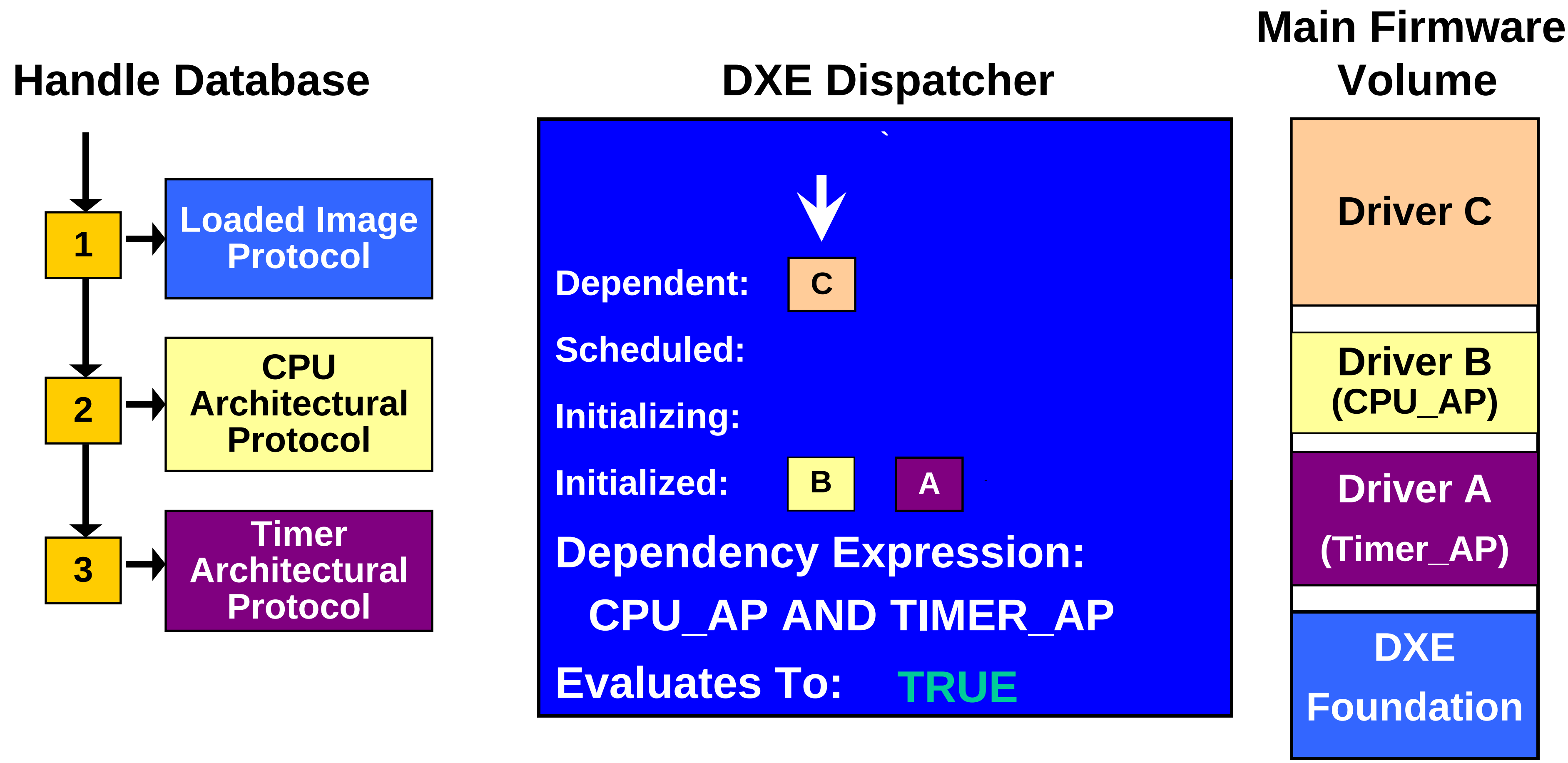
DXE Foundation Dispatcher



DXE Foundation Dispatcher



DXE Foundation Dispatcher



DXE Foundation Dispatcher

Handle Database

1 → Loaded Image Protocol

2 → CPU Architectural Protocol

3 → Timer Architectural Protocol

DXE Dispatcher

Dependent:

Scheduled:

Initializing:

Initialized: B A C

Driver C Initializes

Dependency Expression:

Evaluates To:

Main Firmware Volume

Driver C

Driver B (CPU_AP)

Driver A (Timer_AP)

