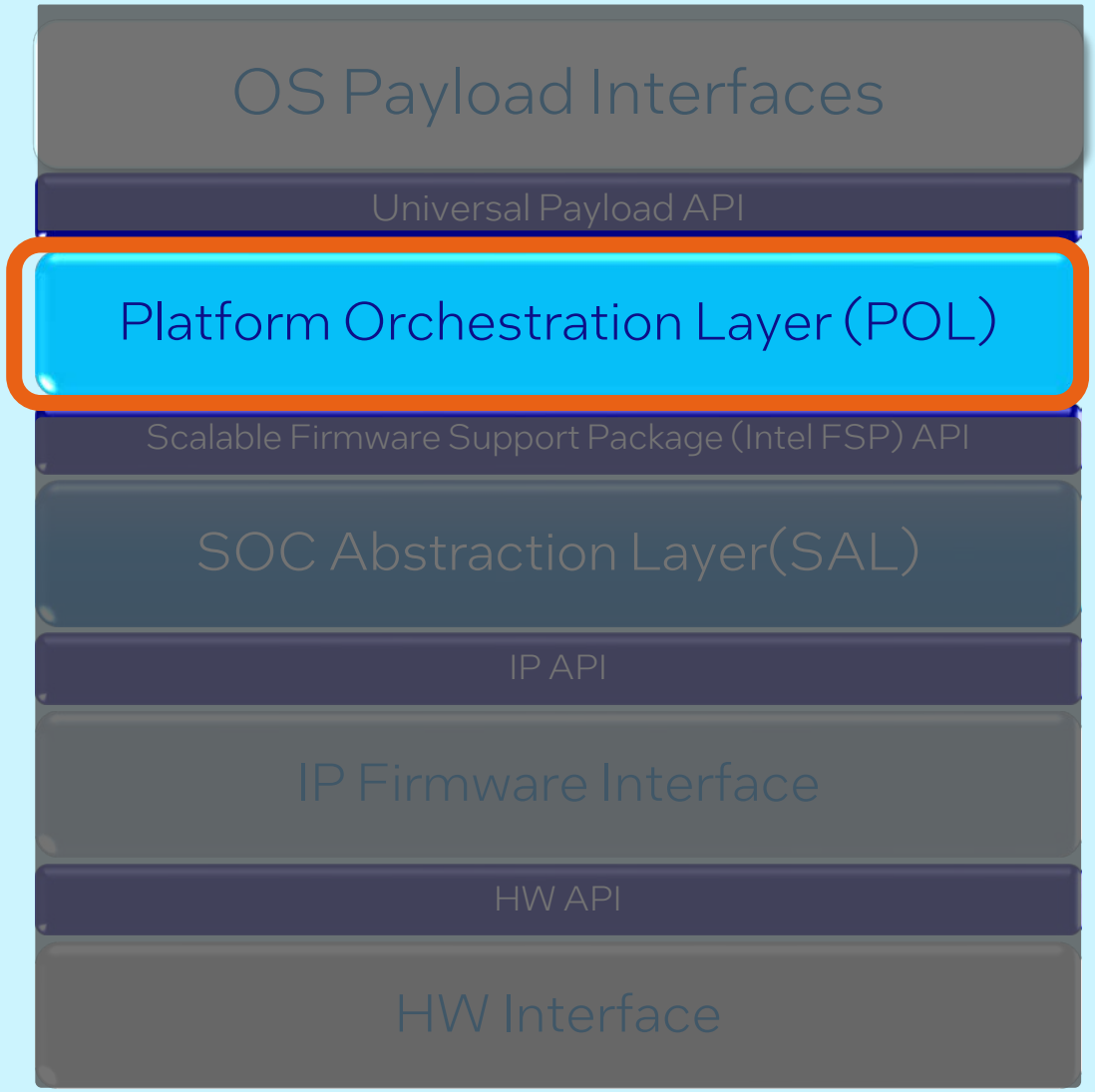
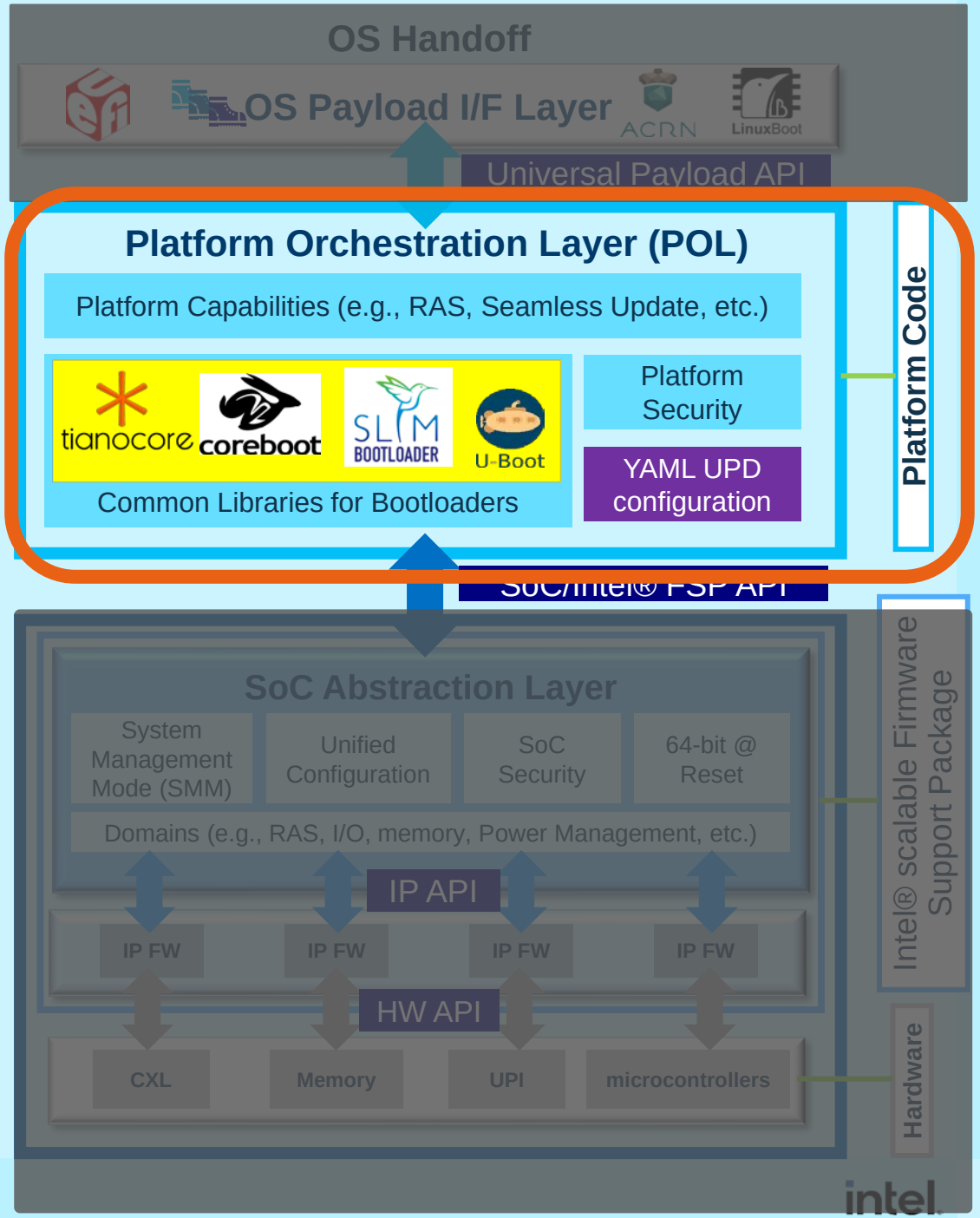
System Firmware Training

