

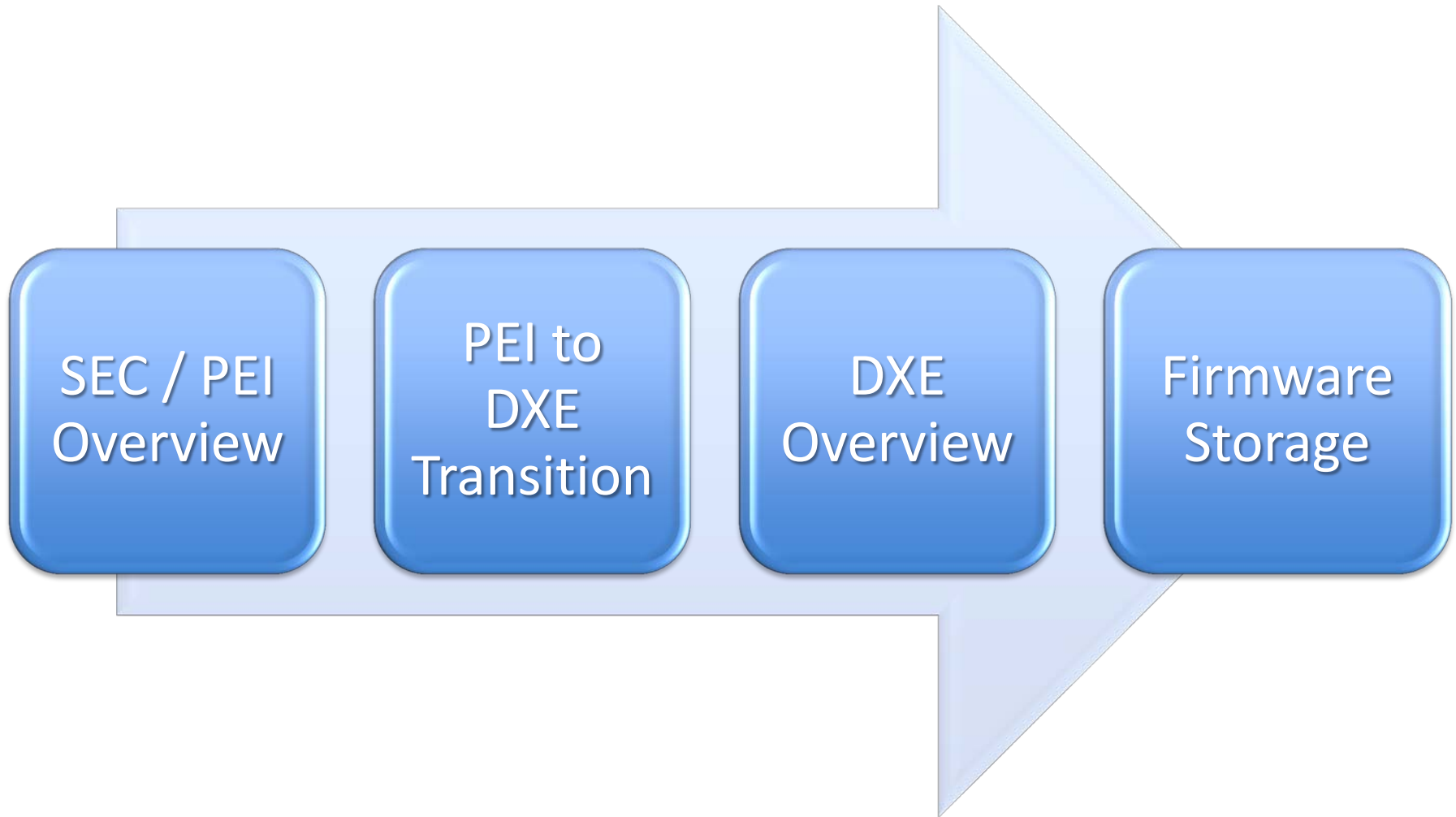
UEFI & EDK II Base Training

Pre-EFI (PEI) and Driver Execution Environment (DXE) Foundation Technical Overview

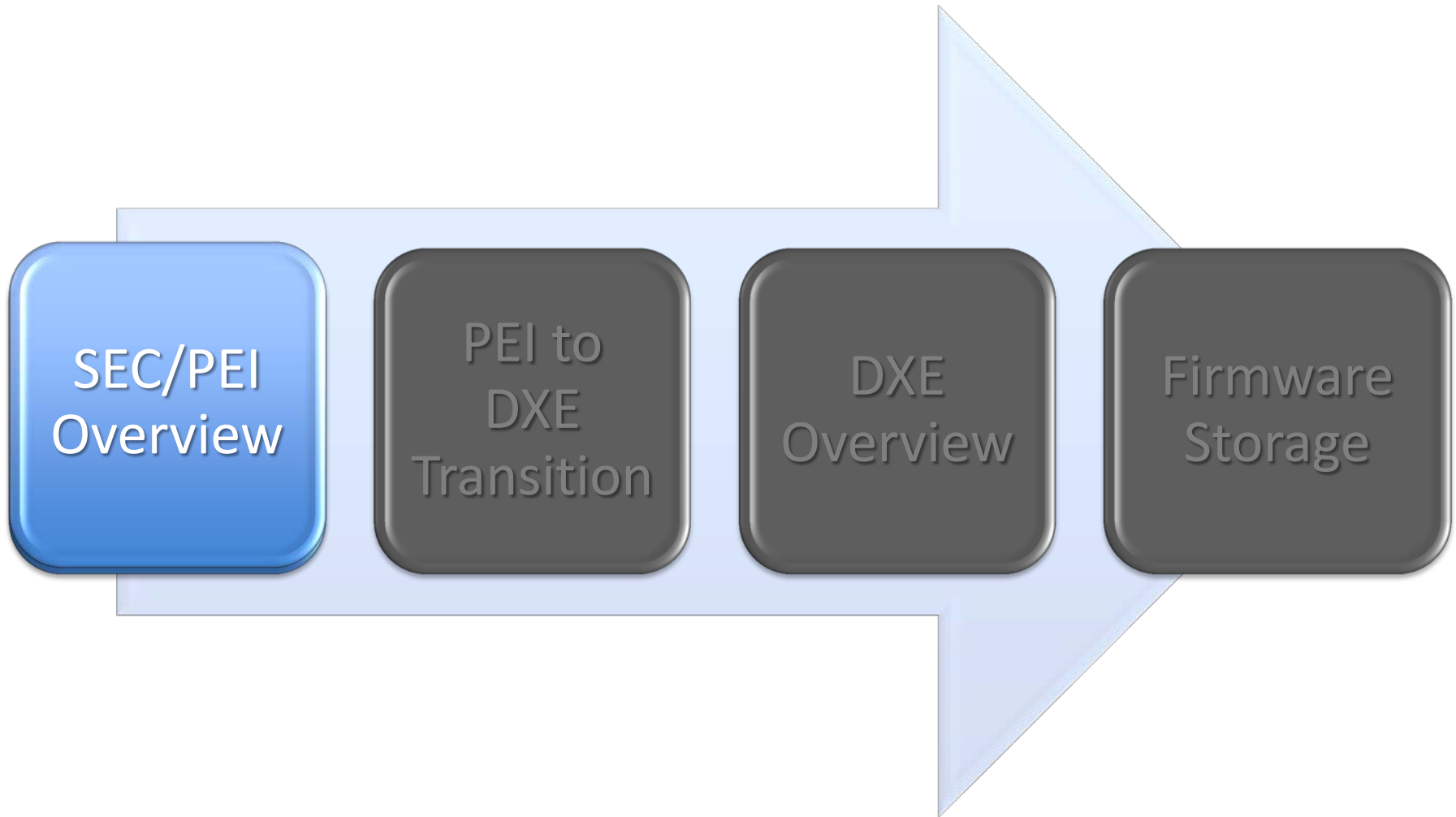
Intel Corporation
Software and Services Group



Agenda



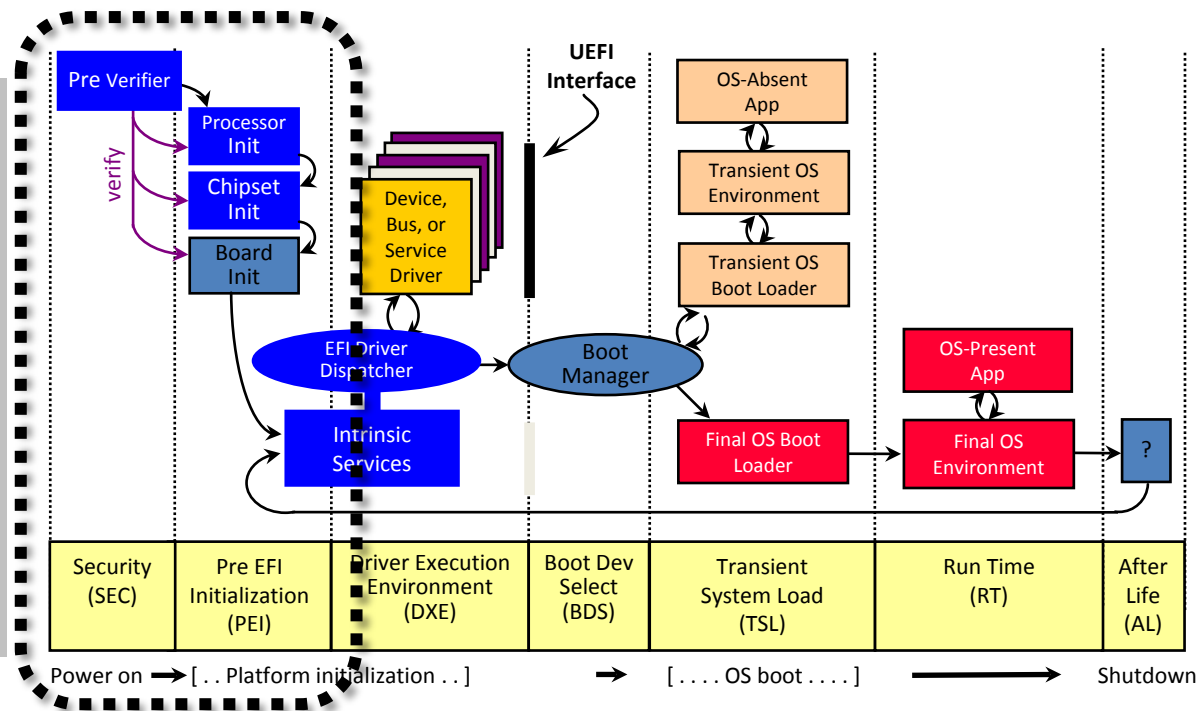
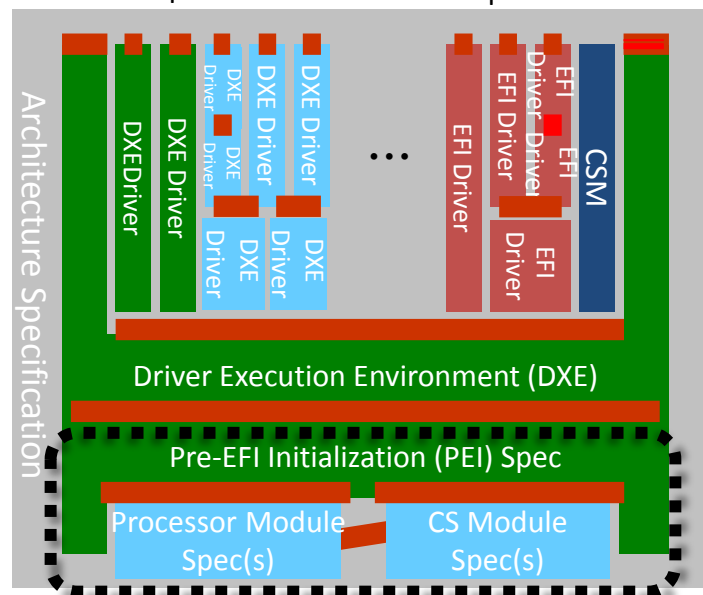
Agenda



PEI Foundation

Chipset/Processor
Function DXE Driver
specs

OEM, ISV &
Intel BU EFI Driver
specs



Why Sec / PEI ?

- Writing Modular code without memory is hard
- Legacy code is hand coded to a variety of different register rules, so it isn't consistent
 - IBV A uses EBP and IBV B used EBX other registers are preserved by code convention
 - Porting from A to B requires rewriting lots of code!
- Quick Path for Memory Initialization and basic chipset initialization
- Modular Code for S3 and Recovery

SEC Phase

- **Security (SEC) is the first phase in the PI Architecture and is responsible for:**
 - Handling all platform restart events
 - Creating a temporary memory store
 - Serving as the root of trust in the system
 - Initial code that takes control of the system
 - May choose to authenticate the PEI Foundation
 - Passing handoff information to the PEI Foundation

Processor
Architecture
Dependent

Platform
Dependency



Initialization Steps IA32 Intel Architecture

SEC Code Start

- Location in open source tree:
 - EDK II (Intel X86)
UefiCpuPkg\ResetVector\Vtf0\la16\ResetVectorVtf0.asm
- Entry point – Processor Reset Vector

Source
Code
Example

```
; The VTF signature
; VTF-0 means that the VTF (Volume Top File);
vtfSignature:
    DB    'V','T','F',0
ALIGN    16
resetVector:

; Reset Vector
; This is where the processor will begin execution
    nop
    nop
    jmp     short EarlyBspInitReal16

ALIGN    16

fourGigabytes:
```

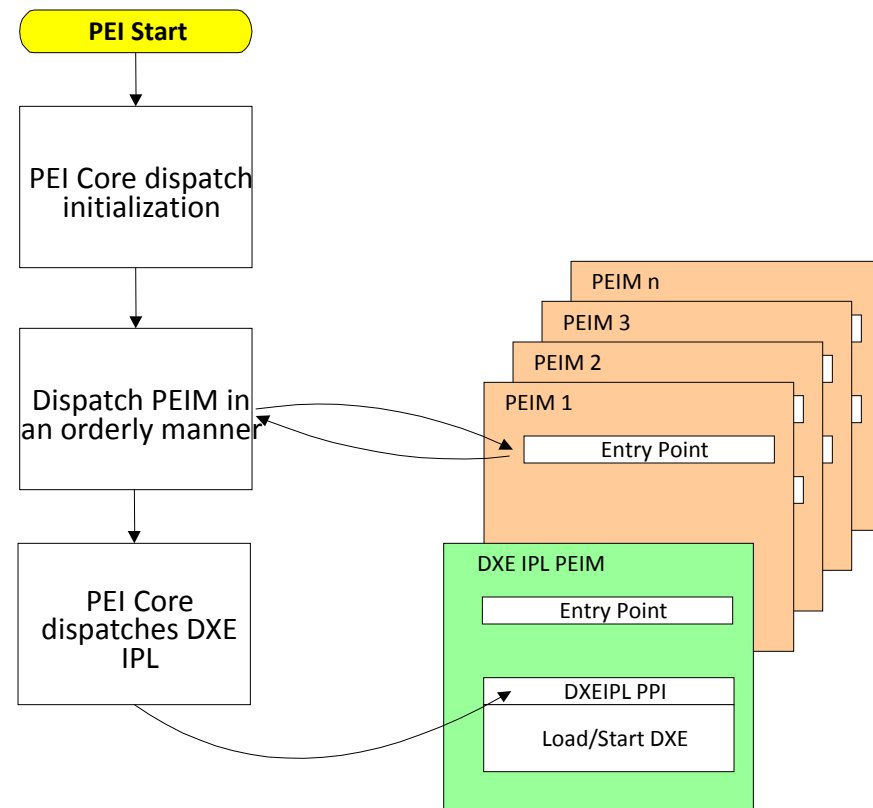
Binary of the compiled
Firmware image

007fffe0	eb db 90 90	90 90 90 90	00 00 00 00	56 54 46 00	ëÛVTF.
007ffff0	90 90 eb c4	90 90 90 90	90 90 90 90	90 90 90 90	ëÄ	

Nt32Pkg – Nt32Pkg\Sec\Secmain..c

Software and Services Group

- **PEI = Pre-EFI Initialization**
- **Small, tight startup code**
 - Startup with transitory memory store for call-stack (i.e., cache)
 - XIP from ROM
- **Core locates, validates, and dispatches PEIMs**
- **Publishes own protocol and call-abstraction w/ PPI**
 - Silicon/platform abstractions
- **Primary goals**
 - Discover boot mode
 - Launch modules that initialize main memory
 - Discovery & launch DXE core-
Convey platform info into DXE



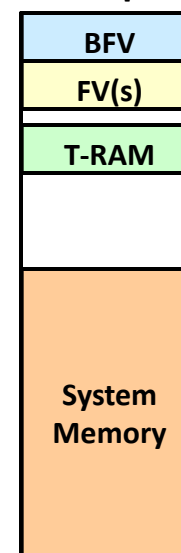
PEI Terminology


- **PEI Core** – The main PEI executable binary responsible for dispatching PEIM and provide basic services.
- **PEIM** – An executable binary that is loaded by the core to do various tasks and initializations.
- **PPI** – PEIM to PEIM Interface. An interface that allows a PEIM to invoke another PEIM.
- **PEI Dispatcher** – The part of the PEI core that searches for and executes the PEIM.
- **PEI Services** – Functions provided by the core visible to all PEIM.