DXE Foundation

Execution Order Determined at Runtime Based on Dependencies

EXAMPLE Firmware Orderings

Firmware Volume

Firmware Volume	Dependency / Produces
***A Priori* File** Security Driver Runtime Driver Variable Driver	
Runtime Driver	Depex= TRUE END Produces : EFI_RUNTIME_ARCH_PROTOCOL
CPU Driver	Depex= TRUE END Produces : EFI_CPU_IO_PROTOCOL, EFI_CPU_ARCH_PROTOCOL
Timer Driver	Depex= EFI_CPU_IO_PROTOCOL AND EFI_CPU_ARCH_PROTOCOL END Produces : EFI_TIMER_ARCH_PROTOCOL
Metronome Driver	Depex= EFI_CPU_IO_PROTOCOL END Produces : EFI_METRONOME_ARCH_PROTOCOL
Variable Driver	Depex= TRUE END Produces : EFI_VARIABLE_ARCH_PROTOCOL, EFI_VARIABLE_WRITE_ARCH_PROTOCOL
Reset Driver	Depex= EFI_CPU_IO_PROTOCOL END Produces : EFI_RESET_ARCH_PROTOCOL
DXE Foundation	
BDS Driver	Depex= TRUE END Produces : EFI_BDS_ARCH_PROTOCOL
Security Driver	Depex= TRUE END Produces : EFI_SECURITY_ARCH_PROTOCOL

See § 10.11 PI 1.1 Vol. 2 Spec

Where the DXE Dispatcher calls a Driver's Entry Point

Source code

Location in open source tree:
- EDK I \Foundation\Core\Dxe\Image\Image.c
- EDK II \MdeModulePkg\Core\Dxe\Image\Image.c
- **Call**: **CoreDispatcher() >>> CoreStartImage()**

```
EFI_STATUS
EFIAPI
CoreStartImage (
  IN EFI_HANDLE  ImageHandle,         \\Handle of image to be started
  OUT UINTN      *ExitDataSize,       \\Pointer of the size to ExitDat
  OUT CHAR16 **ExitData  OPTIONAL  \\Pointer to a pointer to a data buffer that includes a Null-terminated Unicode
                 \\string, optionally followed by additional binary data.  )
{
   Image = CoreLoadedImageInfo (ImageHandle);

   // Call the image's entry point
   Image->Started = TRUE;
   Image->Status = Image->EntryPoint (ImageHandle, Image->Info.SystemTable);

   // if the image returned with error.
   // Thus make the user aware and check if the driver image has already released all the resources in this situation
   DEBUG_CODE (
     if (EFI_ERROR (Image->Status)) {
 ...   DEBUG ((EFI_D_ERROR, "Error: Image at %08X start failed: %x\n", Image->Info.ImageBase, Image->Status));   }  )

 return Status;
}
```

Agenda

- DXE Foundation
  - Overview
  - Data Structures
  - Events
  - Core
  - Architectural Protocols (API)
  - Dispatcher
  - Drivers
- SMM Overview

Platform Initialization (PI) Spec Vol 2 and Vol 4

DXE Drivers

- Module that contains device or service code
- Do not call platform specific core functions – keeps the drivers platform neutral
- Two Types

  (1) Early DXE Drivers - Platform Initialization Drivers

  - Execute first in the DXE Phase
  - Contain Dependency Expression Syntax (DEPEX) to describe dispatch order (typically stored in dxs file)
  - Typically contain:
    - Basic services
    - Processor initialization code
    - Chipset initialization code
    - Platform initialization code
  - Produce Architectural Protocols