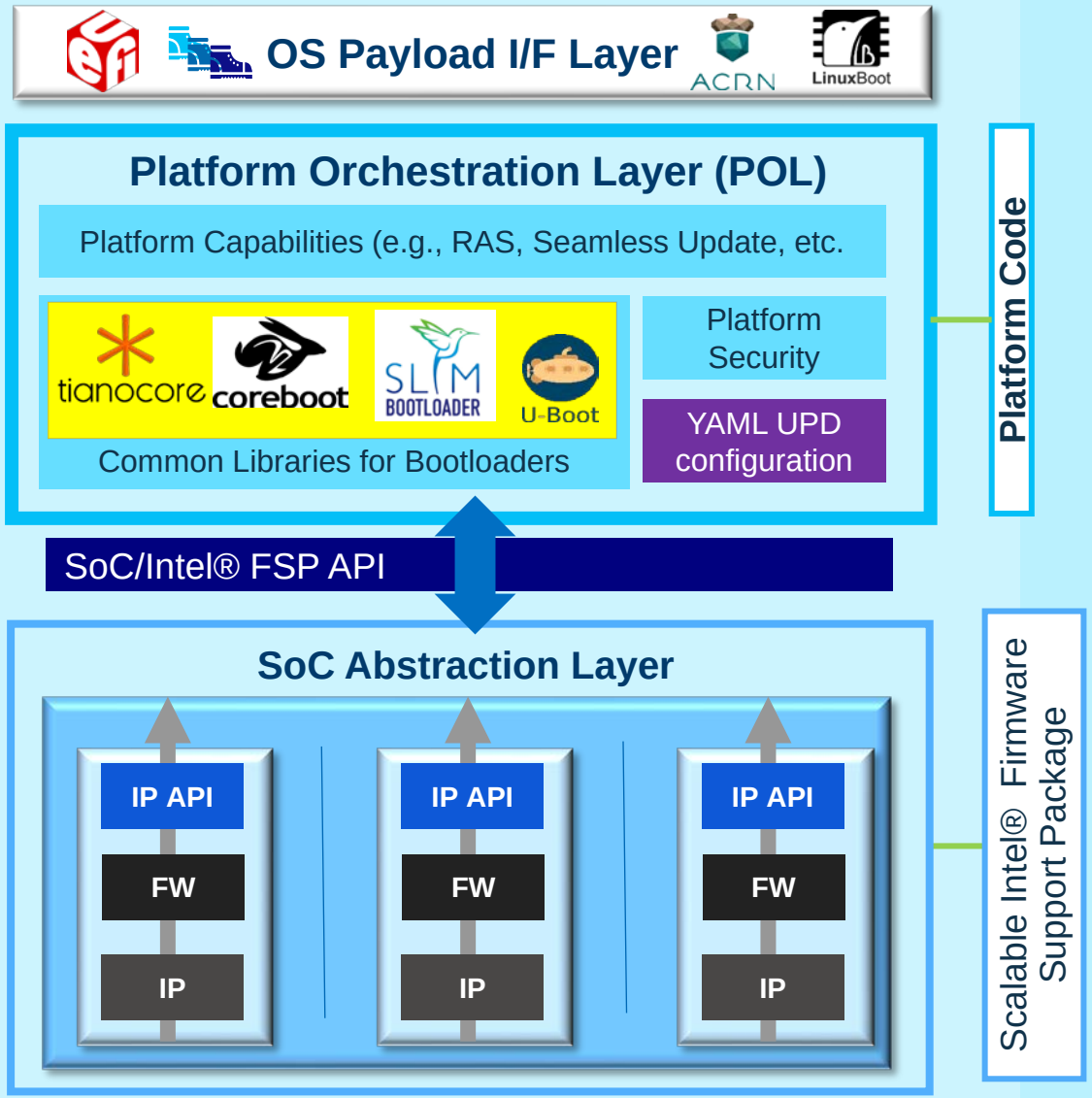
Universal Scalable Firmware (USF) :
Platform Orchestration Layer

Intel Corporation

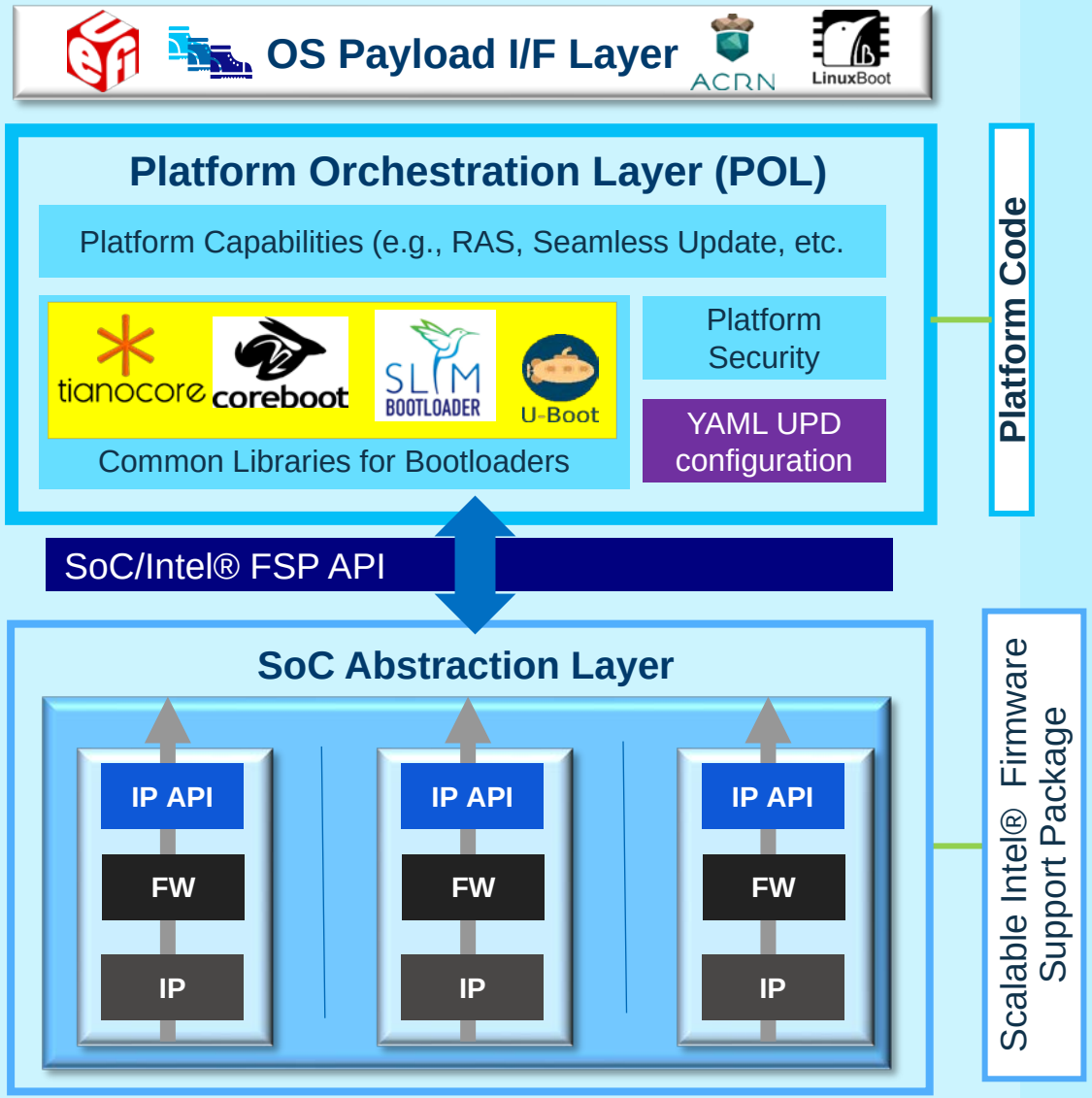

OS Payload Interfaces
Universal Payload API
Platform Orchestration Layer (POL)
Scalable Firmware Support Package (Intel FSP) API
SOC Abstraction Layer(SAL)
IP API
IP Firmware Interface
HW API
HW Interface


OS Handoff
OS Payload I/F Layer
ACRN
LinuxBoot
Universal Payload API
Platform Orchestration Layer (POL)
Platform Capabilities (e.g., RAS, Seamless Update, etc.)
tianocore
coreboot
SLIM BOOTLOADER
U-Boot
Common Libraries for Bootloaders
Platform Security
YAML UPD configuration
Platform Code
SoC/Intel® FSP API
SoC Abstraction Layer
System Management Mode (SMM)
Unified Configuration
SoC Security
64-bit @ Reset
Domains (e.g., RAS, I/O, memory, Power Management, etc.)
IP API
IP FW
IP FW
IP FW
IP FW
HW API
CXL
Memory
UPI
microcontrollers
Intel® scalable Firmware Support Package
Hardware