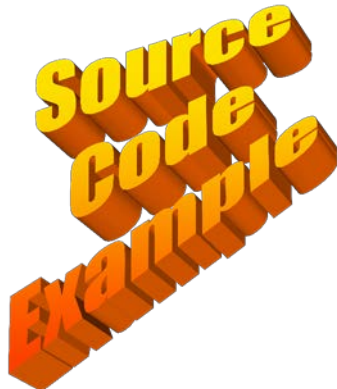
PEI – Initial Memory

- Minimum requirement for PEI is a small amount of temporary RAM
 - Minimum amount is dependent upon processor architecture requirements
 - Size set by parameters for SEC phase
- PEI Temporary RAM requirement is met by architecturally defined mechanisms to allow processor cache to avoid data evictions, allowing the cache to be used as RAM
 - Processor family specific mechanism
 - Example: Intel Pentium 4 is different than Intel Pentium III
 - Itanium (IPF) abstracts mechanism with a PAL call

Memory Map



- 
- PEIMAIN is the main core source code module
 - invoked by PeiMain during transition from SEC to PEI
 - Location in open source tree:
 - EDK I \Foundation\Core\Pei\PeiMain
 - EDK II \MdeModulePkg\Core\Pei\PeiMain
 - Entry point - PeiCore

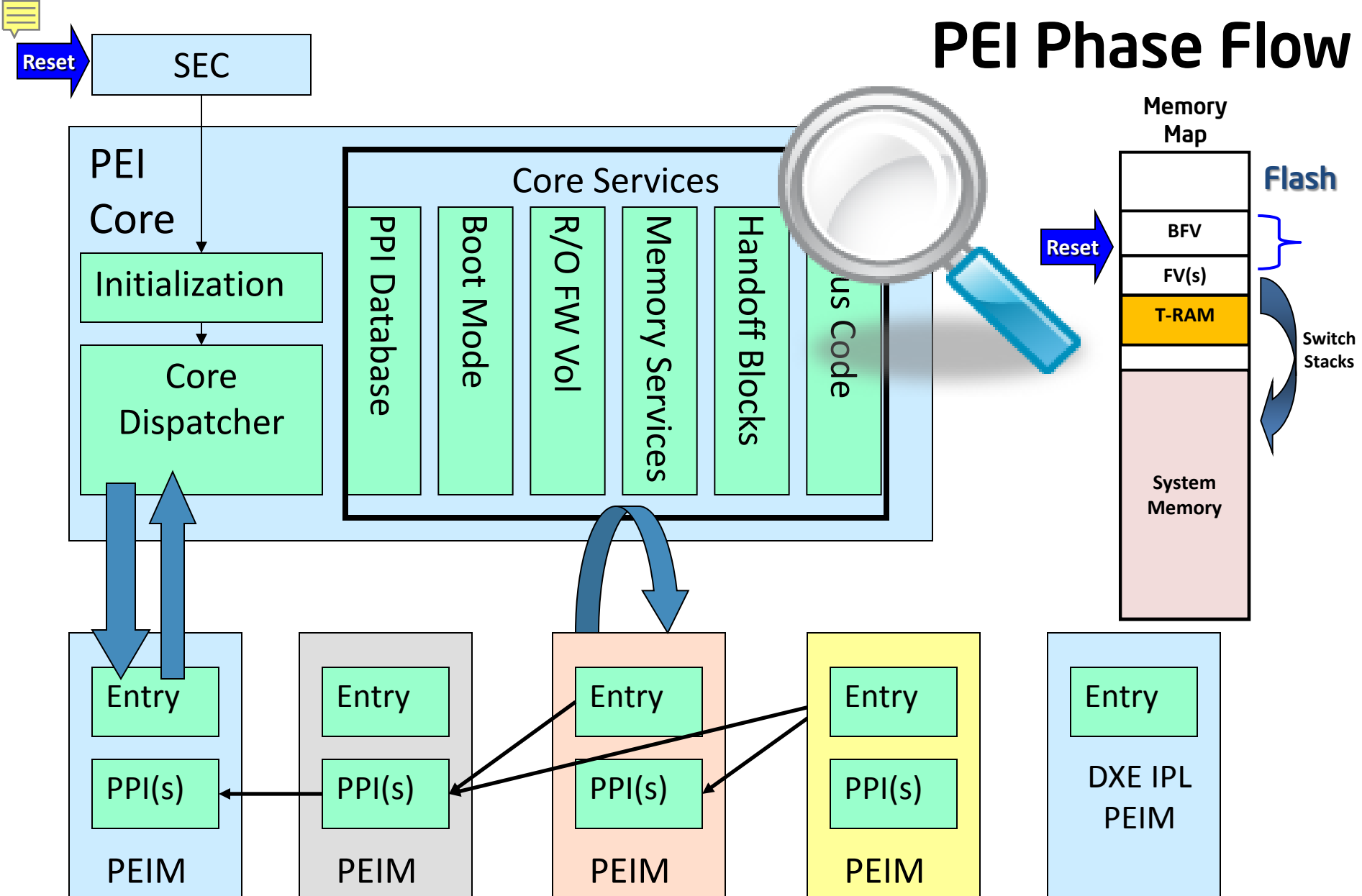


```
VOID
EFIAPI
PeiCore (
    IN CONST EFI_SEC_PEI_HAND_OFF          *SecCoreData,
    IN CONST EFI_PEI_PPI_DESCRIPTOR        *PpiList,
    IN VOID                                *Data
)
{
    ...
    // Initialize PEI Core Services
    InitializeMemoryServices (&PrivateData, SecCoreData, OldCoreData);
    InitializePpiServices    (&PrivateData, OldCoreData);
    ...

    // Call PEIM dispatcher
    PeiDispatcher (SecCoreData, &PrivateData);
    ...
    // Lookup DXE IPL PPI
    ...
}
```

¹Open Source EFI Developer Kit (EDK & EDK II) <http://www.tianocore.Sourceforge.net>

PEI Phase Flow

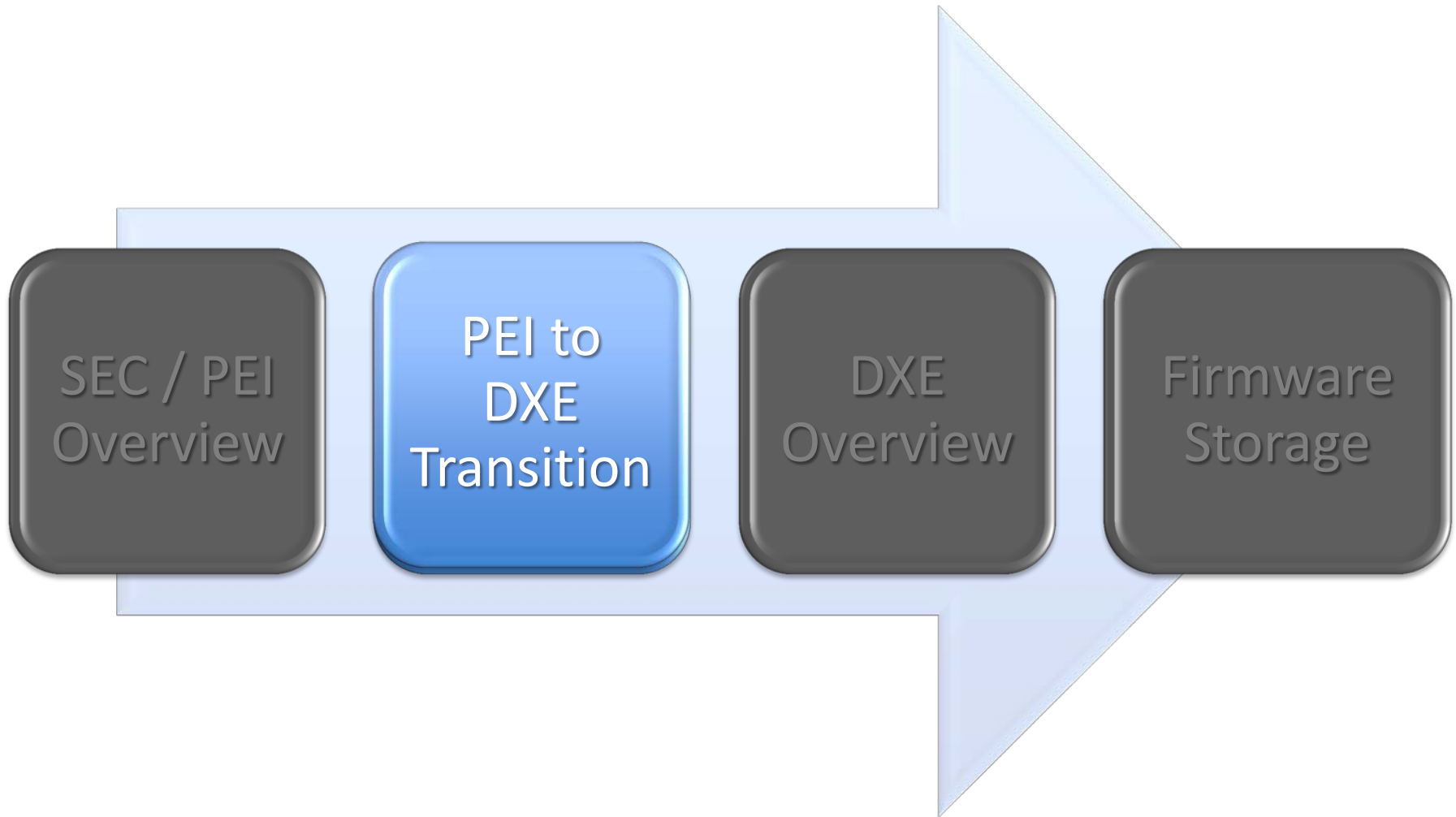


PEI Services

PPI Services	Manages PEIM-to-PEIM Interface (PPIs) to facilitate intermodule calls between PEIMs. Interfaces are installed and tracked on a database maintained in temporary RAM.
Boot Mode Services	Manages the boot mode (S3, S5, normal boot, diagnostics, etc.) of the system.
HOB Services	Creates data structures called Hand-Off Blocks (HOBs) that are used to pass information to the next phase of the PI Architecture.
Firmware Volume Services	Walks the Firmware File Systems (FFS) in firmware volumes to find PEIMs and other firmware files in the flash device.
PEI Memory Services	Provides a collection of memory management services for use both before and after permanent memory has been discovered.
Status Code Services	Provides common progress and error code reporting services (for example, port 080h or a serial port for simple text output for debug).
Reset Services	Provides a common means by which to initiate a warm or cold restart of the system.

See § 4.1 PI Vol. 1 Spec
Software and Services Group

Agenda





End Conditions of PEI

- A dispatch pass is done when no PEIM is dispatched during a pass through the known PEIMs
- We Have Memory
 - Report Data For Subsequent DXE Phase
 - Use HOB's for the hand-off information
- Done with dispatch: Invoke the DXE Initial Program Load (IPL)