See § 11 PI 1.1 Vol. 2 Spec

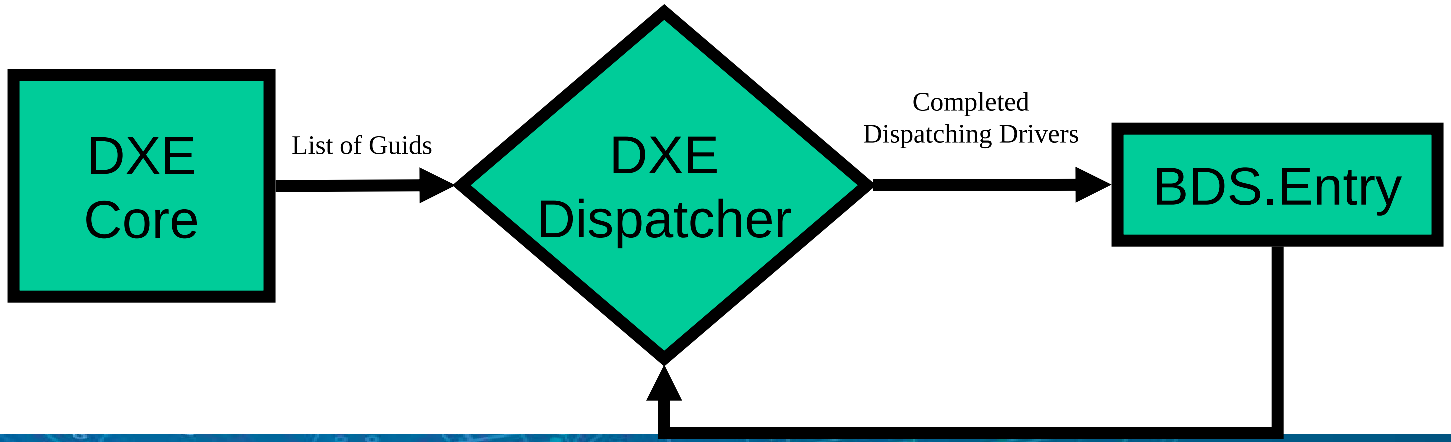
DXE Drivers Cont'd

(2) EFI Drivers - follow the EFI Driver Model

- Do not touch hardware when they initialize
- Follow UEFI driver model (register Driver Binding Protocol)
- Typically provide access to console devices and boot devices
- Abstract Bus controllers
- Only drivers needed to boot OS are initialized (started up)

Last Driver Executed in DXE Boot Device Selection (BDS)

- Invoked after DXE Dispatcher is Complete
- Implemented as a Driver
- Connects EFI Drivers as Required
  - Establishes Consoles (Keyboard, Video)
  - Processes EFI Boot Options (Boots OS)



DXE Summary

- Easy to integrate 3rd party code
  - Silicon Initialization
  - Code from different BIOS vendors
  - OEM code onto ODM board
- Multiple FLASH Devices Supported
- Totally relocateable and Hardware Independent
- Dispatcher enables 3rd party value add
  - OEM can add modules without changing ODM code

Agenda

- DXE Foundation
- SMM Overview

See PI Specification Vol 4 SMM CIS

Agenda

- DXE Foundation
  - Overview
  - Data Structures
  - Events
  - Core
  - Architectural Protocols (API)
  - Dispatcher
  - Drivers
- SMM Overview

Platform Initialization (PI) Spec Vol 4 SMM CIS

System Management Mode Services

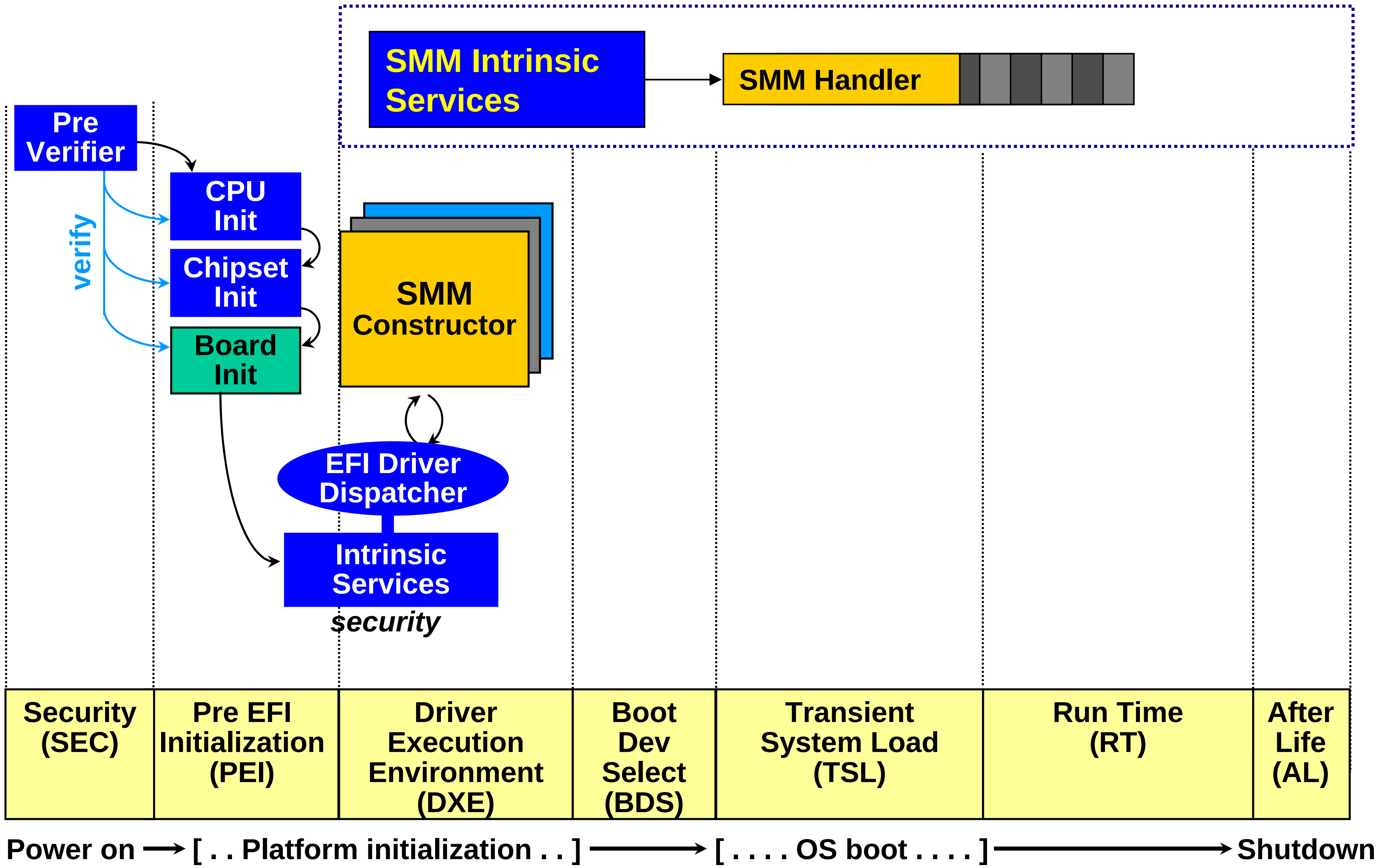
- Registration vehicle for dispatching drivers in response to System Management Interrupts (SMI)
- Dispatch of drivers in System Management Mode (SMM) will not be able to use core protocol services
- SMM handlers will be logically precluded from accessing conventional memory resources
- SmmLib includes a subset of the DXE core services, such as memory allocation, device I/O protocol, and others