Platform Orchestration Layer (POL) - Overview



Platform Orchestration Layer (POL) - Overview

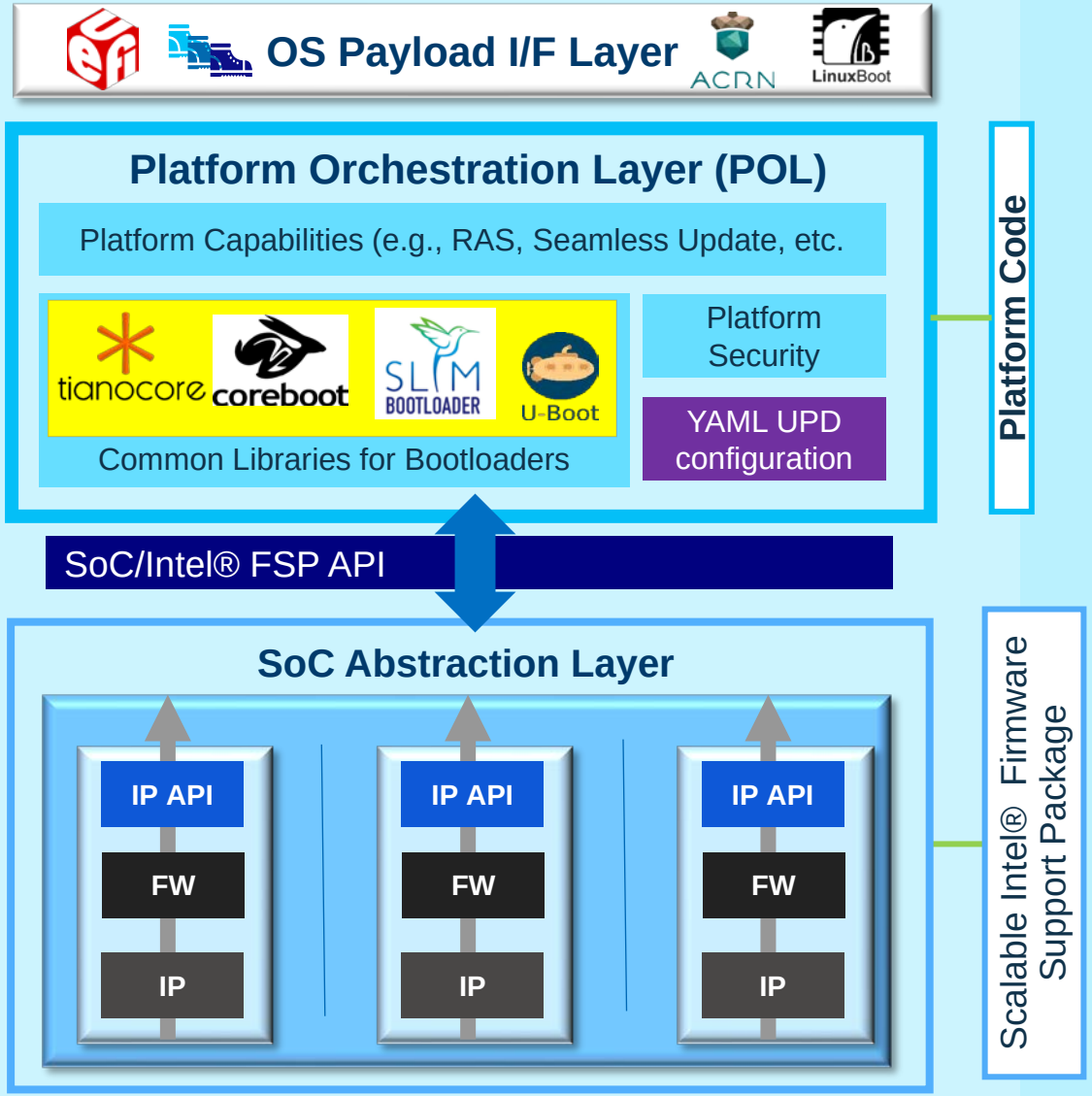


intel.

Platform Orchestration Layer (POL) - Overview

Design Principles

Providing vendor specific features and mainboard specific initialization



intel.

Platform Orchestration Layer (POL) - Architecture

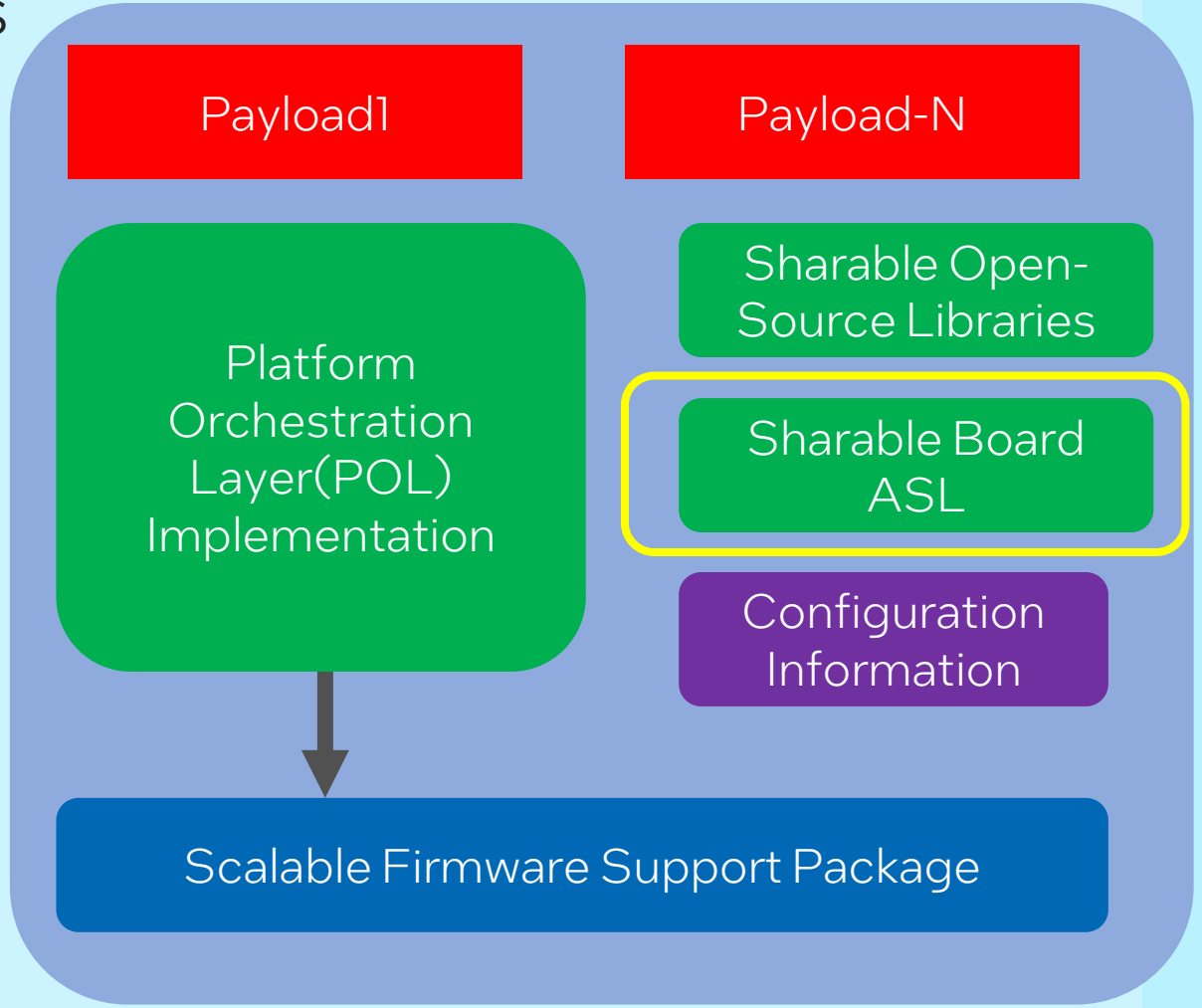
POL Boot Flow Stages w/ Bootloaders

coreboot tianocore SLIM BOOTLOADER

Early Initialization
- coreboot romstage
- EDK II SEC/PEI
- Slim bootloader stage 1