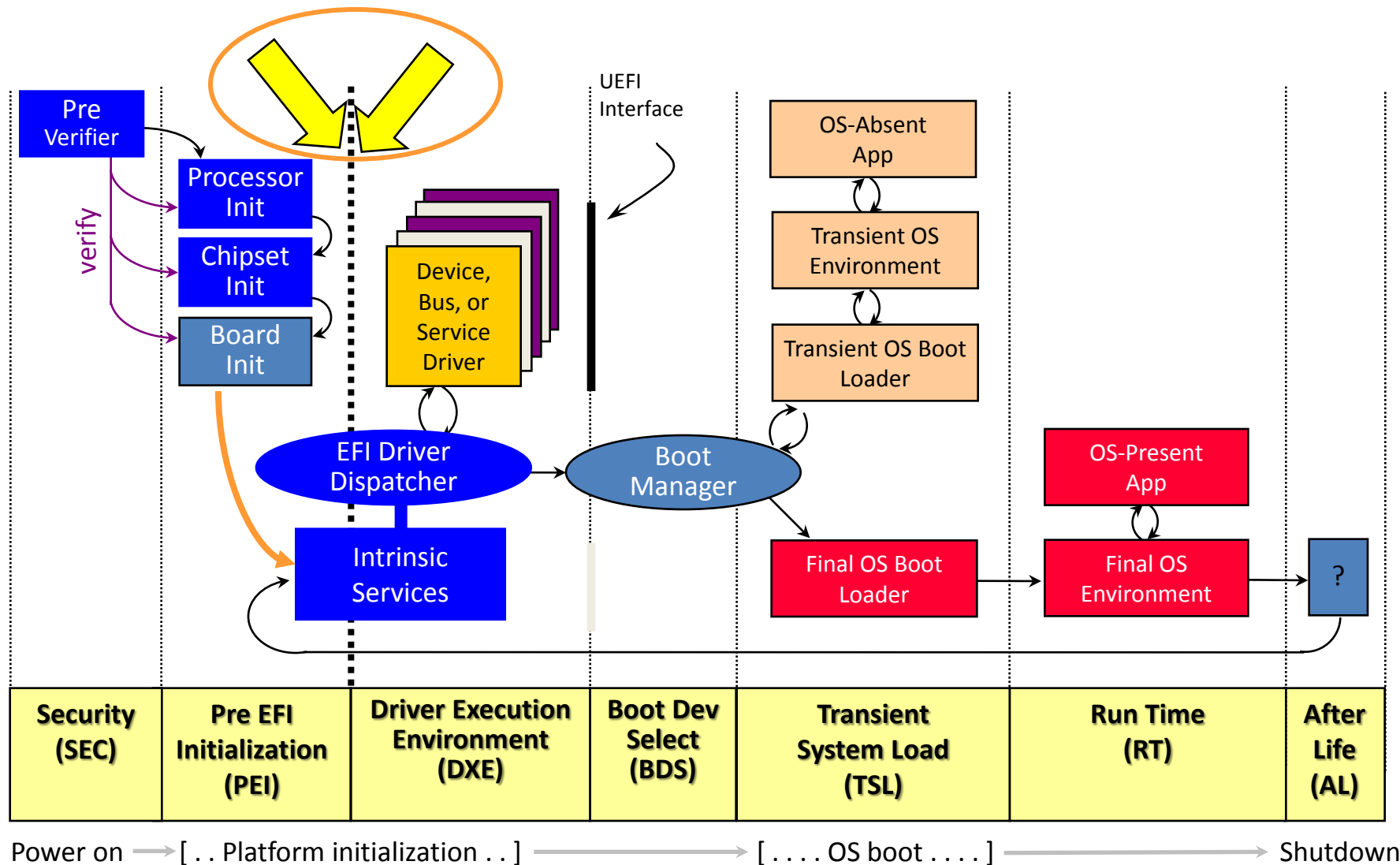


Hand Off Blocks

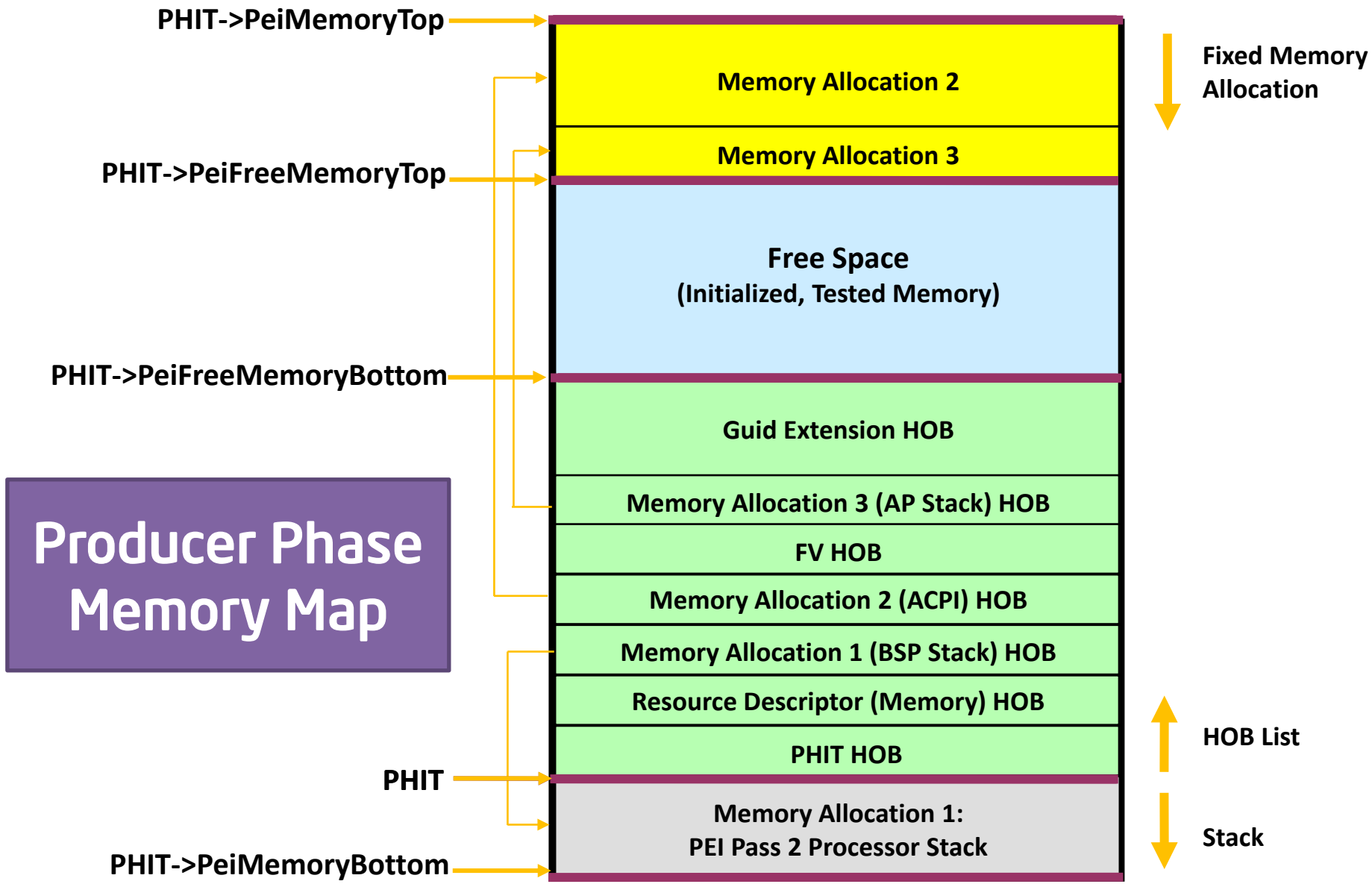
- HOBs bridge the gap between PEI and DXE
 - PEI collects the state in HOBs
- HOBs – a series of data structures in memory, created during PEI, that describe platform features, configuration, or data. HOBs are produced during PEI, and read-only during DXE (consumer).
- PEI must build HOB:
 - PEI Handoff Information Table (PHIT)
 - Resource Descriptor for physical system memory
 - Memory Allocation HOB - BSP Stack



Architecture Execution Flow



PEI Hand-off Block (HOB)

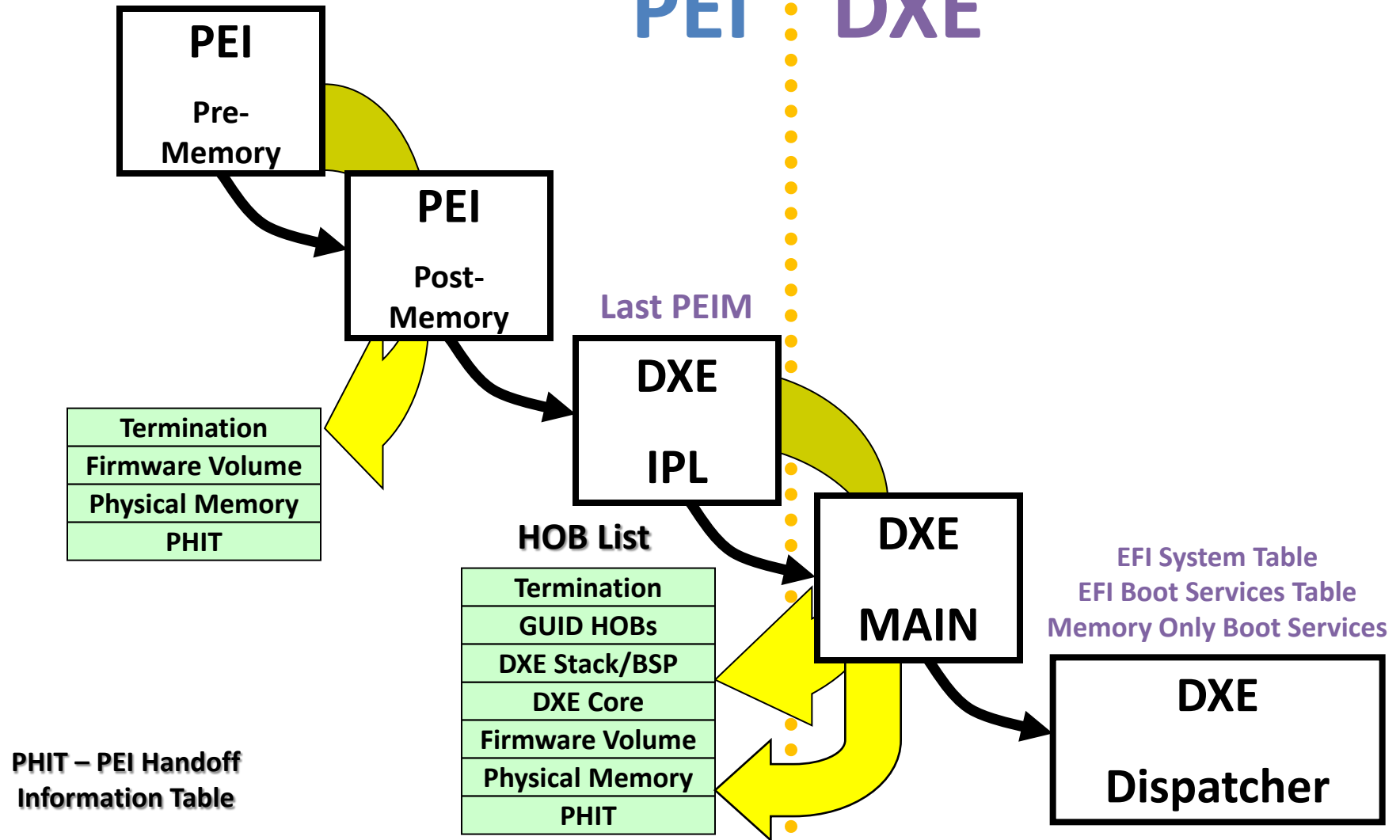


See § 4 PI Vol. 3 Spec

Software and Services Group

PEI to DXE Transition

PEI : DXE



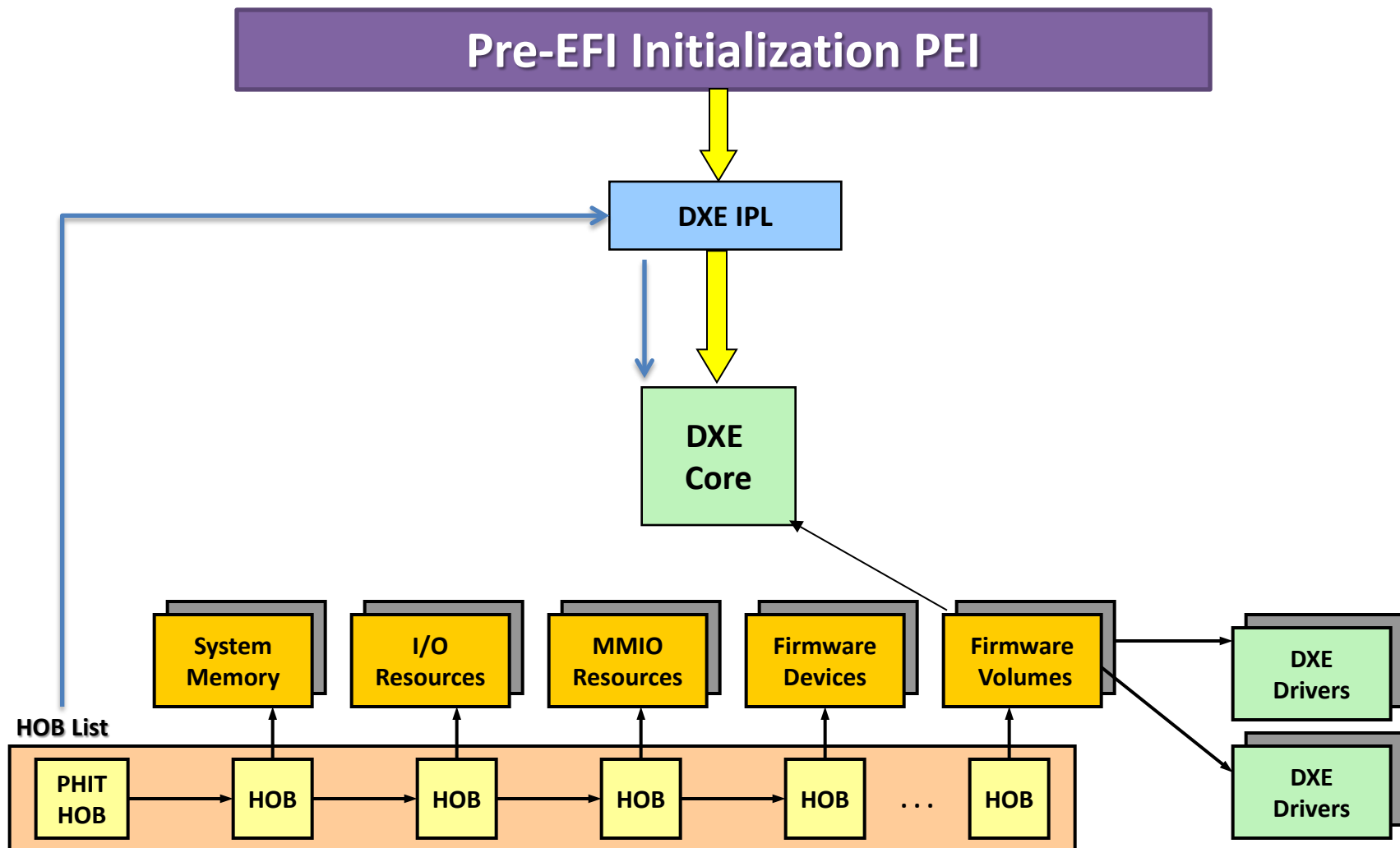


DXE Initial Program Load (IPL)

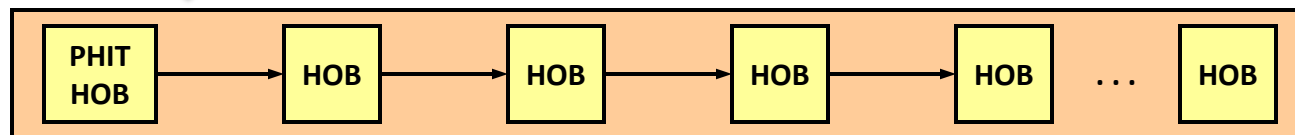
- No hard coded addresses allowed
- Find Largest Physical Memory HOB
 - Ideally this should be near Top Of Memory (TOM)
- Allocate DXE Stack from Top of Memory
 - Build HOB that describes DXE Stack
- Search FVs from HOB List for DXE Core
- Load DXE Core into Memory (PE/COFF)
 - Build HOB that describes DXE Core
- Switch Stacks and Handoff to DXE Core