What is SMM ?

- System Management Mode (SMM) is a generic term used to describe a unique operating mode of the processor which is entered when the CPU detects a special Interrupt
- High priority System Management Interrupt (SMI).
  - CPU will switch into SMM
  - Jump to a pre-defined entry vector
  - save some portion of its state (the “save state”) such that execution can be resumed.
- Generated by software or by a hardware event
- Each SMI source can be detected, cleared and disabled.
- Special memory (SMRAM) is set aside for software running in SMM.
- Usually the SMRAM is locked after initialization so that it cannot be exposed until the next system reset.

See § 1.2 PI 1.1 Vol. 4 Spec.

PI 1.1 Flow for SMM

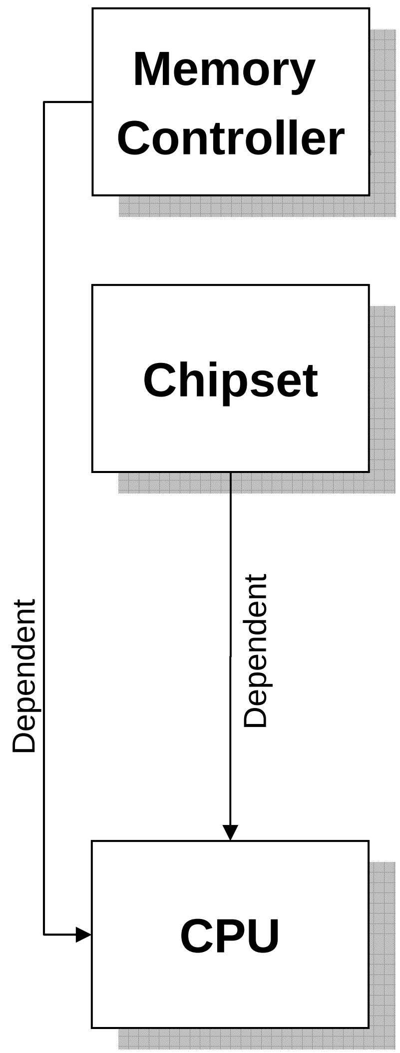


Initializing SMM

- SMM initialization prepares the hardware for SMI generation
- Creates the necessary data structures for managing the SMM resources such as SMM RAM
- It is initialized with the cooperation of several DXE drivers