Late Initialization
- Coreboot ramstage
- EDK II DXE
- Slim bootloader stage 2

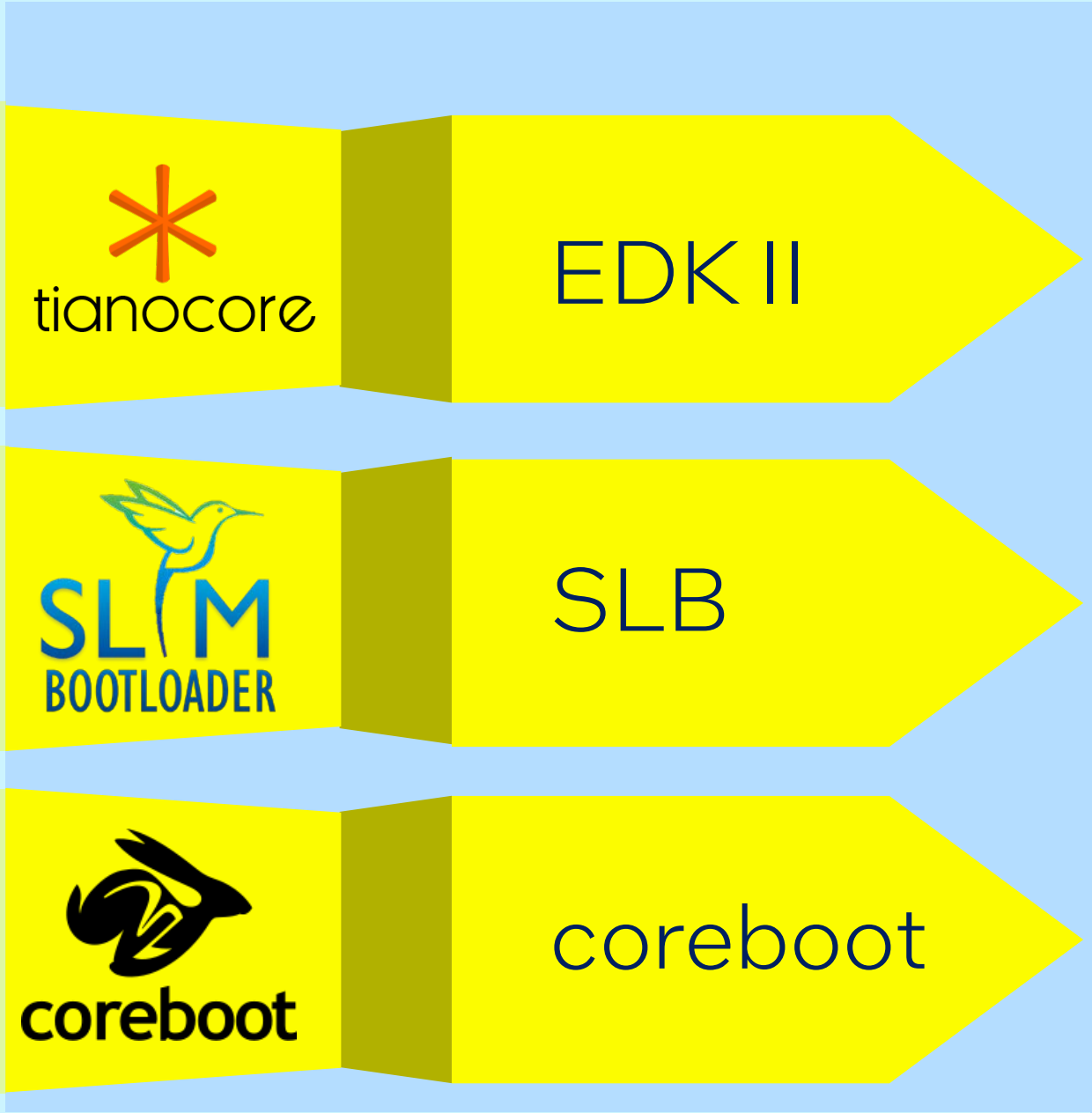
Runtime
- OEM SMM



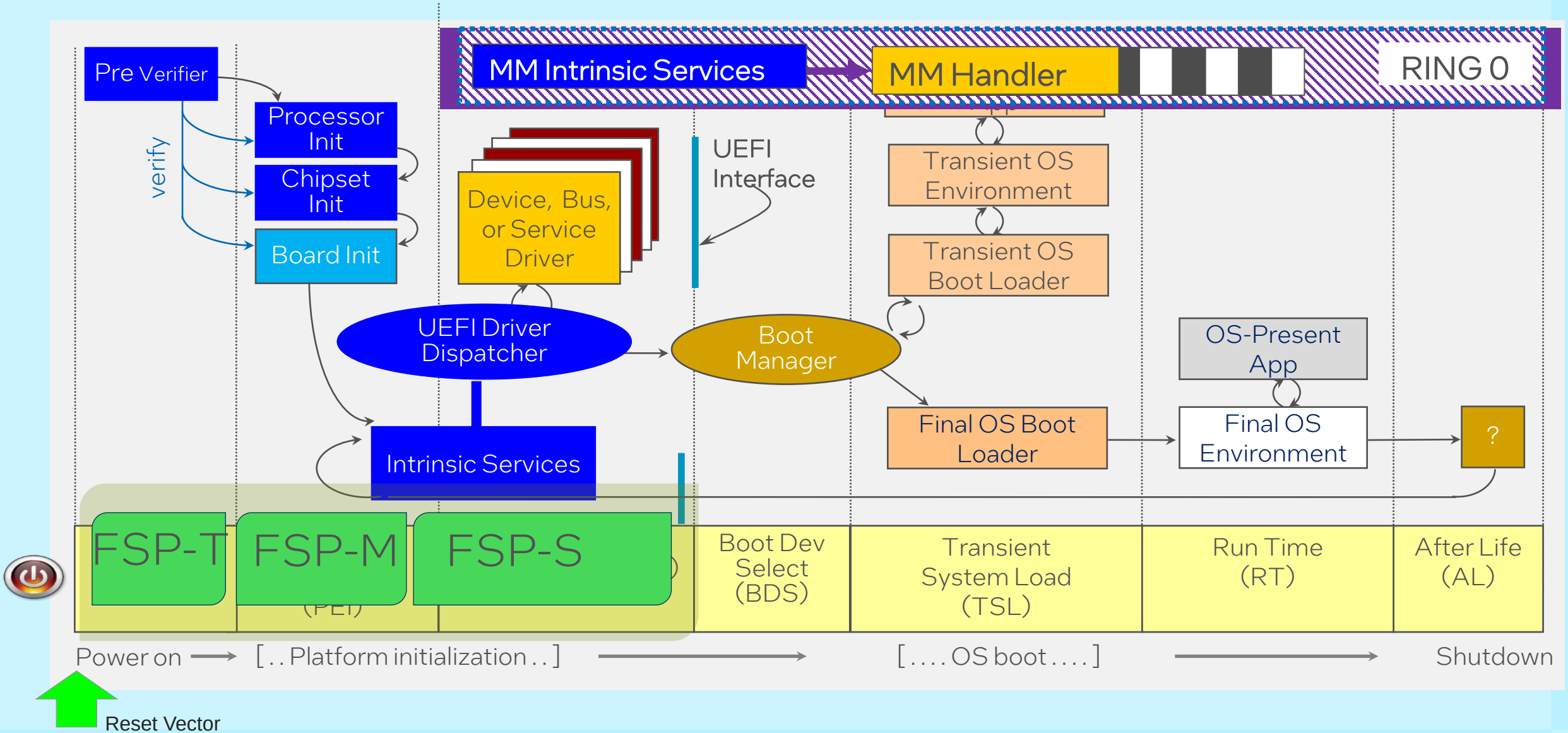
POL High Level Architecture

intel

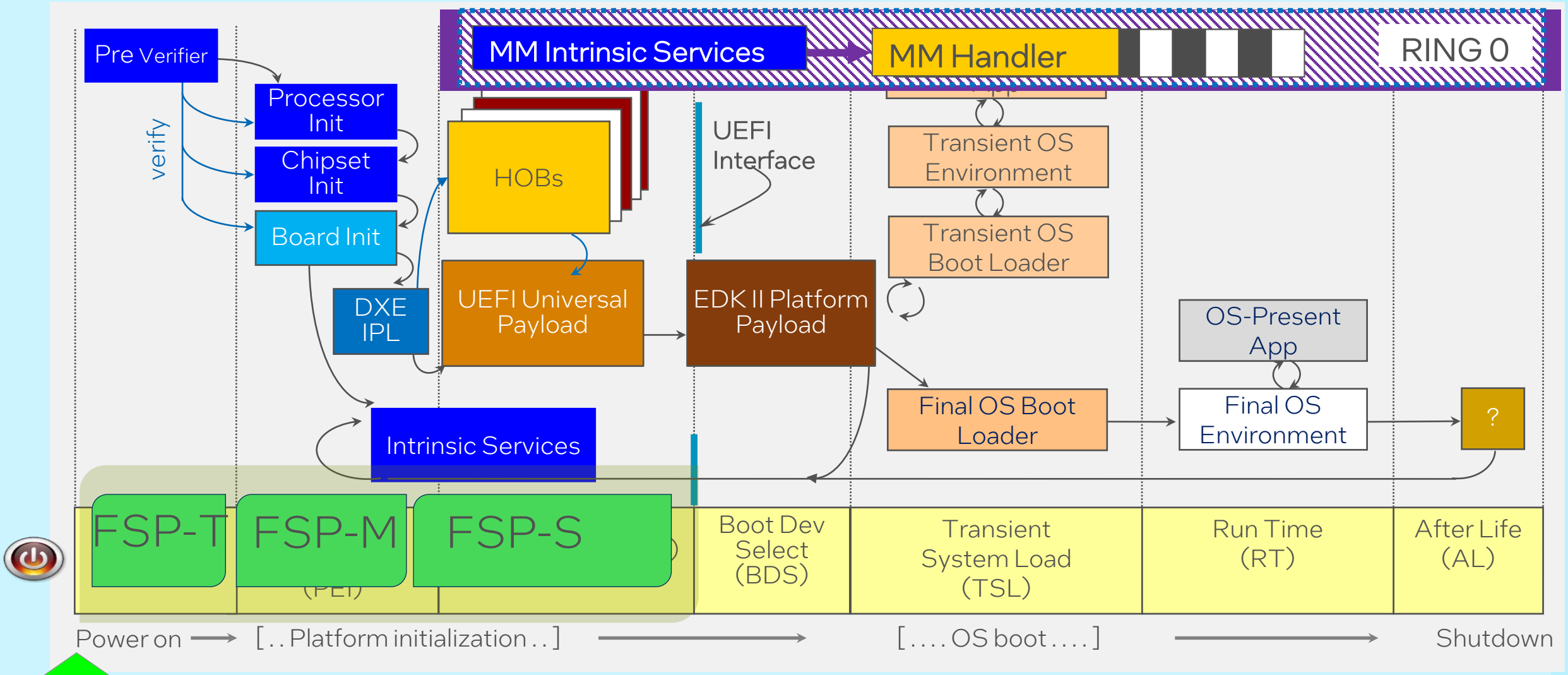
Platform Orchestration Layer (POL) Examples