PEI to DXE Entry



PEI to DXE Entry



DXE Foundation

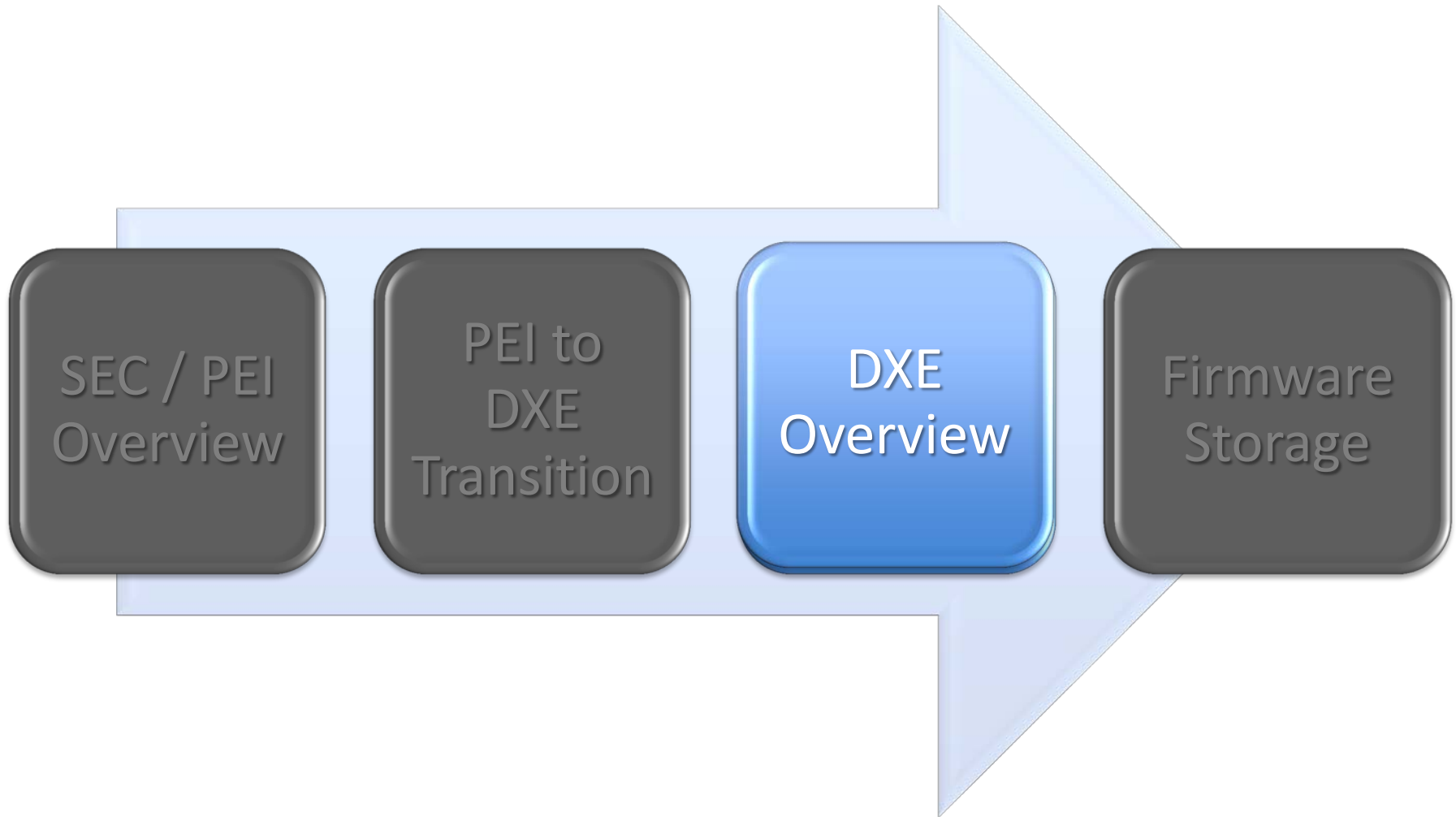
Architectural Protocols

DXE Drivers



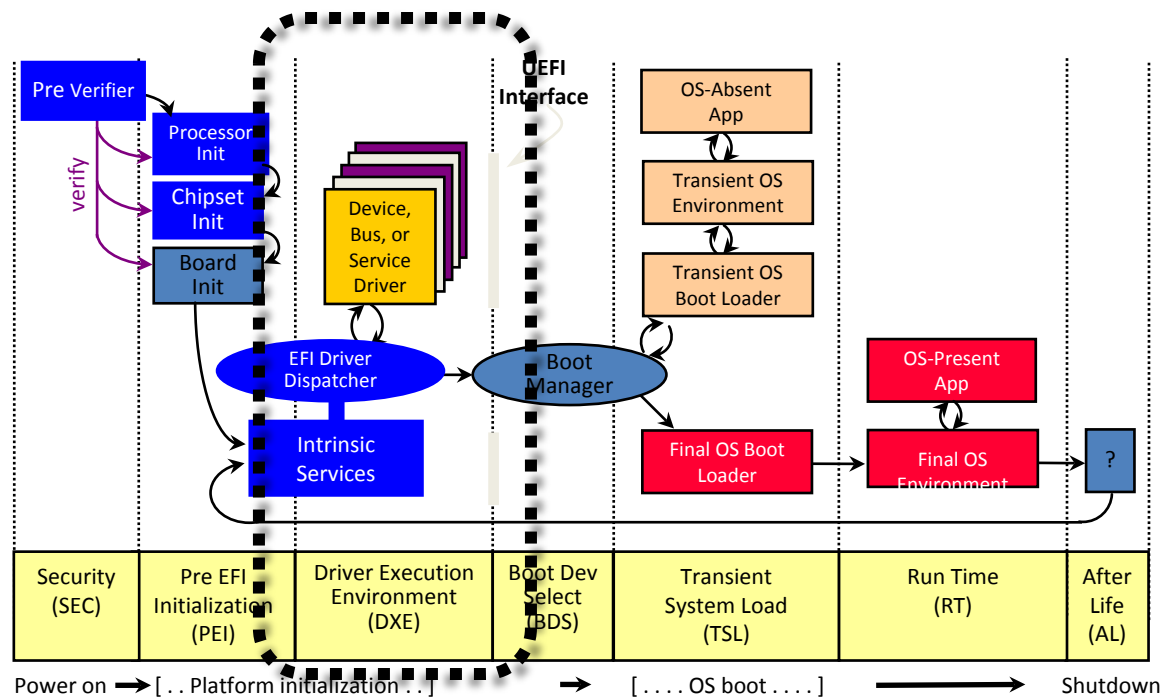
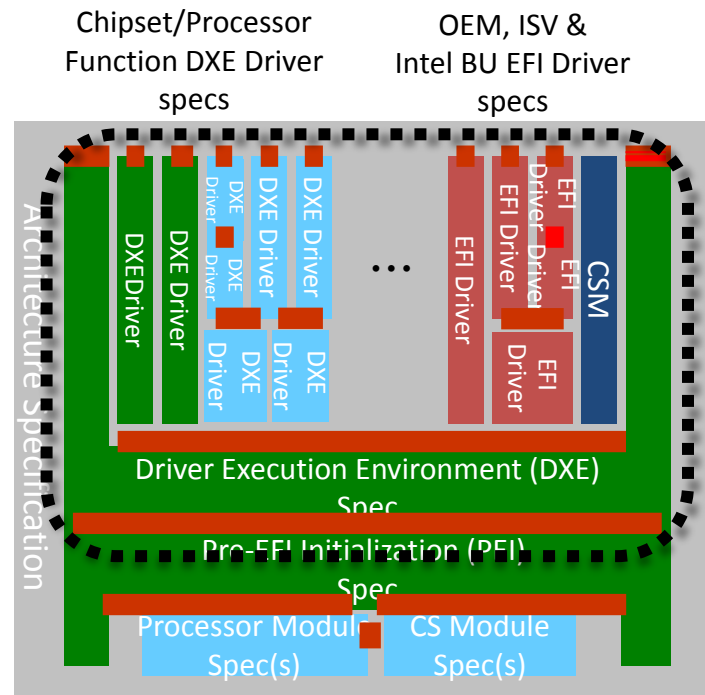
Direct Access to Hardware in DXE uses Architectural Protocols

Agenda



DXE Foundation

Driver Execution Environment



Most features during DXE phase implemented as DXE drivers

Why DXE?

- **Without DXE (legacy BIOS)**
 - BIOS features coded in proprietary fashion
 - ODM has to port features from IBV #1 to IBV #2
 - Third parties can not easily provide value added pre-OS features that are independent of the IBV codebase
 - Single source file controls boot flow
- **With DXE**
 - Modular system for firmware expansion
 - Standard driver model allows a large number of companies to create pre-OS features for UEFI BIOS

What is DXE Foundation Theory of Operation?

Initialize Platform

- Initialize chipset and platform

Initialize Services

- Loads drivers to construct environment that can support boot manager and OS boot

Dispatch DXE Drivers

- Dependencies provide driver ordering
- Logic based grammar description of drivers' requirements based on GUID (AND OR)

Dispatch UEFI Drivers

- UEFI drivers with no dependency started last
- EFI 1.10 drivers, UEFI 2.x drivers, IHV cards, etc

Load Boot Manager

- Required hardware init performed by driver on call to entry point
- UEFI driver entry points just register protocol
- Defer initialization of boot devices until we know which are needed

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

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 UEFI service tables (boot & runtime services, handle database, ...)
- **Events** – A means of messaging, signaling and passing control in response to some other activity

DXE Main

Responsible for Initializing DXE Core

- Consumes HOB List
- Builds UEFI and DXE Service Tables
 - EFI System Table
 - UEFI Boot Services Table
 - UEFI Runtime Services Table
 - 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()**
 - DXE Drivers will build the Architectural Protocols
 - May require decompression service

DXE Main in Source Code

```
VOID
EFIAPI
DxeMain (
    IN VOID *HobStart // Pointer to the beginning of the HOB List from PEI
)
{
    ...
    // Initialize ...
    // Initialize Memory Services
    CoreInitializeMemoryServices (&HobStart, &MemoryBaseAddress, &MemoryLength);
    ...
    // Invoke the DXE Dispatcher
    CoreDispatcher ();
    ...
    // Transfer control to the BDS Architectural Protocol
    gBds->Entry (gBds);
    // BDS should never return
    //
    ASSERT (FALSE);
    CpuDeadLoop ();
}
```

Source
Code
Example

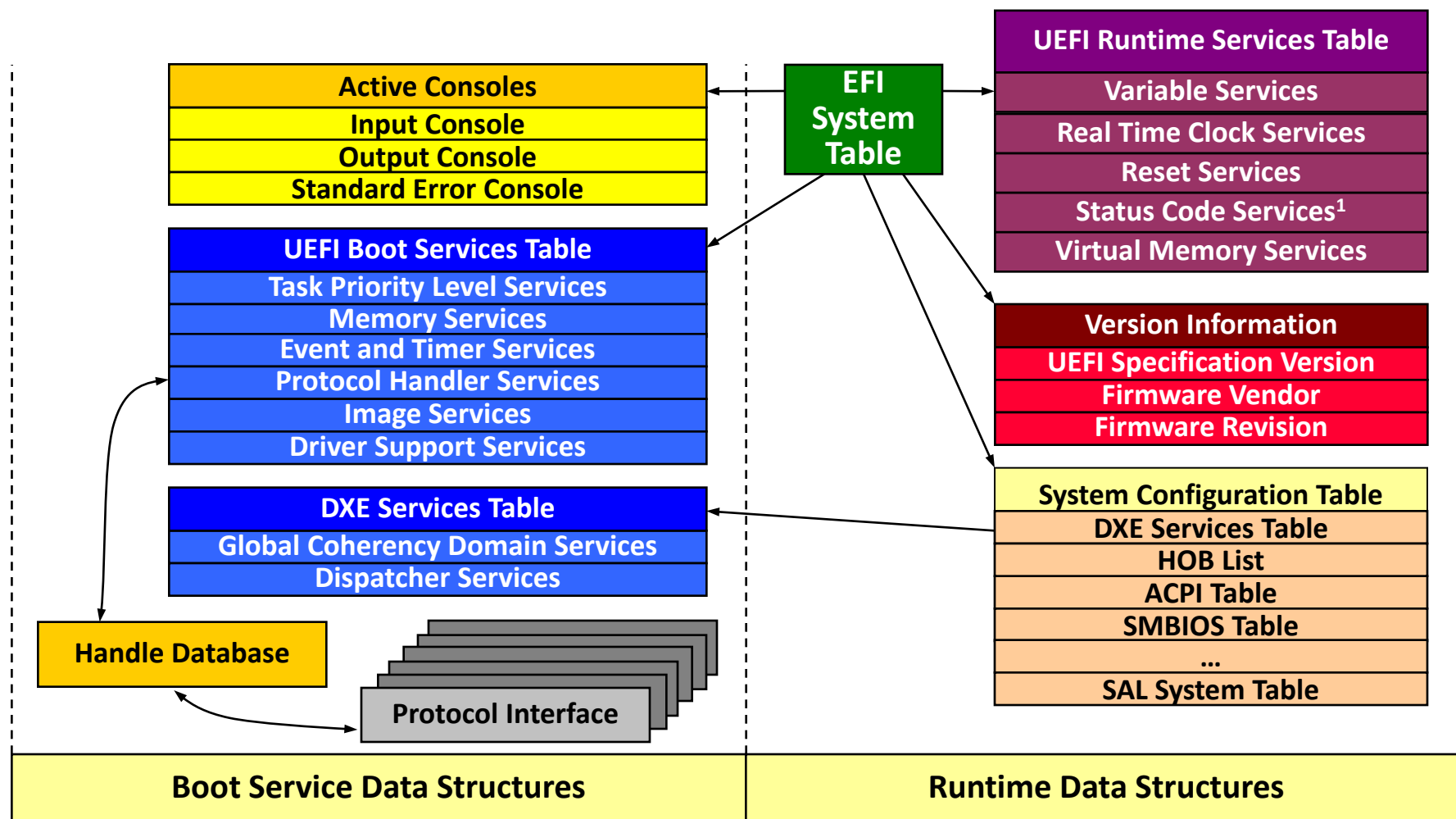
Call: DxeMain

EDK I - \Foundation\Core\Dxe\DxeMain\DxeMain.c
EDK II - \MdeModulePkg\Core\Dxe\DxeMain\DxeMain.c

UEFI System Services

- System services are interfaces that all UEFI compliant systems offer
- Boot Services are a subset are available only before `ExitBootServices()` is called
- Runtime Services are a subset available before and after `ExitBootServices()` is called
- System configuration tables will contain information about industry standard specs and a pointer to the DXE Services. These table remain after `ExitBootServices()`

DXE Foundation Data Structures



¹Status Code Services is part of the DXE CIS § 6.1 Vol 2 PI Spec
Software and Services Group

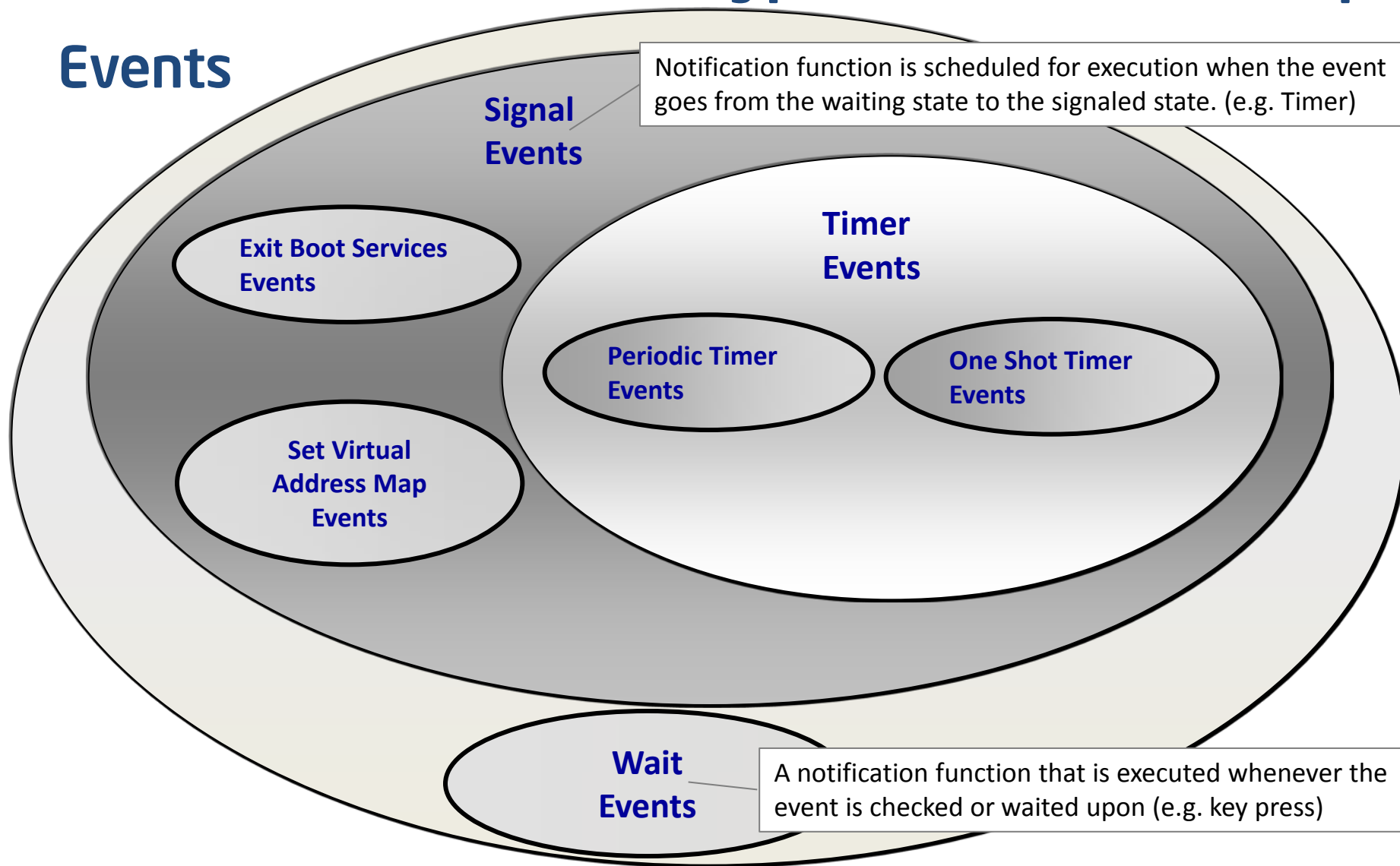


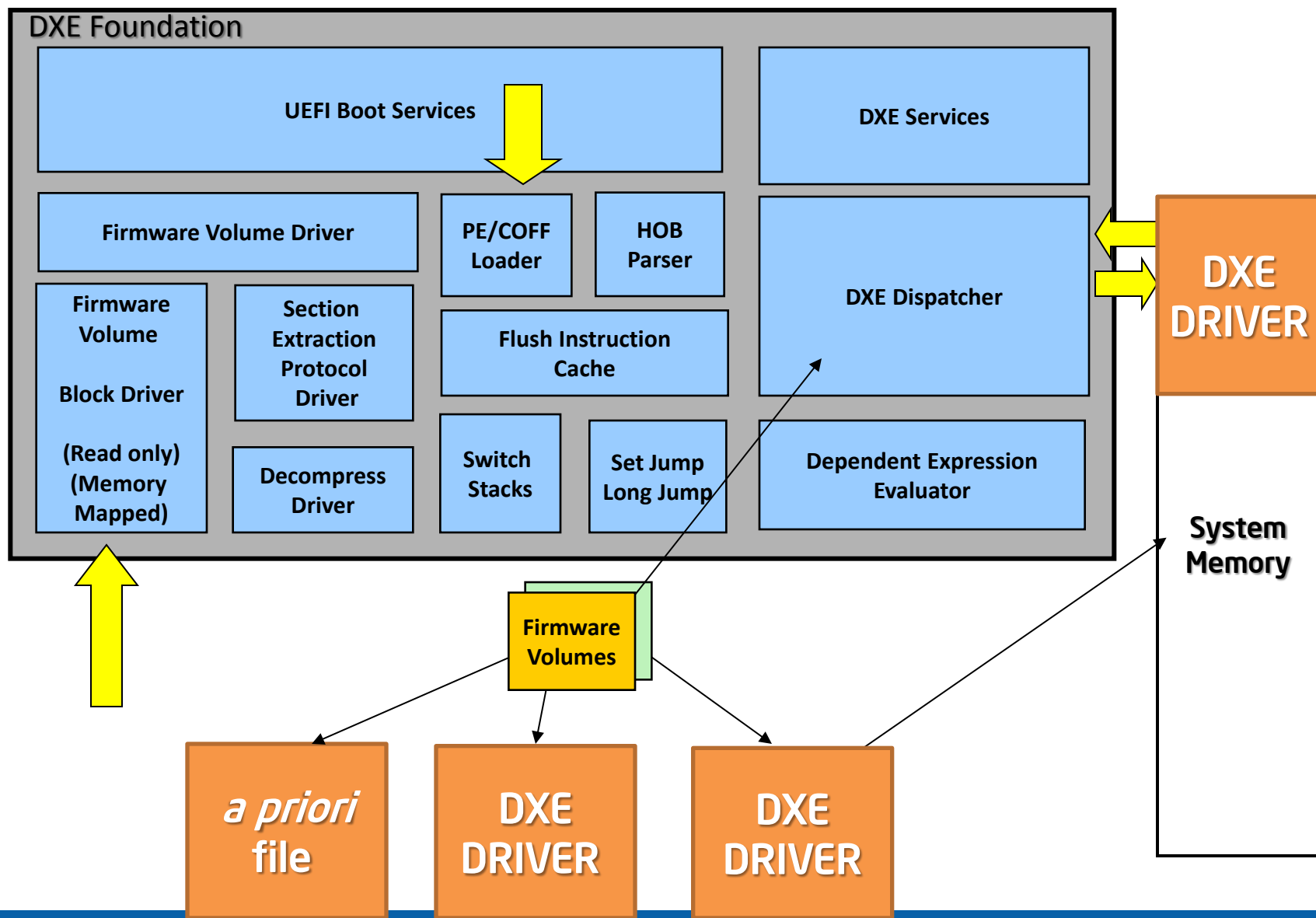
DXE Foundation Control Flow Environment

- Single threaded environment
 - Boot Strap Processor (BSP) controls one code flow
- One software interrupt: Timer tick
 - Only means of asynchronous control transfer
 - Use Events instead
 - 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 Processor architecture specific
 - Hard to model with “good” s/w abstractions
 - Experience: Use UEFI Events instead of interrupts
 - Note that timer flexibility required, e.g. power management, PPP stack serial port flow control
- *DXE Main* is the source code starting point of the DXE Foundation