SMM Initialization Components

DXE Drivers **Produces**



EFI_SMM_ACCESS2_PROTOCOL

SMM RAM Regions Description

EFI_SMM_CONTROL2_PROTOCOL

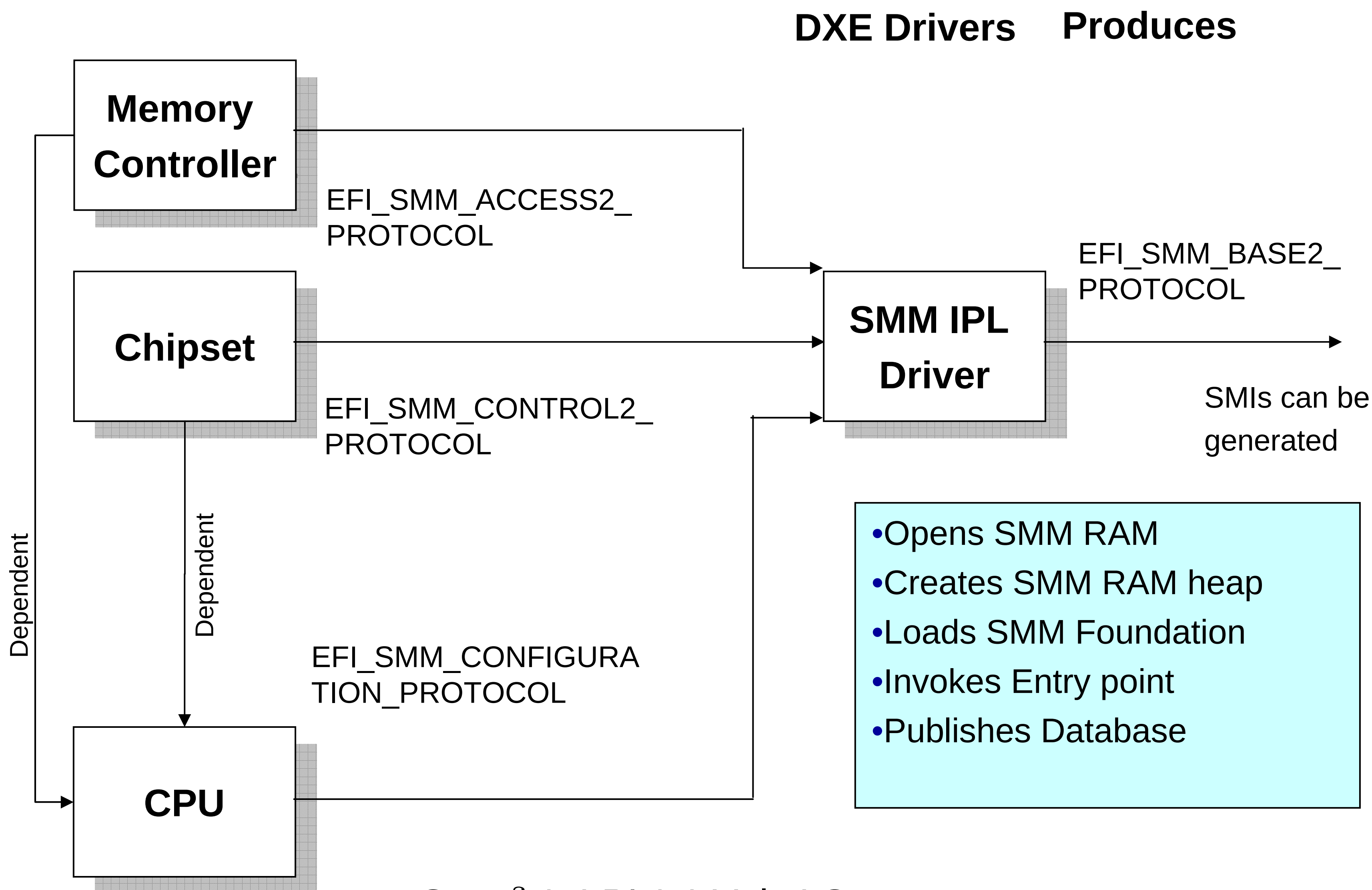
Synchronous SMIs

EFI_SMM_CONFIGURATION_PROTOCOL

1. Initialize SMM entry vector
2. SMM RAM Memory Map

See §1.4 PI 1.1 Vol. 4 Spec.