UEFI – PI & EDK II Boot Flow – Intel® FSP

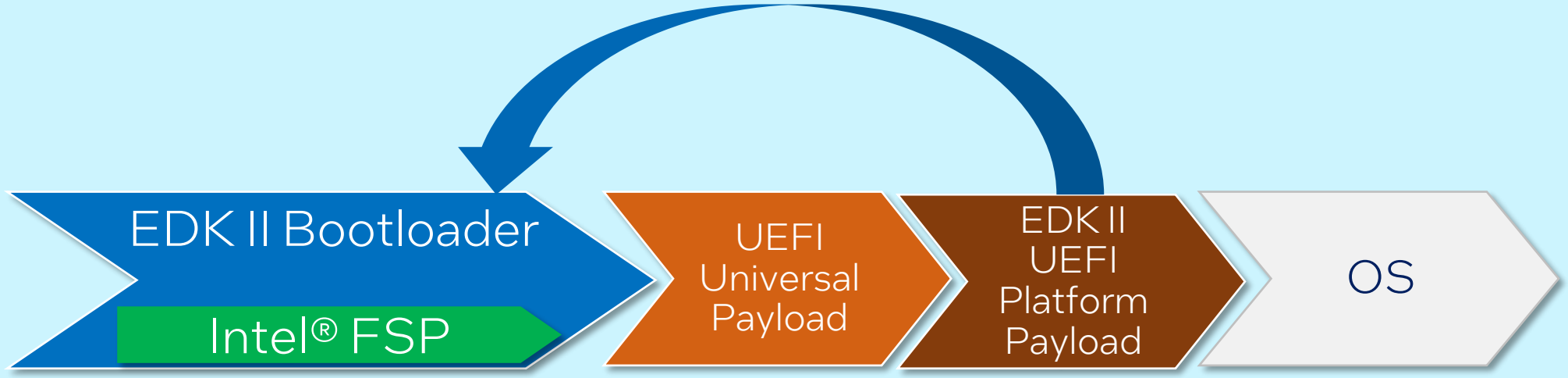


UEFI – PI & EDK II Boot Flow – Intel® FSP w/ USF

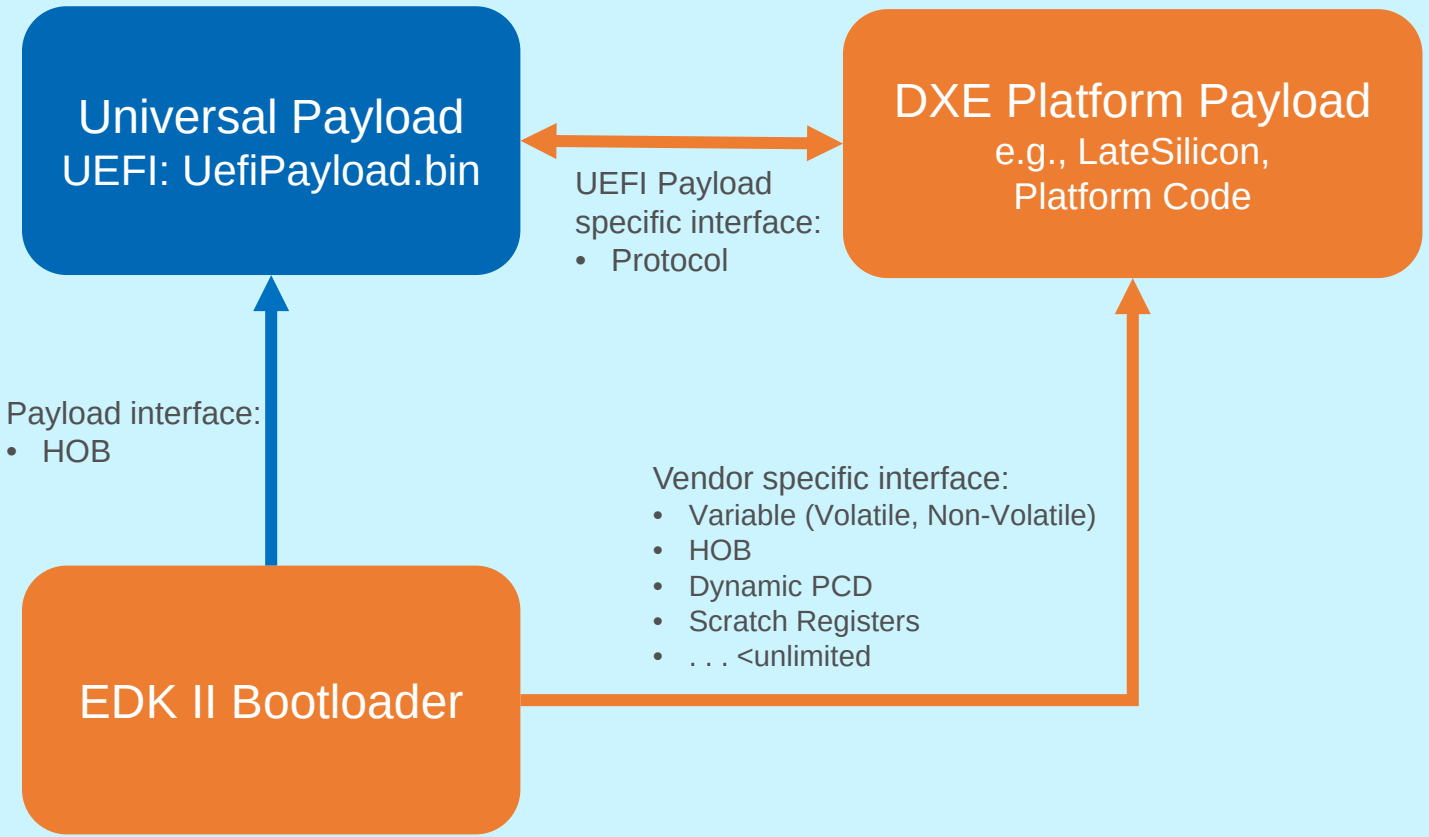


UEFI – PI & EDK II Boot Flow – Intel® FSP w/ USF

- EDK II UEFI Platform Payload is an optional component. It consists of platform specific implementations that must be done in payload phase.
- Theoretically this component can be eliminated by moving all implementations to EDK II Bootloader.



UEFI – PI & EDK II Data Flow w/ Universal Payload



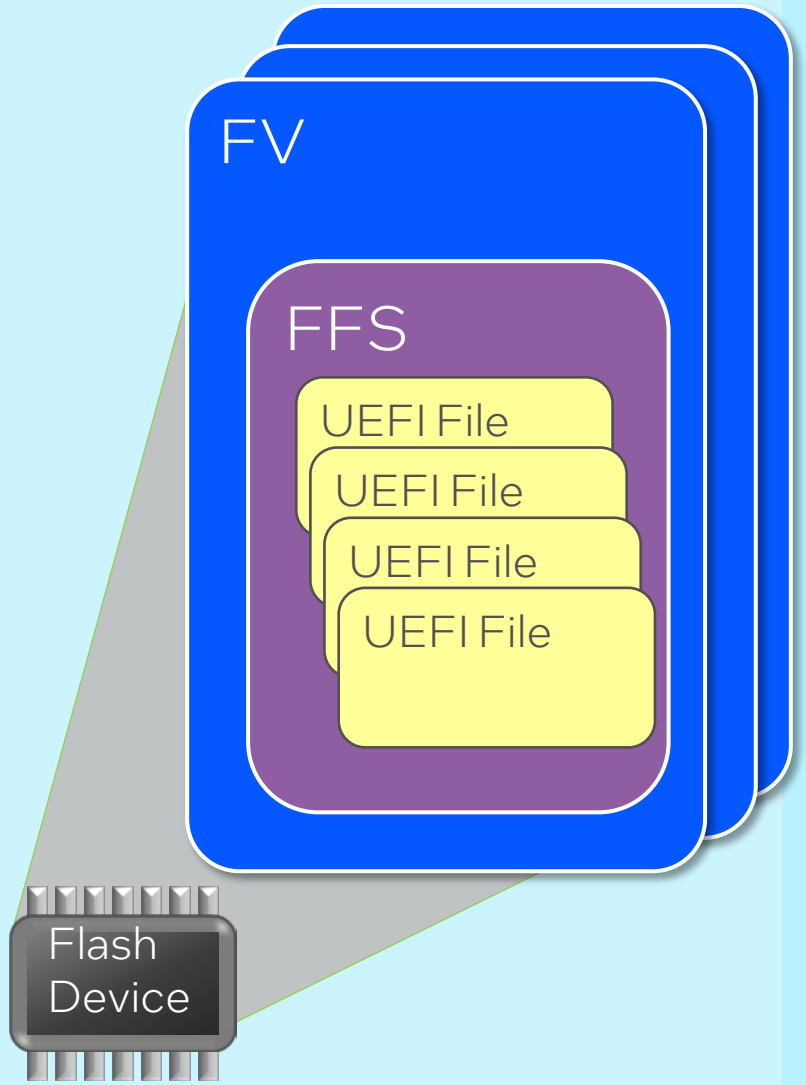
Universal Payload, EFI Platform Payload and EDK II Bootloader

- EDK II Bootloader passes SOC and platform information through HOBs to UEFI Universal Payload.
- UEFI Universal Payload interacts with EDK II UEFI Platform Payload through Protocols.
- EDK II Bootloader can use any mechanism to pass information to EDK II UEFI Platform Payload since both are owned by the platform vendor.