Event Types and Relationships

Events



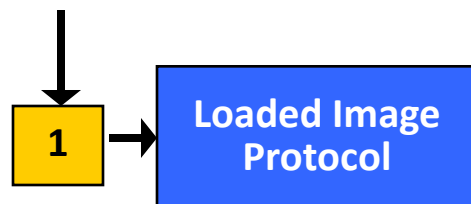


DXE Dispatcher

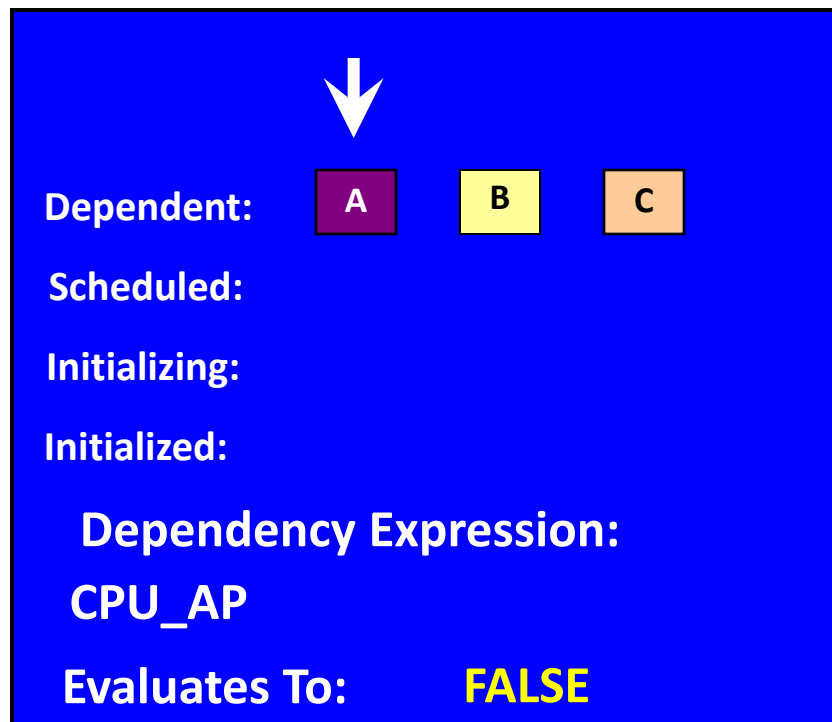
- **DXE Drivers are dispatched first:**
 - A Priori list of drivers are run first
 - Remaining drivers are dispatched according to a list of requirements
 - Architectural Protocols are dispatched and produced
 - Protocols (GUIDs) are “Requirements”
 - “List of Requirements”
 - Boolean Expressions
 - Before/After Ordering Operators
 - Based on state machine (see next slides)
- **UEFI Drivers after DXE Drivers**
 - 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)