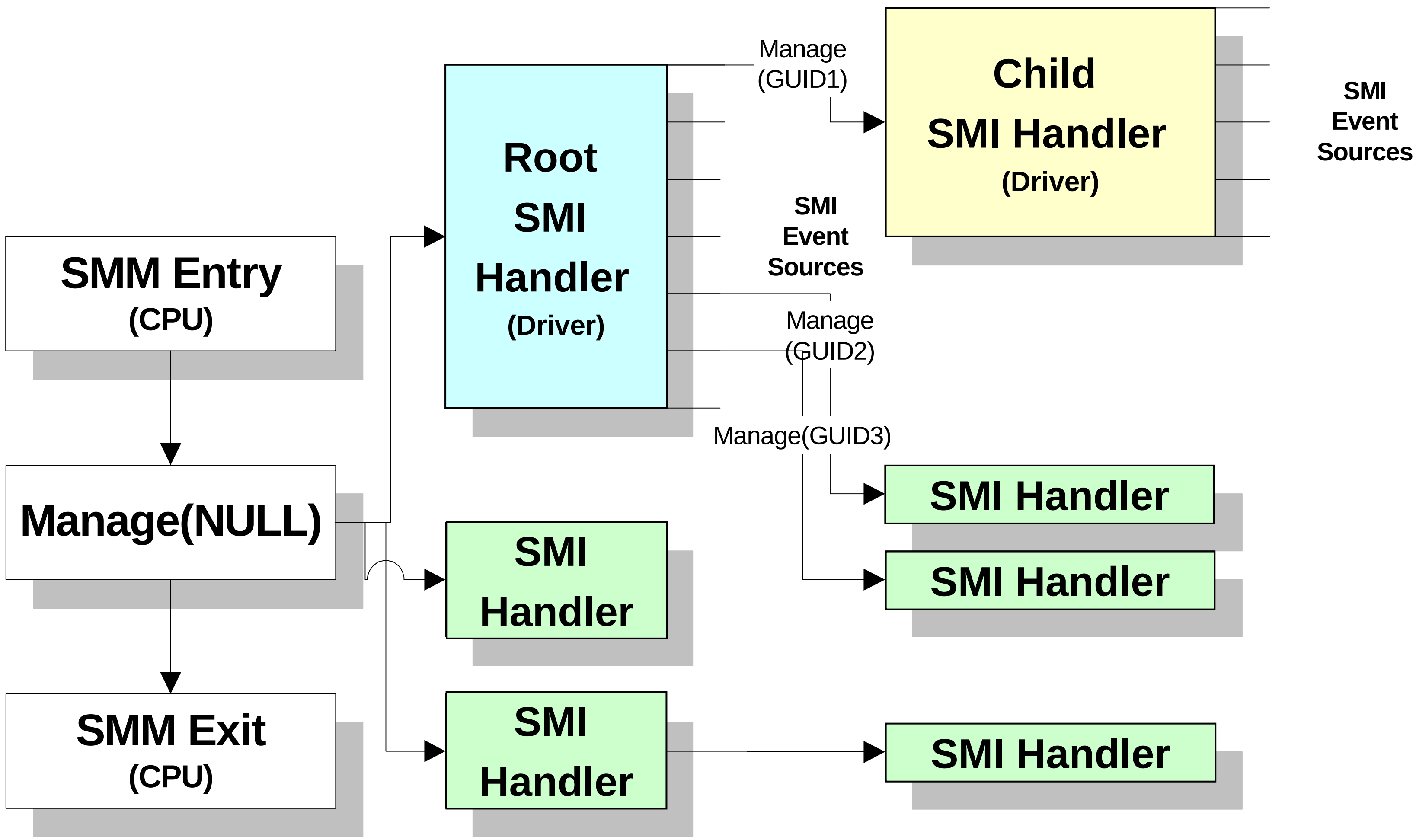
SMM Initialization Components



- Opens SMM RAM
- Creates SMM RAM heap
- Loads SMM Foundation
- Invokes Entry point
- Publishes Database

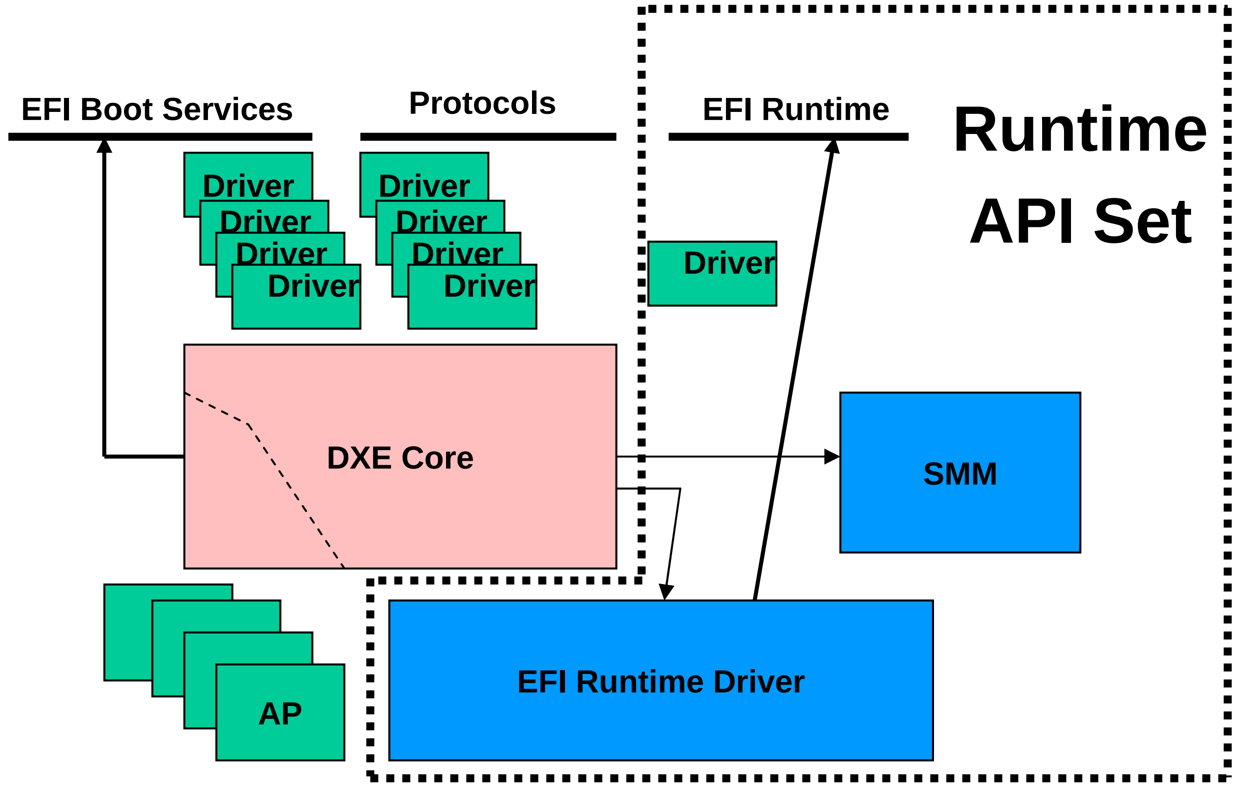
See § 1.4 PI 1.1 Vol. 4 Spec.