EDK II Uses UEFI Firmware Volumes (FV)

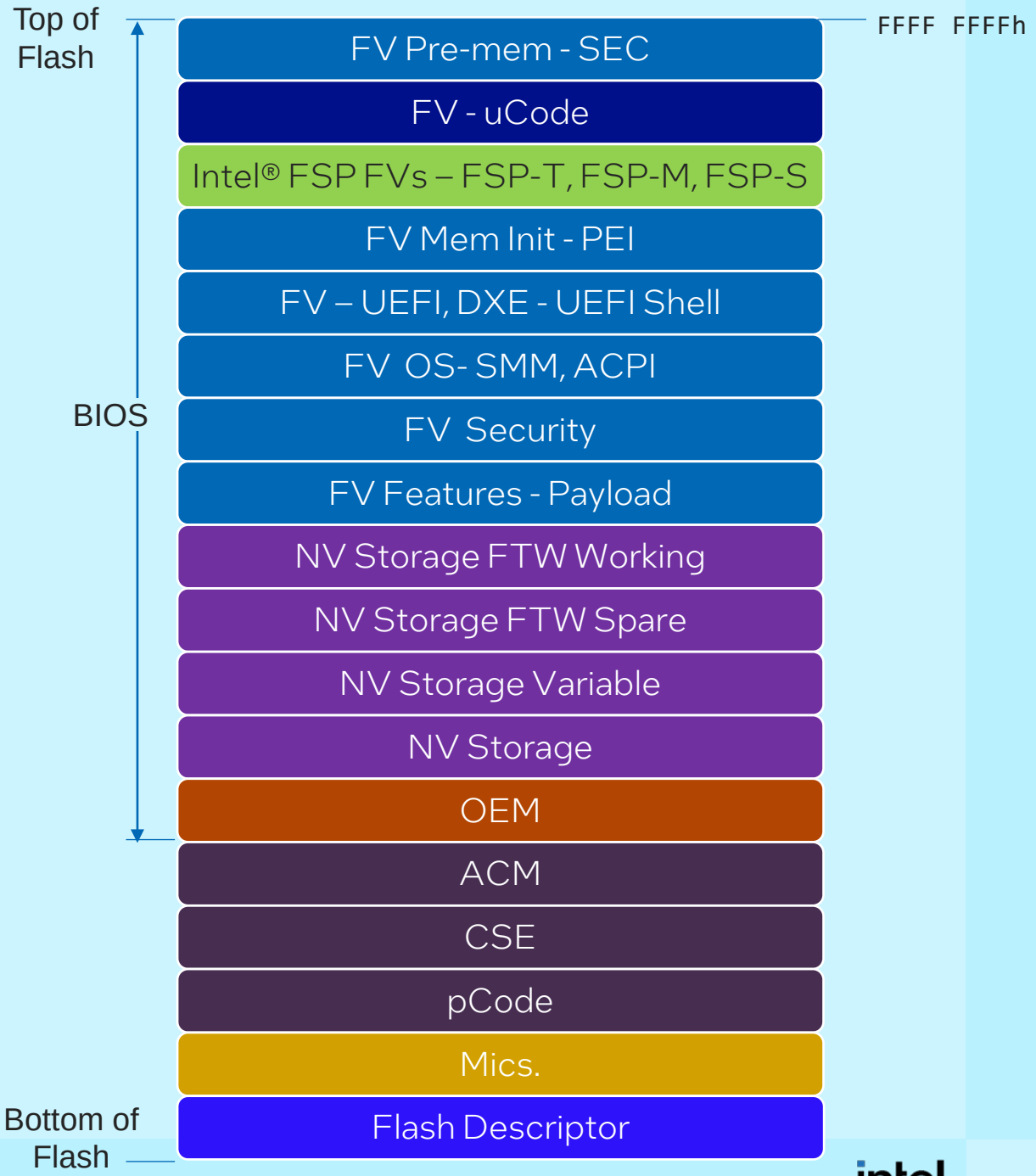
- Platform Initialization - Firmware Volume
  - Basic storage repository for data and code is the Firmware Volume (FV)
  - Each FV is organized into a file system, each with attributes
  - One or more Firmware File Sections (FFS) files are combined into a FV
  - Flash Device may contain one or more FVs.
  - .FDF file controls the layout → .FD image(s)

PI Spec Vol 3



EDK II Integrated Firmware Image (IFWI) Layout

- Example of the Flash layout of the EDK II using the Minimum Platform Architecture Specification, Min-Platform
- Different Firmware Volumes for different stages of the boot flow corresponding to the UEFI boot flow.
- Minimum Platform Architecture Specification



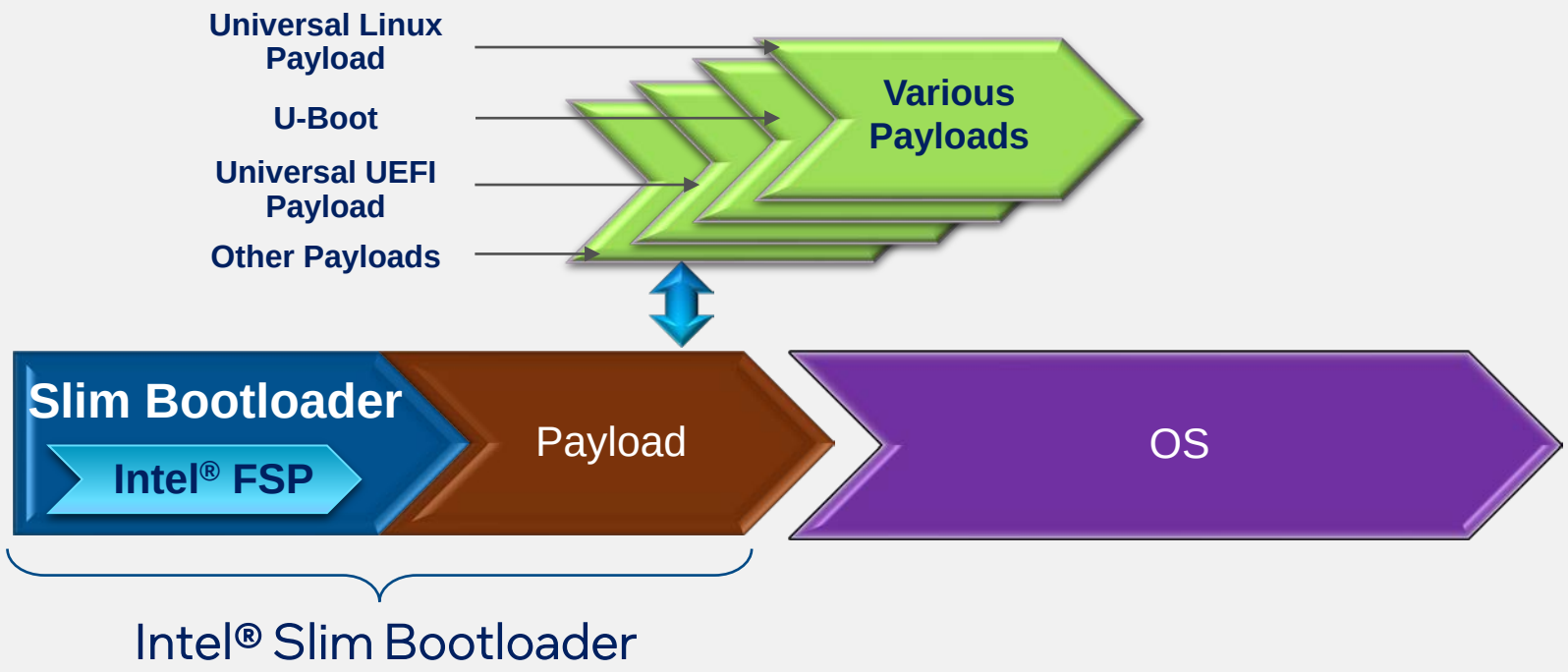
Intel® Slim Bootloader

Intel Internet of Things (IoT) Group focus on:

- Extreme fast boot
- Small footprint
- Security & Function safety

- Intel® Slim Bootloader - https://github.com/slimbootloader

High Level Architecture



Initialization: Board and silicon initialization, including resource allocation, GPIO, ACPI, etc.
Payloads: Generic media drivers, custom features, OS specific loading protocols, etc.