DXE Foundation Dispatcher

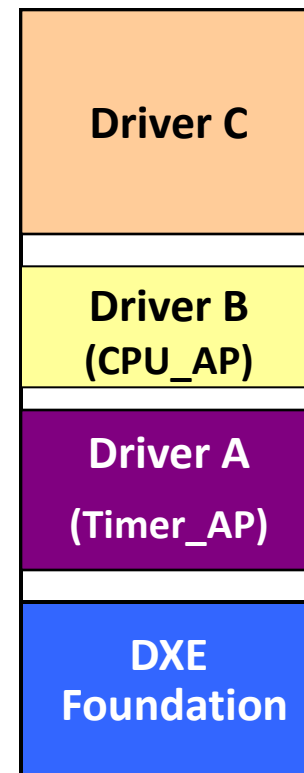
Handle Database



DXE Dispatcher

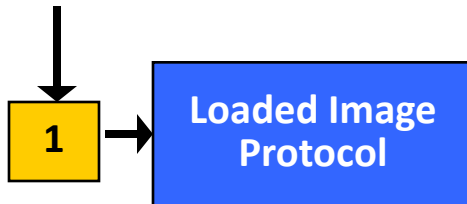


Main Firmware Volume

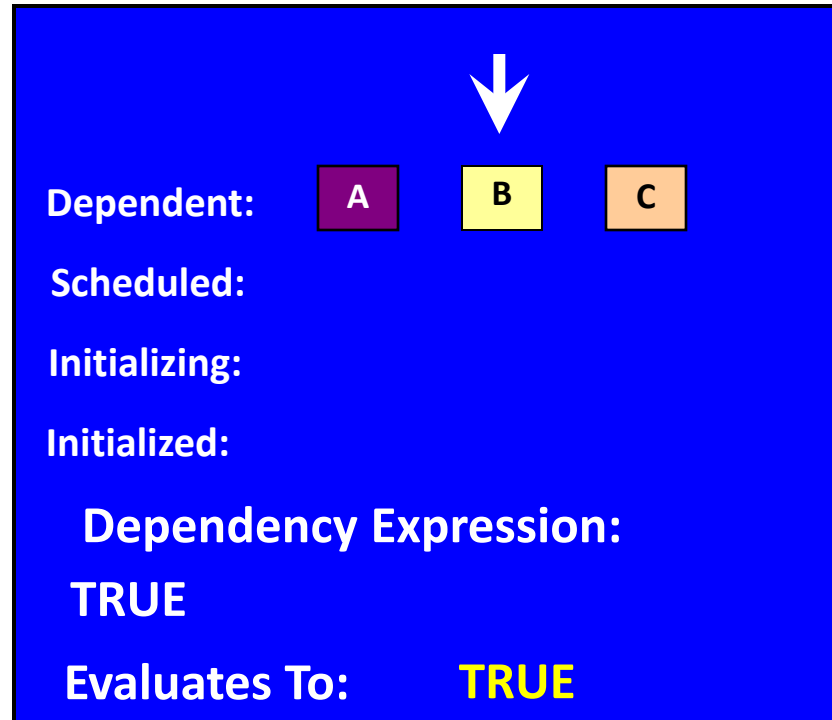


DXE Foundation Dispatcher

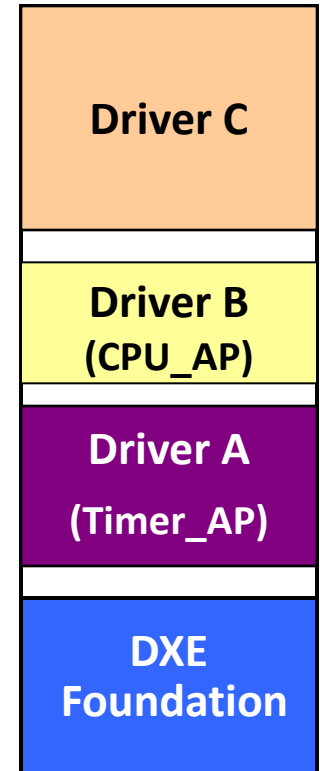
Handle Database



DXE Dispatcher

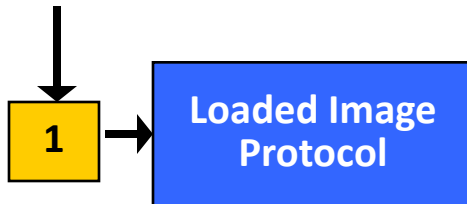


Main Firmware Volume

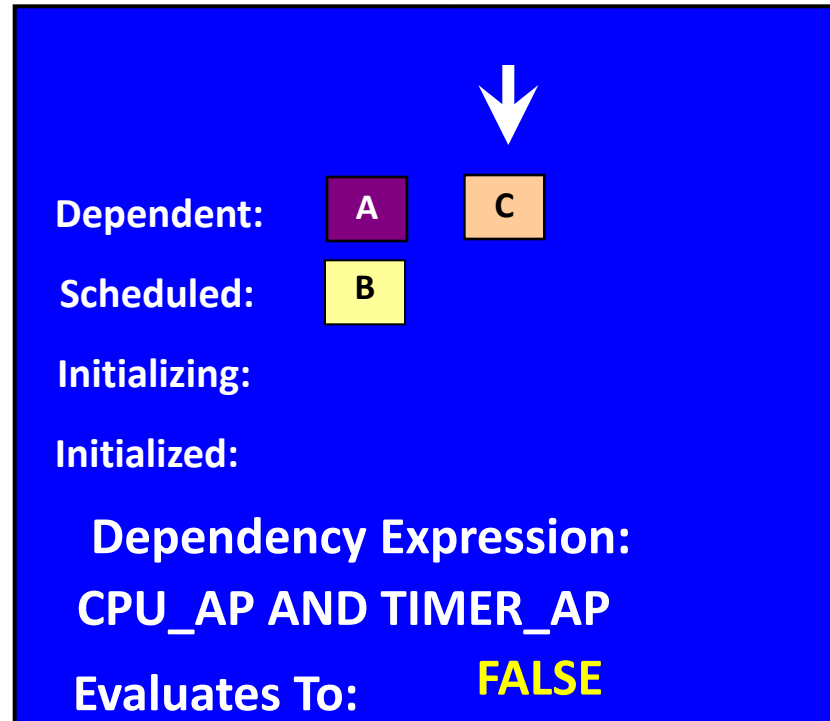


DXE Foundation Dispatcher

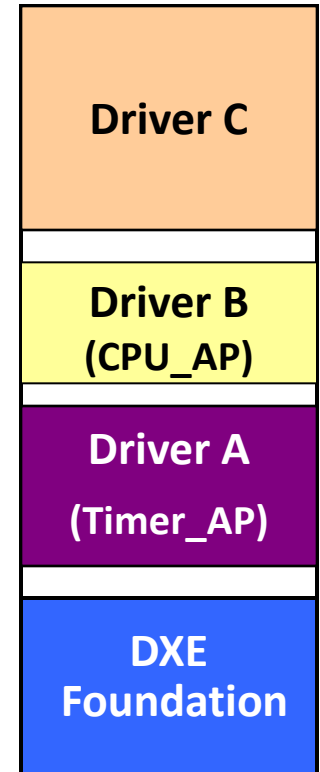
Handle Database



DXE Dispatcher

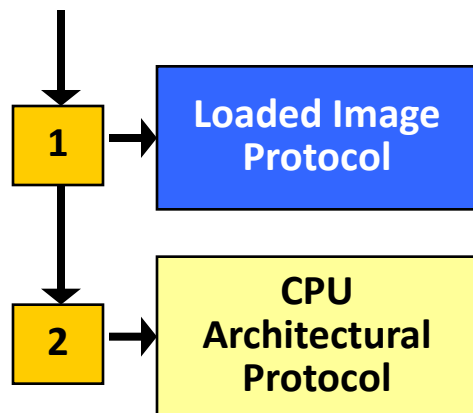


Main Firmware Volume

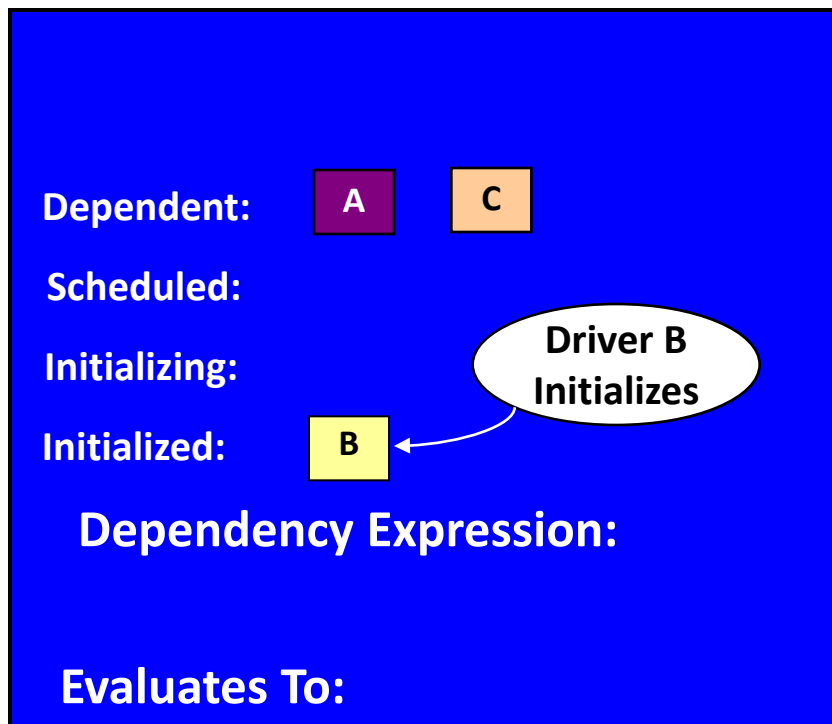


DXE Foundation Dispatcher

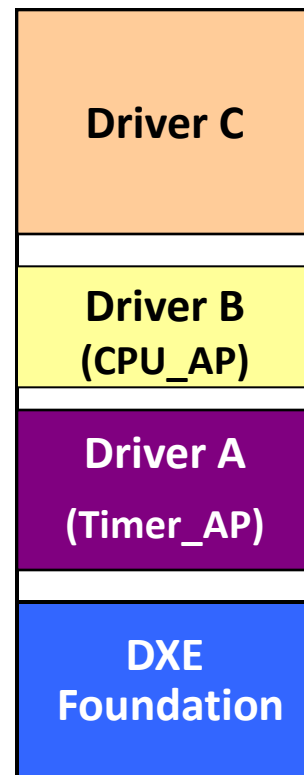
Handle Database



DXE Dispatcher

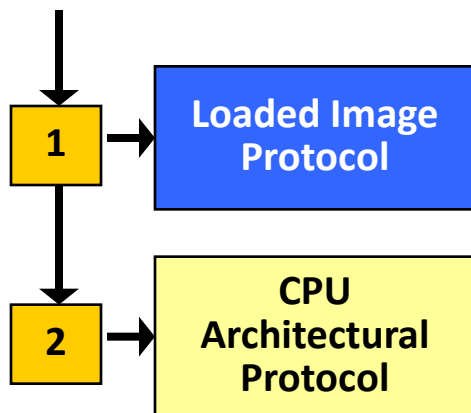


Main Firmware Volume

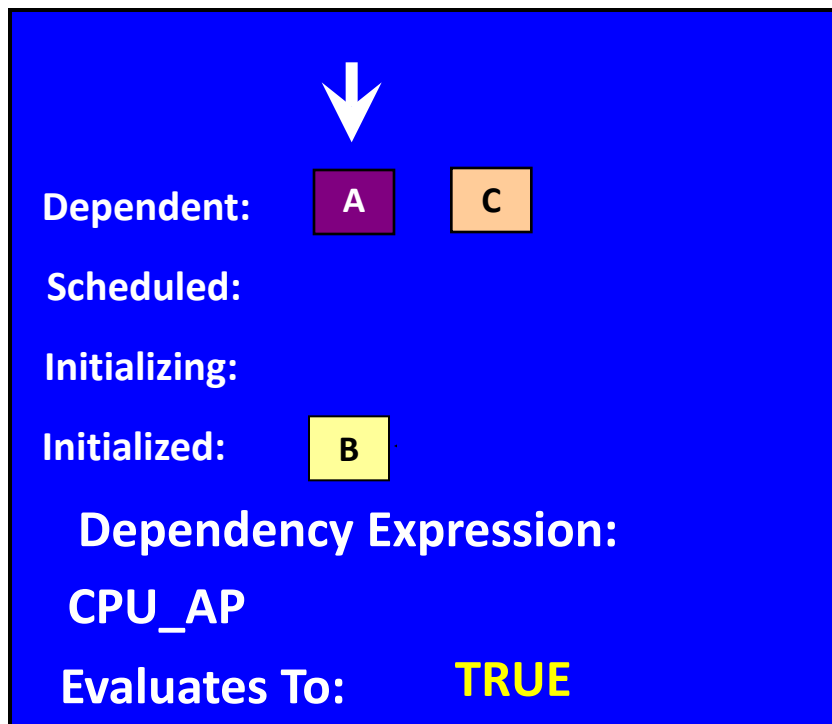


DXE Foundation Dispatcher

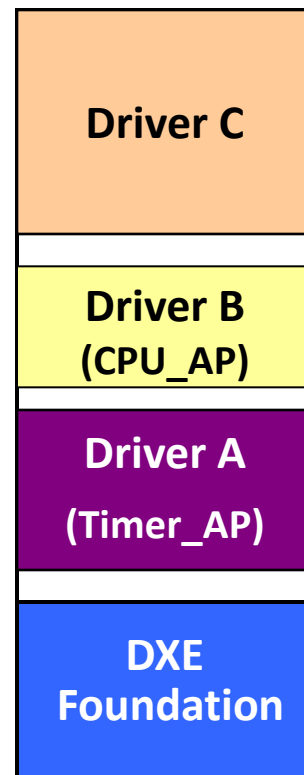
Handle Database



DXE Dispatcher

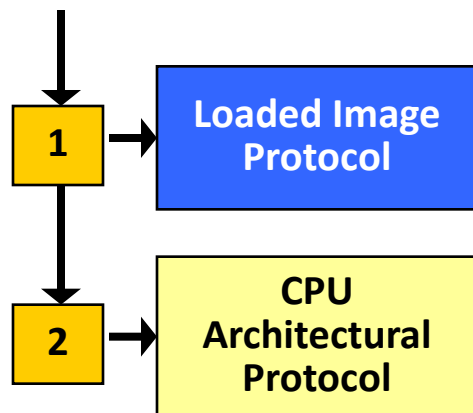


Main Firmware Volume

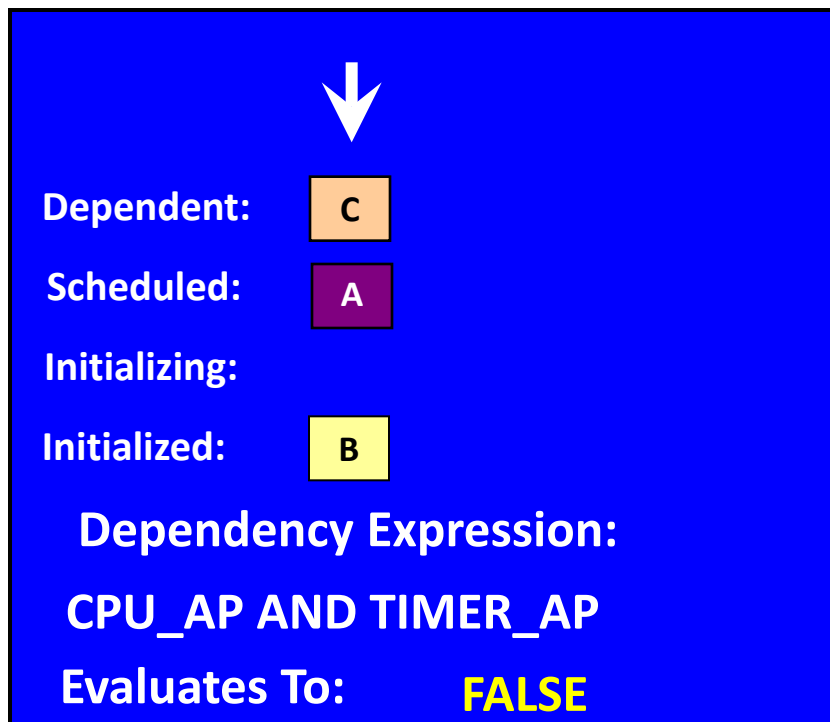


DXE Foundation Dispatcher

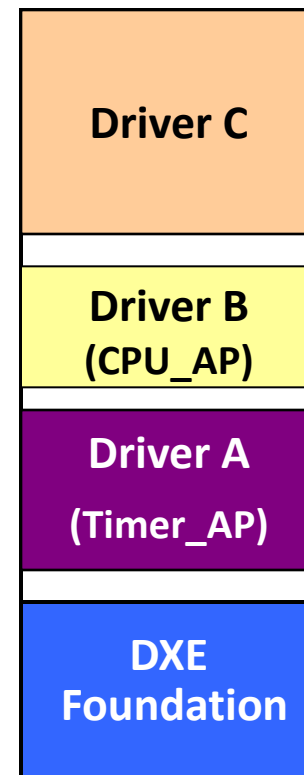
Handle Database



DXE Dispatcher

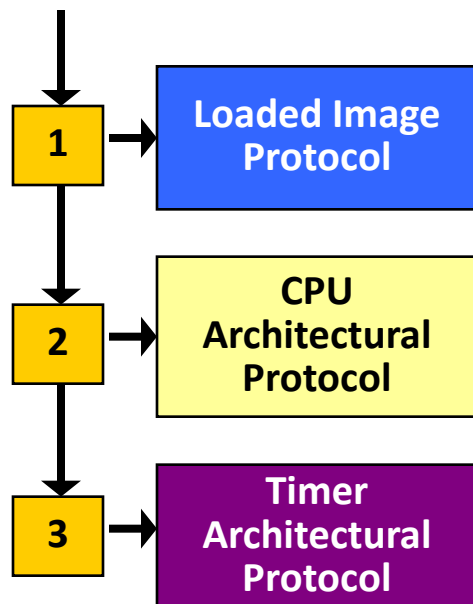


Main Firmware Volume

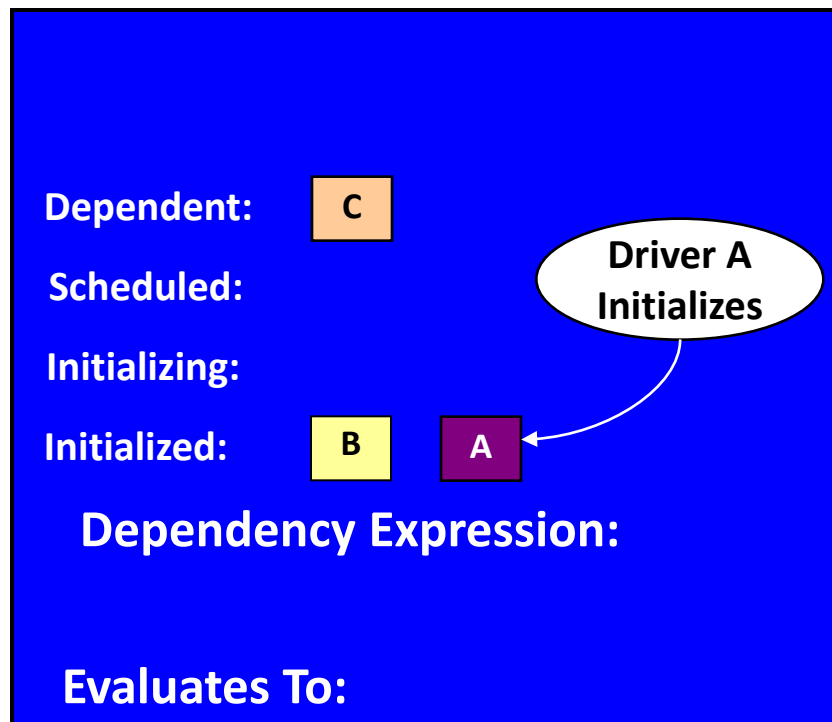


DXE Foundation Dispatcher

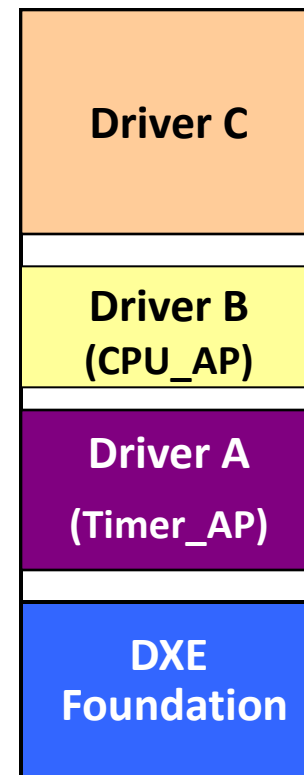
Handle Database



DXE Dispatcher

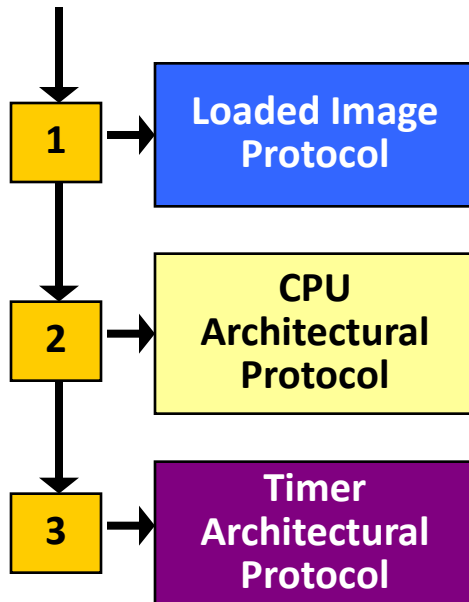


Main Firmware Volume

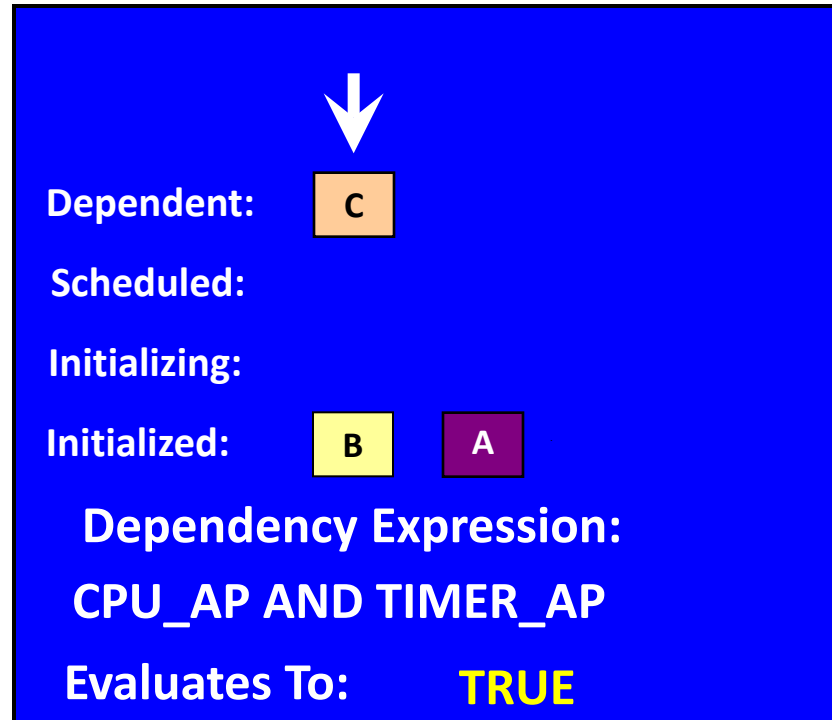


DXE Foundation Dispatcher

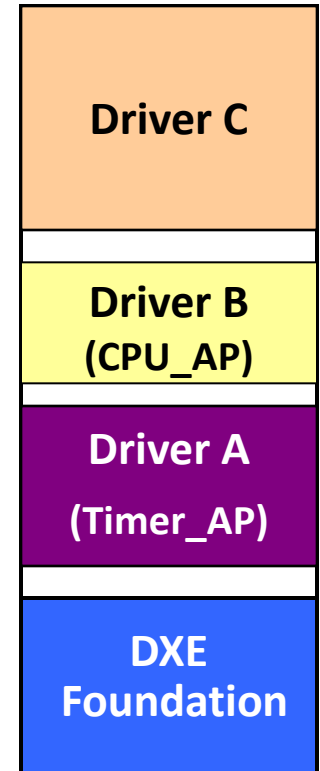
Handle Database



DXE Dispatcher

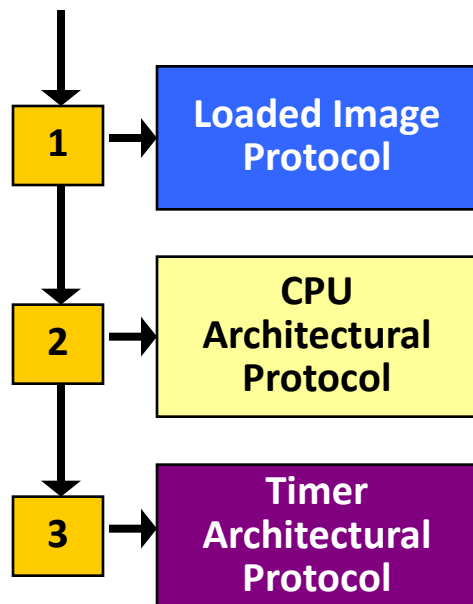


Main Firmware Volume

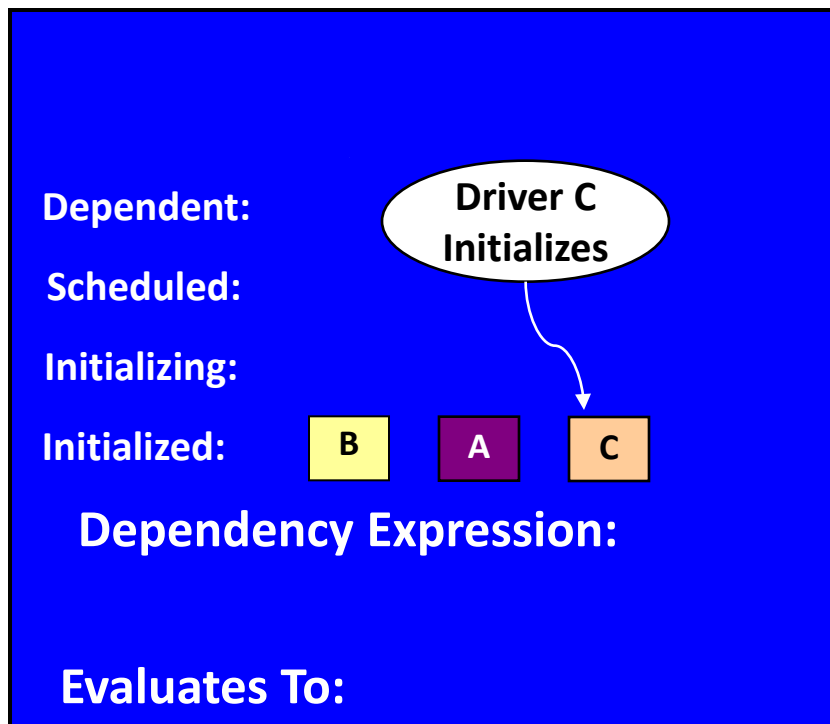


DXE Foundation Dispatcher

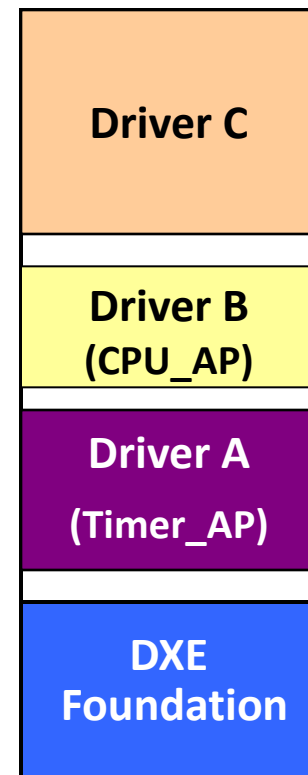
Handle Database



DXE Dispatcher



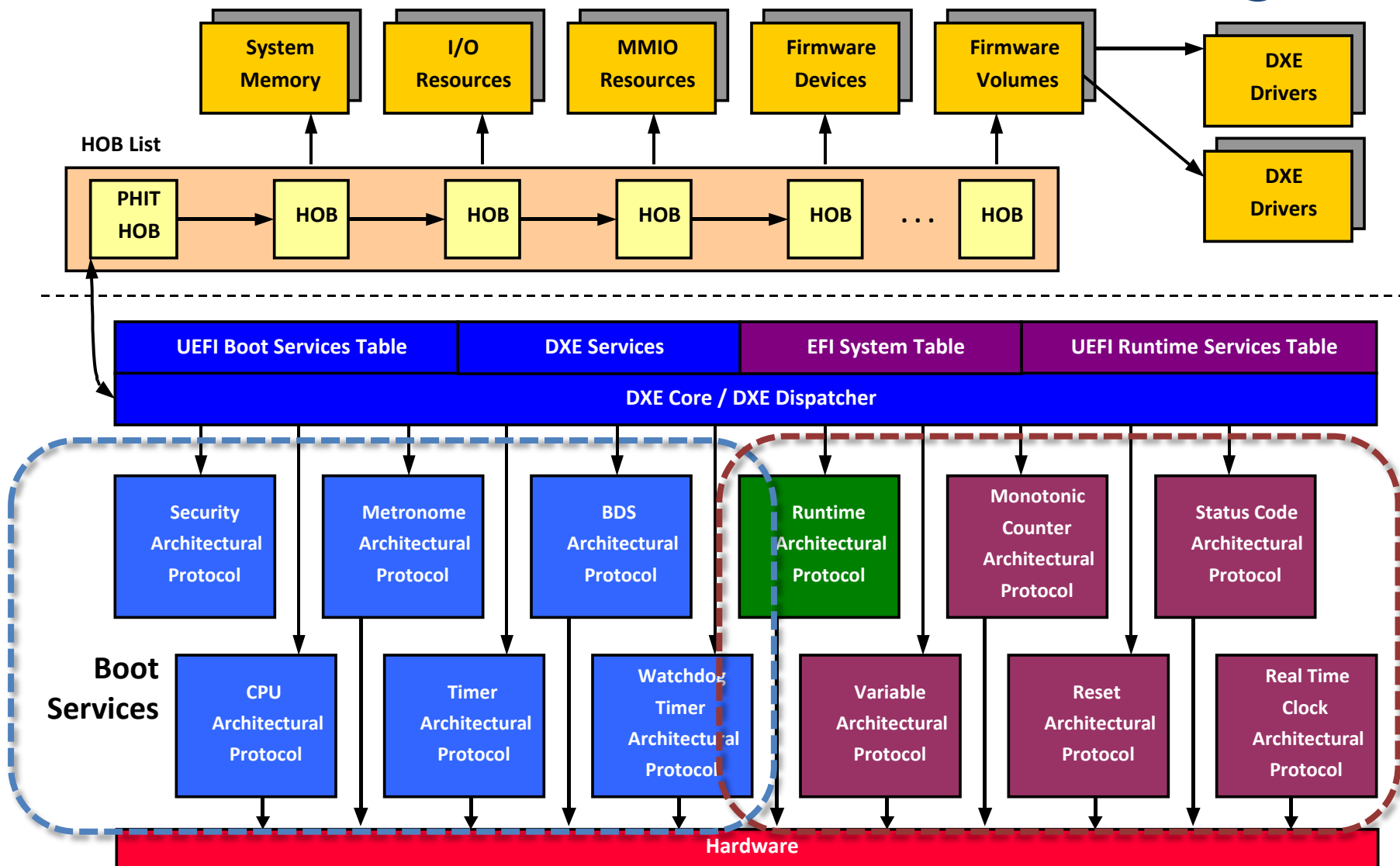
Main Firmware Volume



Execution Order Determined at Runtime Based on Dependencies



DXE Core Block Diagram



See § 12 of PI Spec Vol. 2