Event Handlers



See Backup for Itanium® Processor Family Platform Management Interrupt (PMI)

DXE Construction with SMM



Same DXE Core Binary Works in OS hosted and Native UEFI Systems

SMM Summary

- SMM is modular and similar to the DXE phase
- Platform Management Interrupt (PMI) is on Itanium® platforms
- SMM/PMI share the feature of **OS-transparency**

Q & A

Back up

Event Types

- Timer event – used to delay by a certain time
- Runtime event – an event that will be used after ExitBootServices()
- Notify Wait event – an event whose function is not spontaneously called
- Notify signal event – an event whose function is spontaneously called
- Exit Boot Services event – the special event that signals that ExitBootServices() has been called.

Three Elements of all Events

- **The Task Priority Level** (TPL) of the event
  - Priority at which the notification function is executed
- **A notification function**
  - Is executed when the state of the event is checked or the event is being waited upon.
  - The notification function of a signal event is executed whenever the event transitions from the waiting state to the signaled state
- **A notification context**
  - is passed into the notification function each time the notification function is executed

Event Usage

- CreateEvent() – creates an event structure
- CreateEventEx() – creates an event in a group
- CloseEvent() – closes and free event structure
- SignalEvent() – sets event to signaled state
- WaitForEvent() – stops execution until signaled
- CheckEvent() – checks the state of an event

Event Usage *(continued)*

- Call CreateEvent() to create an event handle
  - Or use CreateEventEx() to make a set of events
- Wait on or periodically check the state of the event
  - Or wait for the function to be called automatically
- Signal the event

	CreateEvent()	CreateEventEx()	WaitForEvent()	Calls a function
Wait Event type	Yes	Yes	Yes	No
Signal Event type	Yes	Yes	No	Yes
Set of events	No	Yes	OR	OR
Single event	Yes	No Effect	OR	OR

Timer Usage

1. Create an event using CreateEvent() with EVT_TIMER bitmask.
2. Set the timer using SetTimer()
   1. Use TimerCancel to cancel an existing timer
   2. Use TimerPeriodic to set a repeating timer
   3. Use TimerRelative for a single event
3. If this is a "one-shot" item close the event with CloseEvent() immediately.