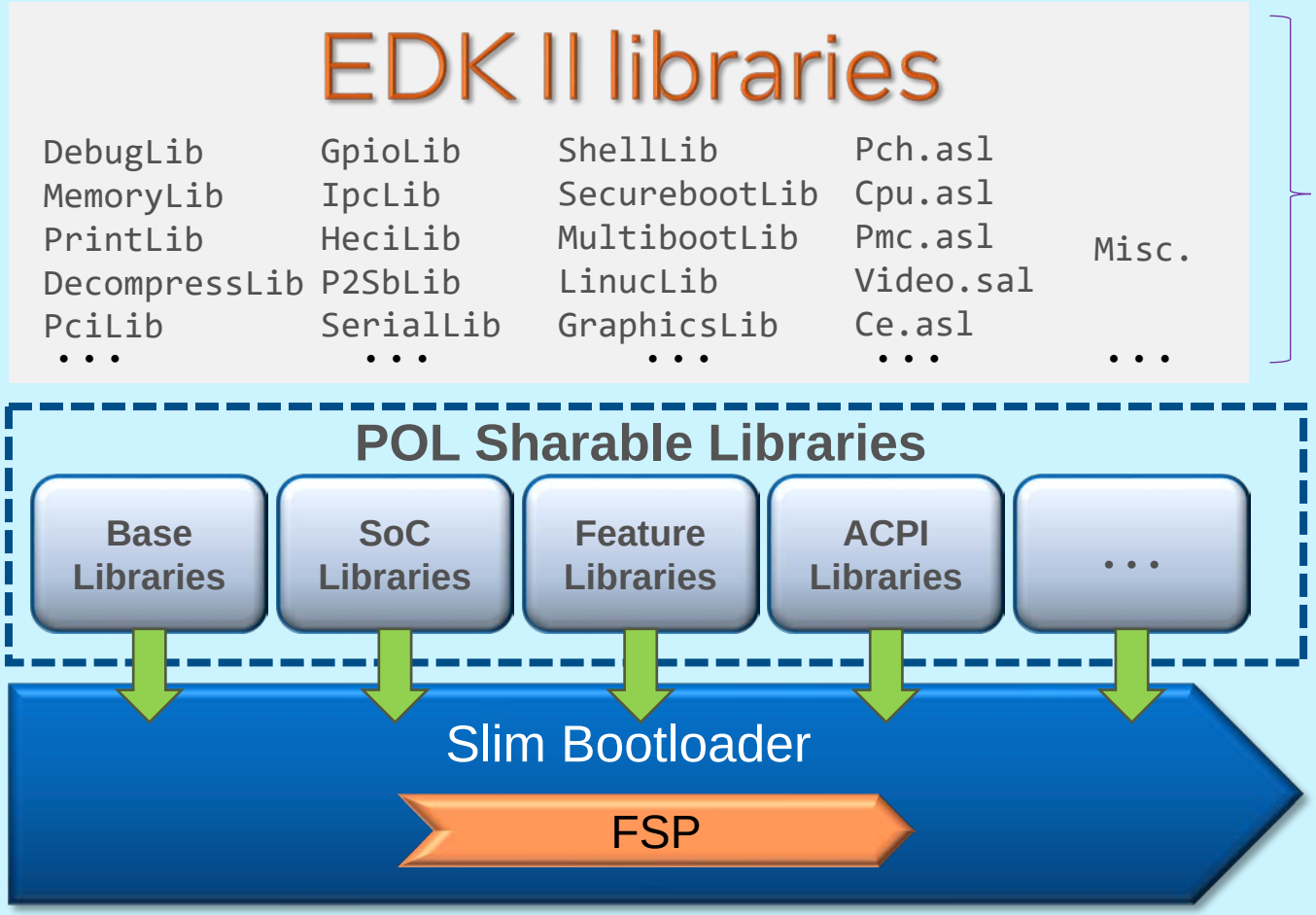
Slim Bootloader Boot Stages

- Stage 1A
  - Reset Vector stage starts with assembly code
  - Basic initialization including setting up temporary memory, debug output
- Stage 1B
  - Memory initialization stage
  - Loads configuration data
- Stage 2
  - Post Memory stage
  - Silicon initialization
  - ACPI, PCI Enumeration, etc
- Universal Payload
  - OS boot logic
  - Media drivers
  - Firmware Update (FWU)

Slim Bootloader Static Library Class as in EDK II

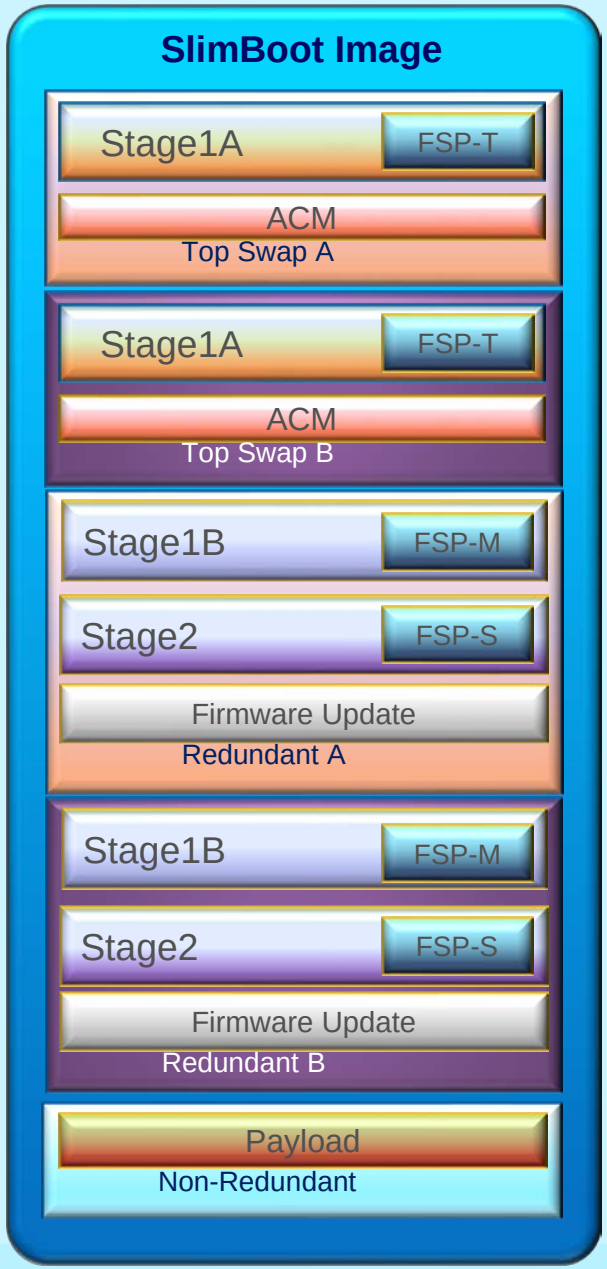
Static library class to abstract API interfaces for platform initialization.

- Slim bootloader leverages EDK II style libraries as well as the ACPI ASL files
- Makes faster enabling of a platform



Flash Image Layout

- Slim Bootloader supports redundancy
- Relies on hardware support for boot block redundancy
  - Boot block includes reset vector code
  - HW support needed to switch to redundant boot block
  - Firmware update flow depends on this layout



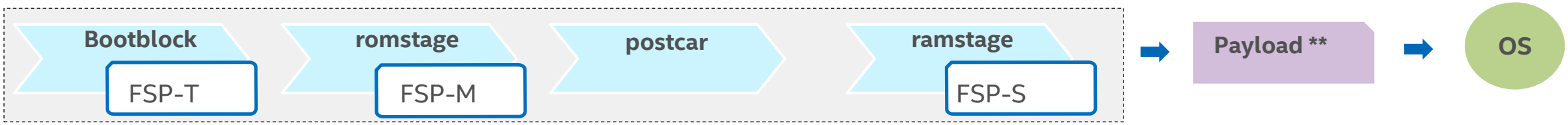
Intel Architecture - Coreboot Overview

[coreboot documentation]

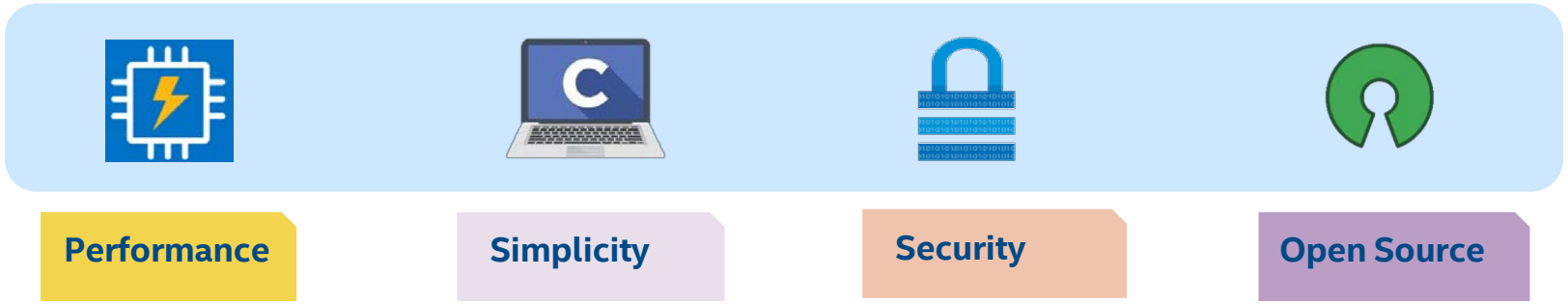
https://www.coreboot.org/

Coreboot

- Core (essential, stripped-down firmware) boot (to boot the platform)