Software and Services Group



DXE – Architectural Protocols

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

Location of Architectural Protocols in EDK II

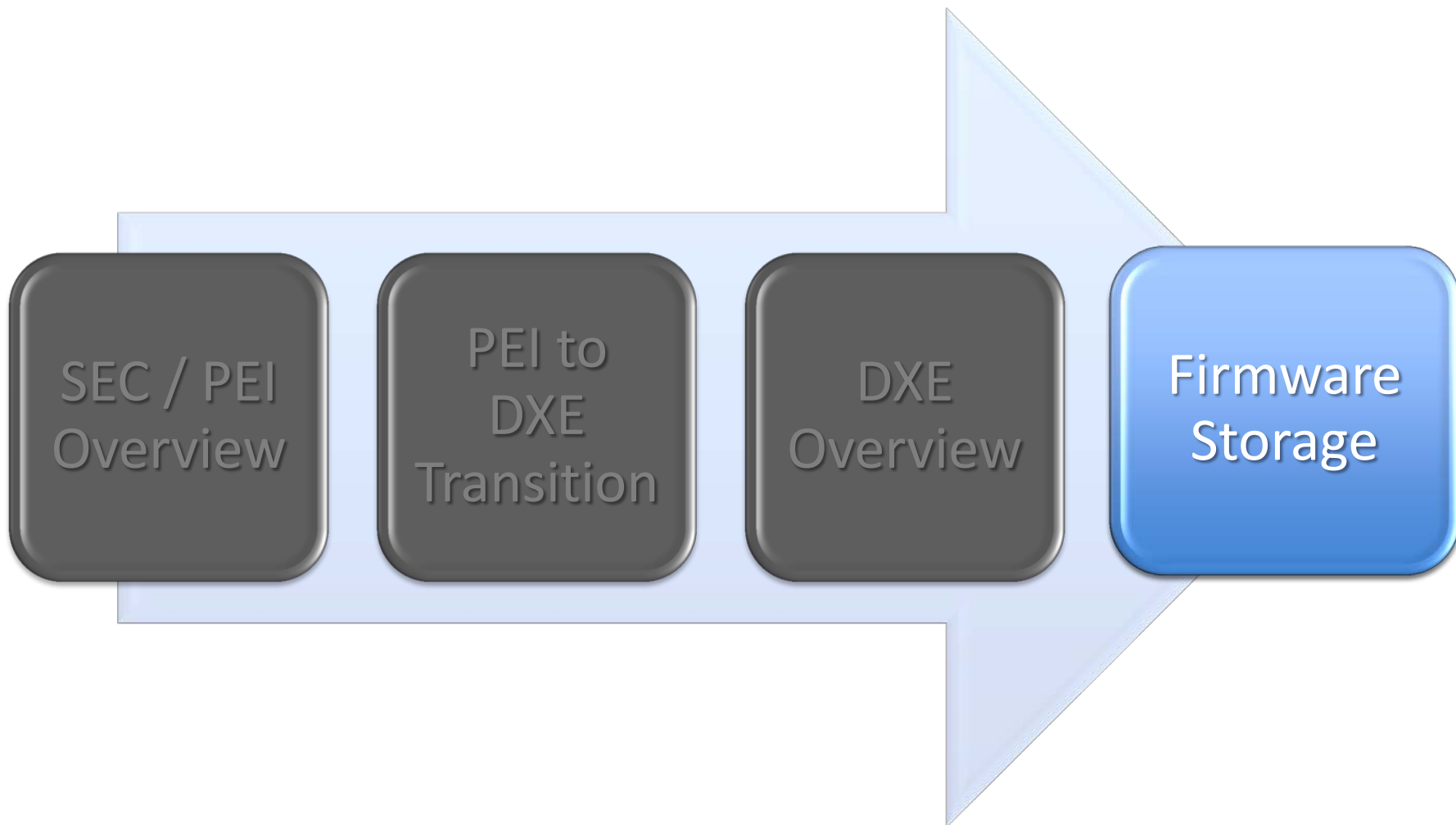
gEfiBdsArchProtocolGuid	IntelFrameworkModulePkg\Universal\BdsDxe\
gEfiCapsuleArchProtocolGuid	MdeModulePkg\Universal\CapsuleRuntimeDxe
gEfiCpuArchProtocolGuid ¹	IA32FamilyCpuPkg\CpuArchDxe *
gEfiMetronomeArchProtocolGuid	MdeModulePkg\Universal\MetronomeDxe
gEfiMonotonicCounterArchProtocolGuid	MdeModulePkg\Universal\MonotonicCounterRuntimeDxe
gEfiRealTimeClockArchProtocolGuid ¹	IbexpeakPchPkg\RtcDxe *
gEfiResetArchProtocolGuid	MdeModulePkg\Universal\ResetSystemRuntimeDxe
gEfiRuntimeArchProtocolGuid	MdeModulePkg\Core\RuntimeDxe
gEfiSecurityArchProtocolGuid	SecurityPkg\Tcg\TcgDxe
gEfiStatusCodeRuntimeProtocolGuid	IntelFrameworkModulePkg\Universal\StatusCode\Dxe
gEfiTimerArchProtocolGuid ¹	IbexpeakPchPkg\SmartTimerDxe\ *
gEfiVariableArchProtocolGuid	MdeModulePkg\Universal\Variable
gEfiVariableWriteArchProtocolGuid	MdeModulePkg\Universal\Variable\RuntimeDxe
gEfiWatchdogTimerArchProtocolGuid	MdeModulePkg\Universal\WatchdogTimerDxe

¹ Example from Intel® DQ57TM Desktop Board project

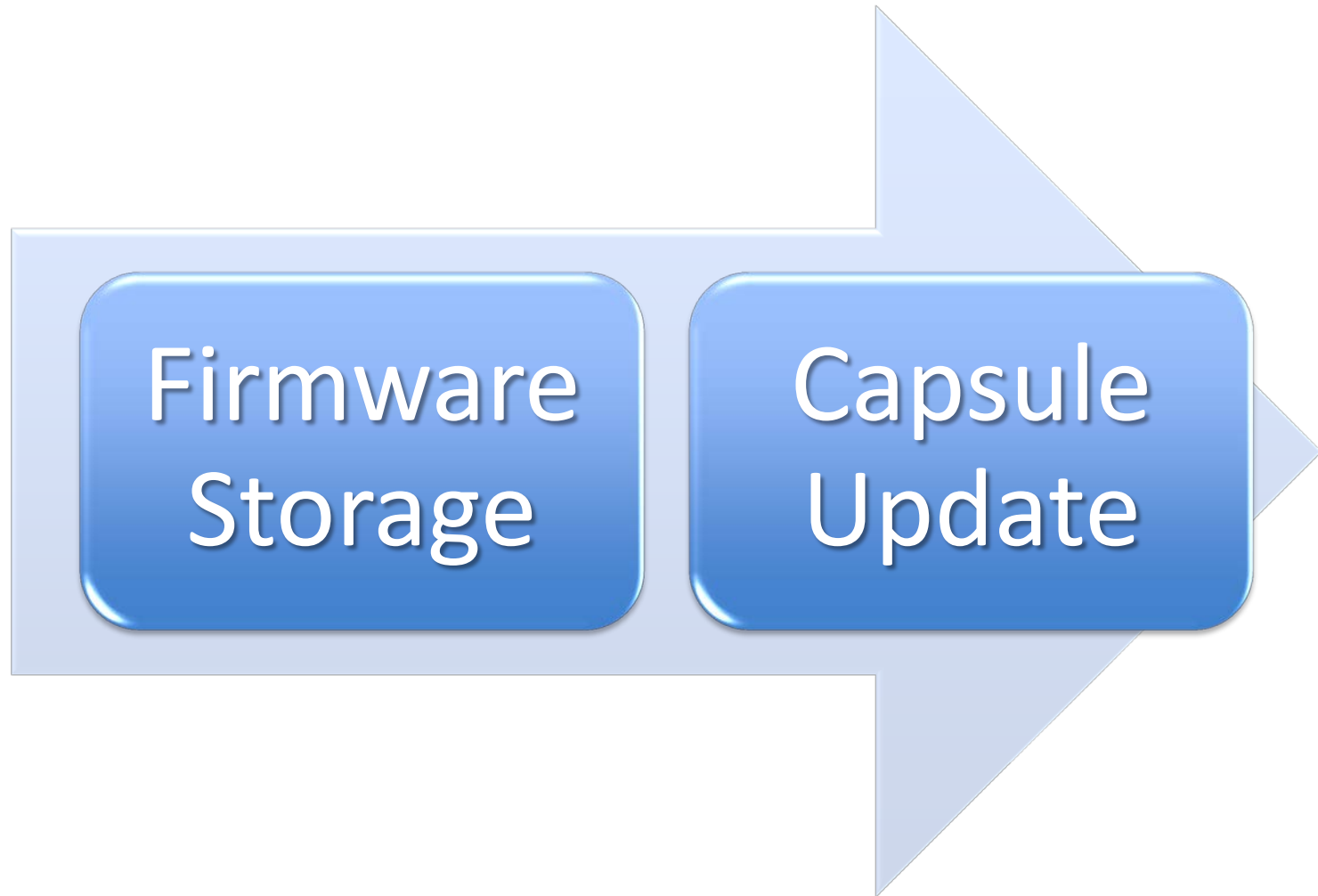


- Non Open Source

Agenda



Agenda

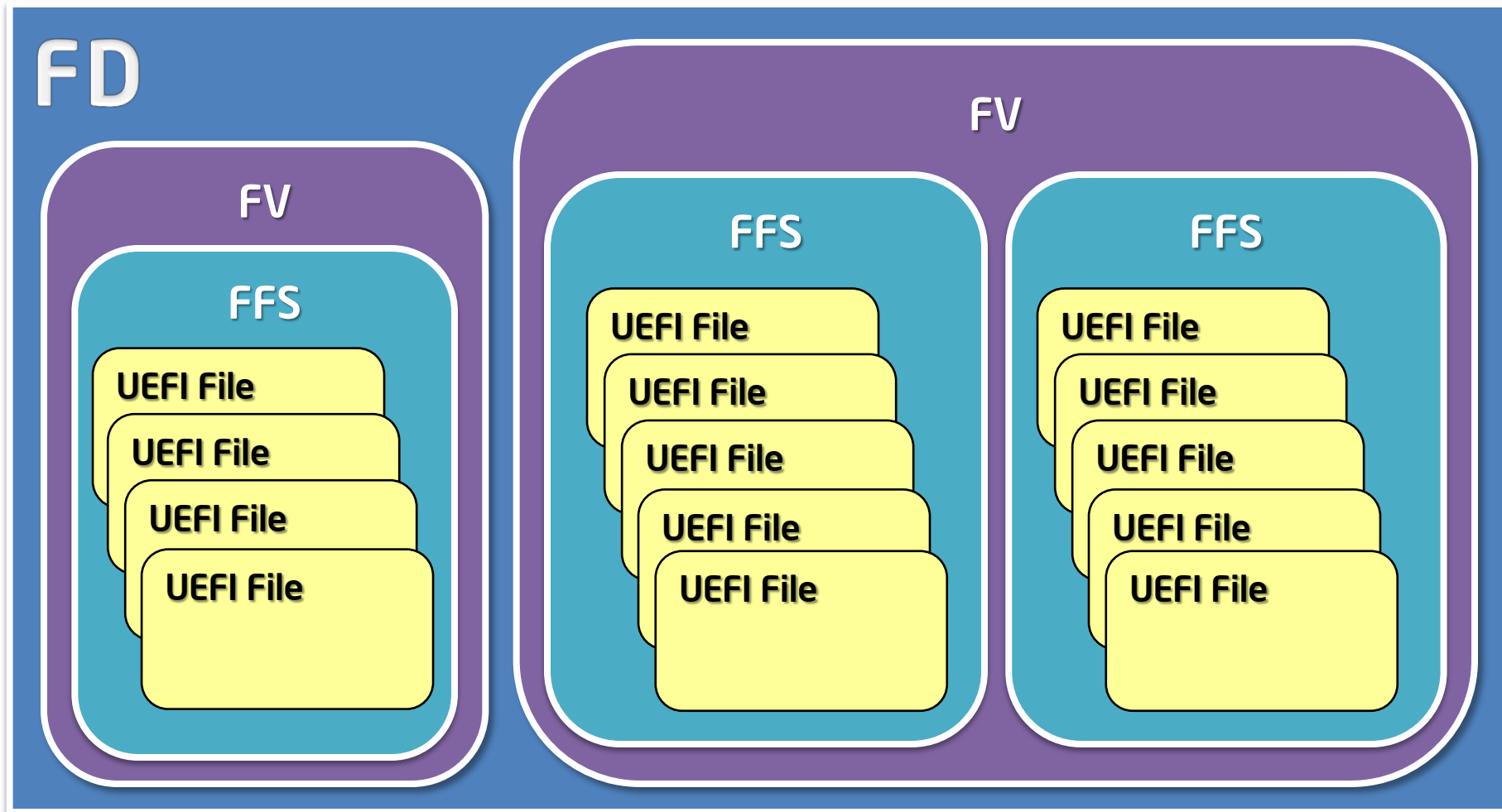




Firmware Storage - Overview

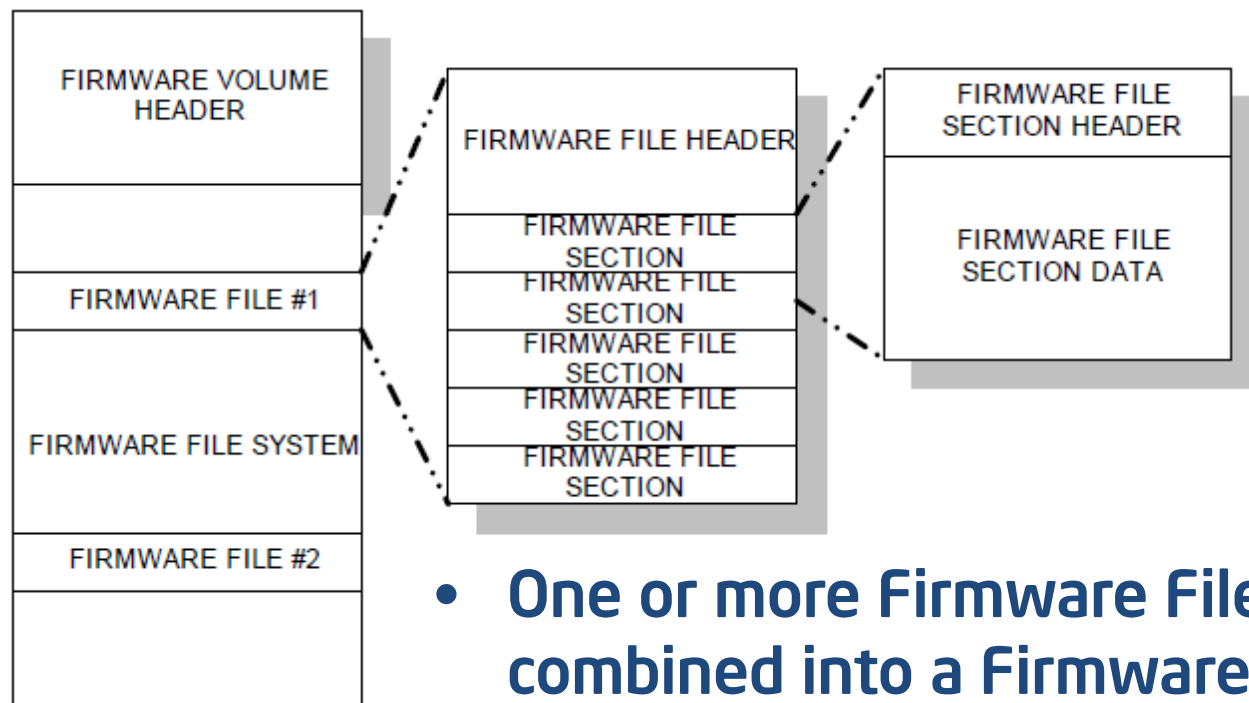
- **Platform Initialization Firmware Volume**
 - Basic storage repository for data and code is the FV
 - Each FV is organized into a file system, each with attributes
- **Firmware File System format**
 - A firmware file system (FFS) describes the organization of files and free space within the firmware volume.
 - Each firmware file system has a unique GUID, which is used by the firmware to associate a driver with a newly exposed FV
- **Firmware Files**
 - Code and data stored in firmware volumes
 - Each of the files has the following attributes
 - Name, Type, Alignment, Size