Source code

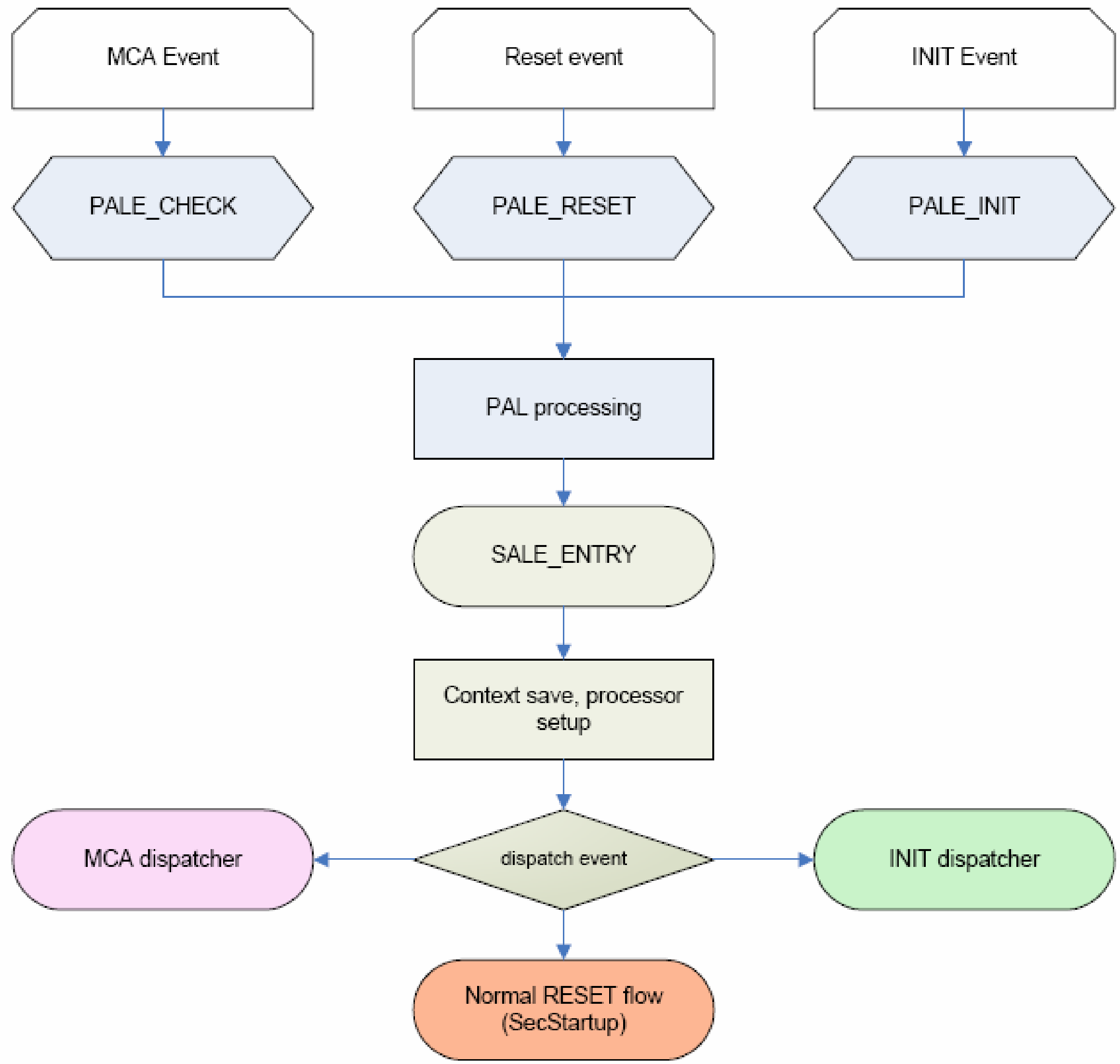
Example Timer Usage

```
// Create the event.
Status = gBS->CreateEvent (
     (EFI_EVENT_NOTIFY_SIGNAL | EFI_EVENT_TIMER),
     EFI_TPL_NOTIFY,
     EFI_EVENT_NOTIFY* MyEventFunctionToCall,
     NULL,
     &Event
     );
// Set off event for every ½ second.
Status = gBS->SetTimer (
     Event,
     TimerPeriodic,
     5000000
     );
// note that the time is in 100ns so 5000000 is ½ second.
```

SMM on the Itanium® Processor Family

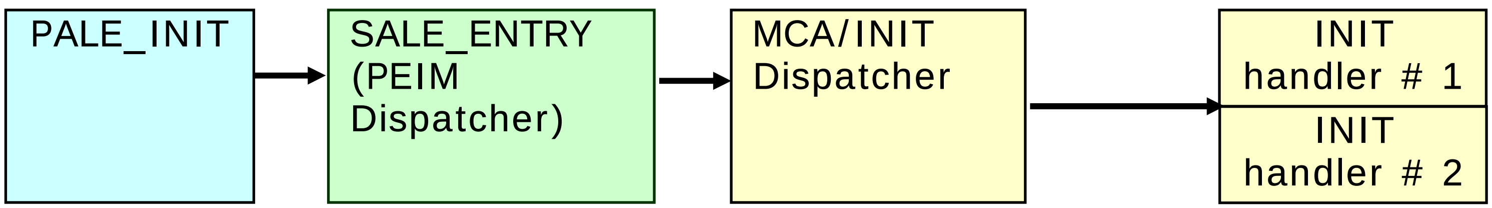
- Itanium® processor family - mode of firmware operation that is invoked by the Platform Management Interrupt (PMI)
- Control passed to firmware in response to the PMI
- The characteristic that PMI-based firmware on Itanium® processors and SMI-based firmware on IA-32 (X86) share is the **OS-transparency**

Machine Check Abort (MCA)Interrupt and INIT



Platform Management Interrupt

- PMI similar model to SMM
  - Installable drivers
  - Event dispatch
- Beyond PMI events, infrastructure allows for registration of Machine Check Abort (MCA) and INIT
- PMI handlers useful for Reliability/Availability/Serviceability (RAS), such as CPU and memory hot-plug
- MCA useful for error logging/containment



PMI / MCA / INIT model for SI, OEM contributions