Firmware Device/FFS/FV and UEFI File Relationship



Firmware Volumes

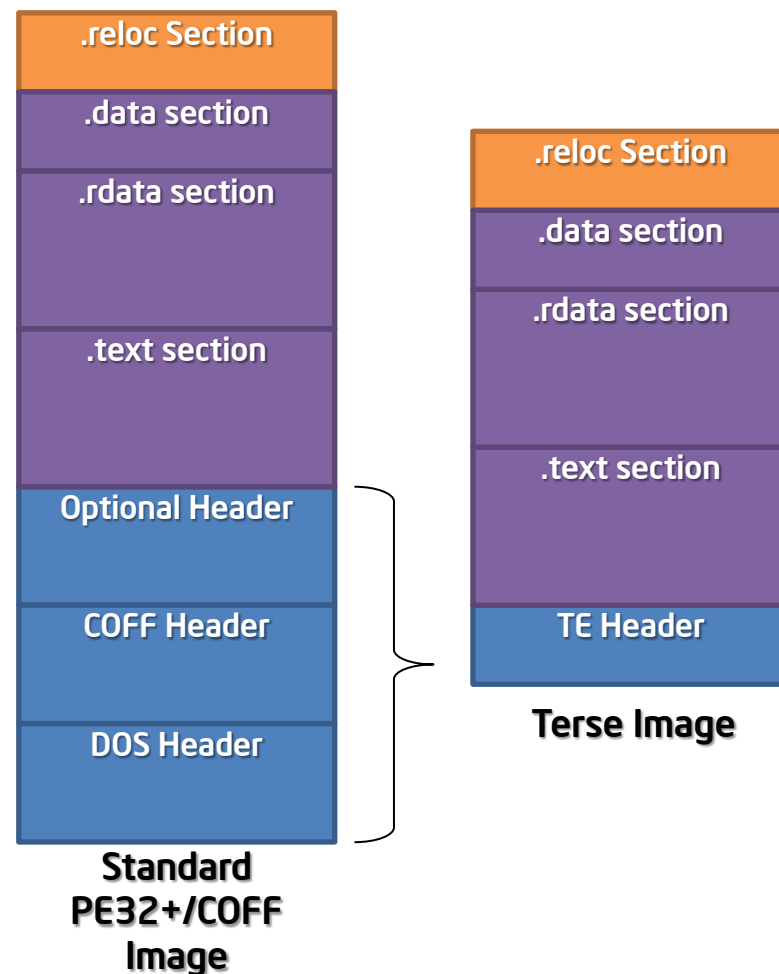
FV



- One or more Firmware File Sections files are combined into a Firmware Volume (FV.)
- The format for a an FV is a header followed by an optional extended header, followed by zero or more Firmware File Sections

Creating UEFI Firmware File Images

- UEFI code is compiled
- Dynamic linker generates the relocatable image
- Image is formatted as PE32/PE32+/COFF
- Standard header is replaced with UEFI format or Terse Image (TE) header



Flash Device Configuration

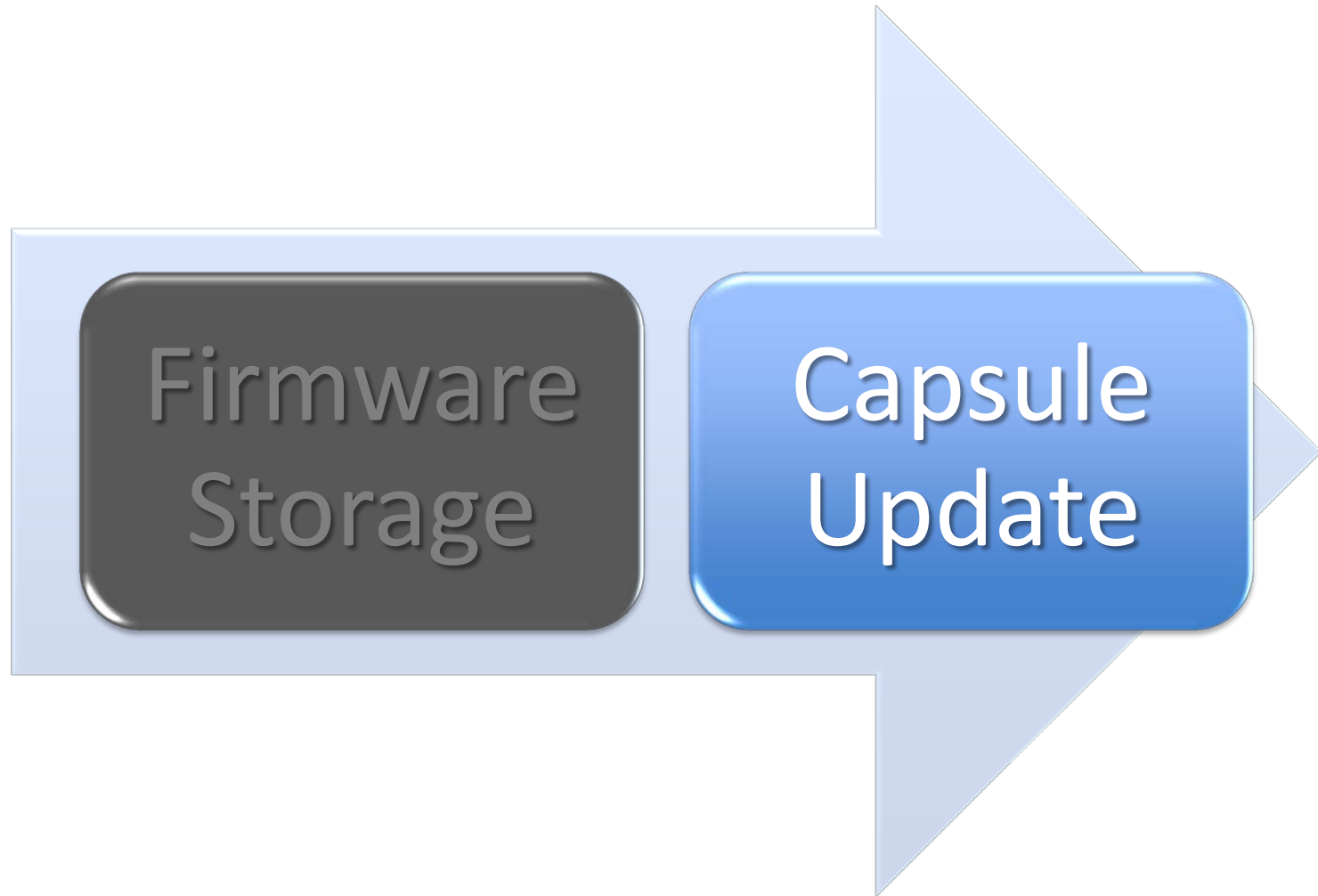
FV Recovery	Used to store SEC/PEI phase code
FTW spare space	Fault Tolerant Write (FTW) regions
FTW working space	
Event Log	NVRAM storage for event logs
Microcode	CPU Microcode
Variable Region	Variables & platform settings
FV Main	Contains DXE phase drivers

Demo

- Loading Drivers from non-FLASH locations



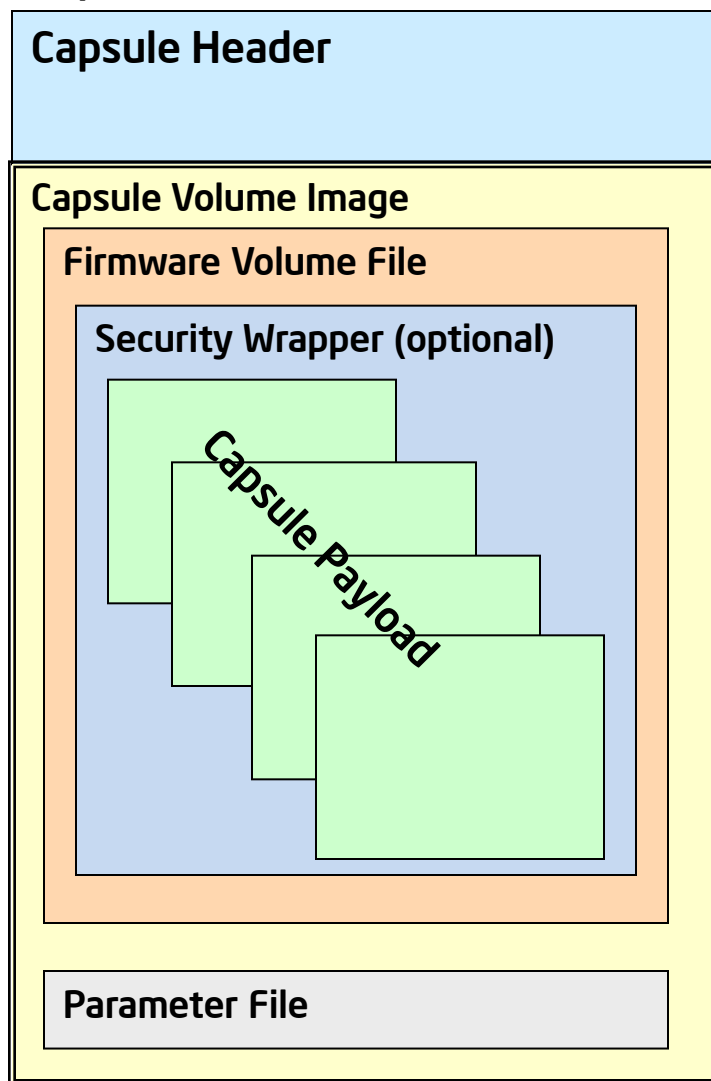
Agenda



Capsules

- Capsules allow the OS to place DXE drivers in memory to be executed at the next reboot
 - Examples: firmware update, firmware recovery
 - Other non-architectural mechanisms to support update and recovery may be provided as value-add
- EFI Capsule Architectural Protocol in the PI Spec
 - Provides the DXE RT services for capsule update
See § 12.13 Vol. 2 PI Spec
 - EFI PEI Recovery Module PPI
See § 8.3.3 Vol. 1 PI Spec

Capsule



Capsule Structure

Header - used by OS present application, not loaded into RAM

Capsule volume ("CV") Image - firmware volume with a different type GUID

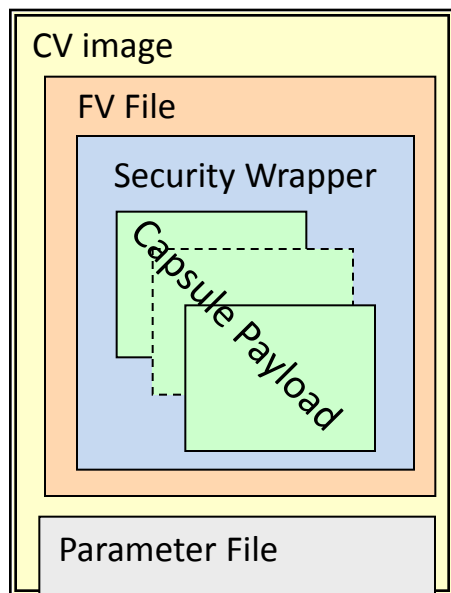
Firmware Volume ("FV") file: a firmware volume stored as a single file

Security 'wrapper' - for example, a digital signature of the contents to be validated while processing the CV

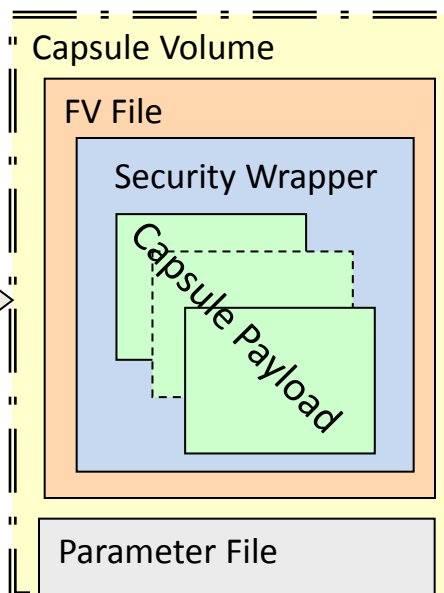
"Capsule Payload": Various files. Can include executable DXE drivers, images, etc. In the case of update, might include both the image to update and the drivers to perform the update

Parameter file - file generated by Setup, path data, etc.

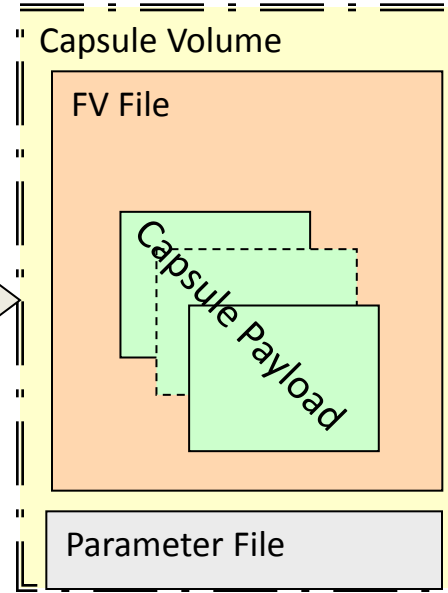
1. Load CV image into RAM



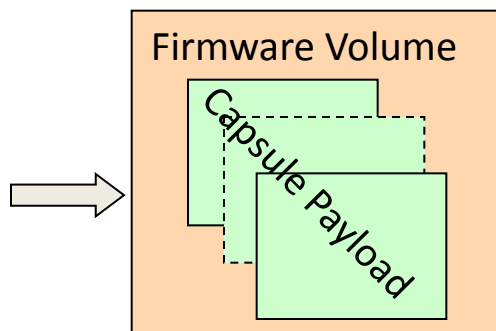
2. Make capsule volume from CV image



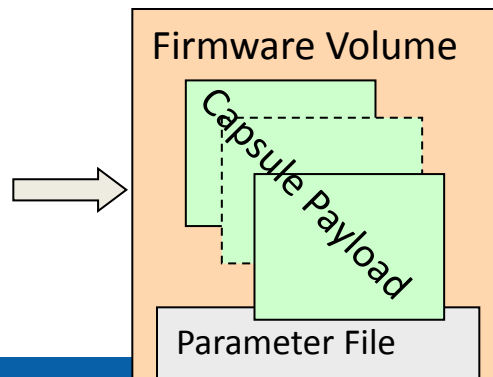
3. Validate security



4. Create new RAM Firmware Volume from FV File



5. Write non-executable (non-viral) parameter files



The security wrapper is optional. An alternative, which would preserve or even add security, would be to sign each driver. Security requirements are defined by platform policy.

At #4, the dispatcher will scan the files in the new FV, discovering drivers to schedule.



Summary

- **Modular Code without memory**
- **Easier Silicon Initialization**
 - Reference code would just work
- **Code from multiple vendors can coexist**
- **Modularity makes porting easier**
- **Complex Chipset initialization in C**
 - Code easier to maintain
- **DXE can be totally relocatable and Hardware Independent**

Q & A





Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2[®], SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804