- Idea: Perform basic hardware initialization before passing control to a payload** that boots the OS***
- Principles:

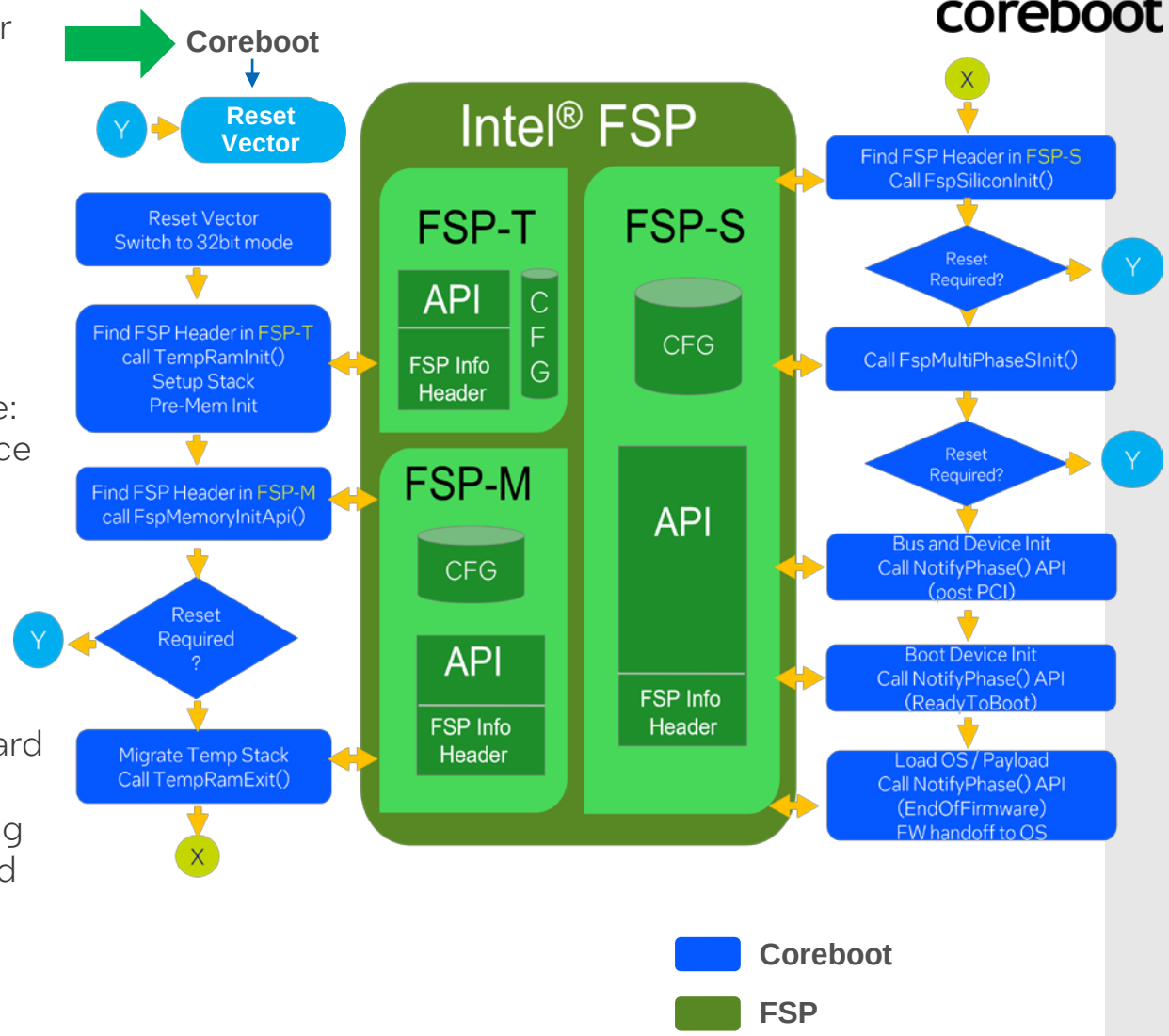


**Payload: Depth charge, Linux Boot, Tianocore, SeaBios .... Almost all payload are compatible with coreboot

***OS: Any market standard operating system

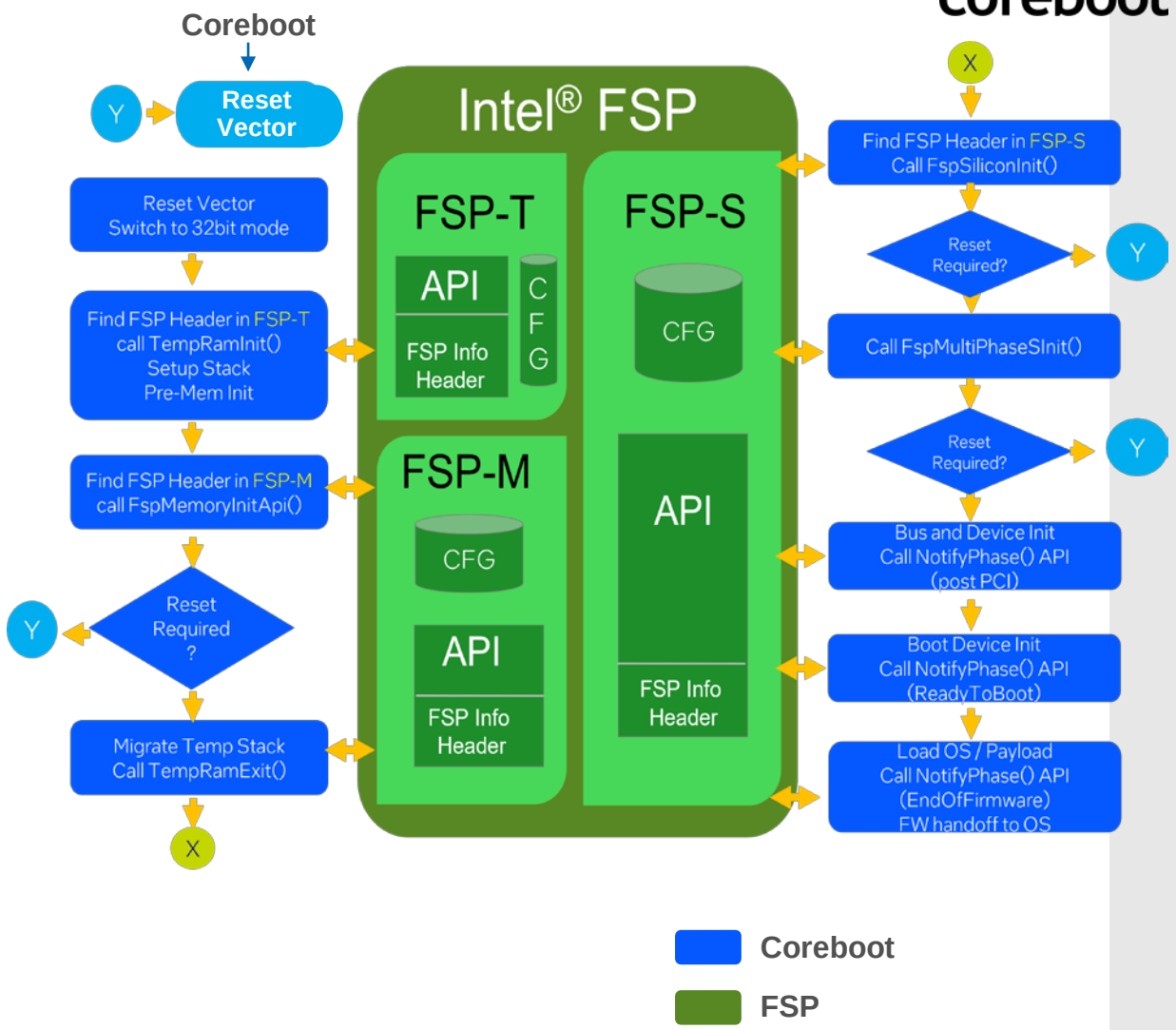
Coreboot boot flow using Intel® FSP v2.3

- coreboot owns reset vector and contains the real mode reset vector handler code
- Optionally coreboot can call FSP-T for CAR setup and create stack
- coreboot to fill in required UPDs before calling FSP-M for memory
- On exit of FSP-M, coreboot to tear down CAR and do the required Silicon programming including filling up UPDs for FSP-S before calling FSP-S to initialize Chipset
- If supported by the FSP and the bootloader enables multi-phase silicon initialization by setting FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit to a non-zero value:
- On exit of FSP-S, coreboot to perform PCI enumeration and resource allocation.
- Bootloader calls the FspMultiPhaseSiInit() API with the EnumMultiPhaseGetNumberOfPhases parameter to discover the number of silicon initialization phases supported by the bootloader.
- Bootloader must call the FspMultiPhaseSiInit() API with the EnumMultiPhaseExecutePhas parameter n times, where n is the number of phases returned previously. Bootloader may perform board specific code in between each phase as needed.
- The number of phases, what is done during each phase, and anything the bootloader may need to do in between phases shall be described in the Integration Guide.
- coreboot calls NotifyPhase at proper stages before handing over to payload.



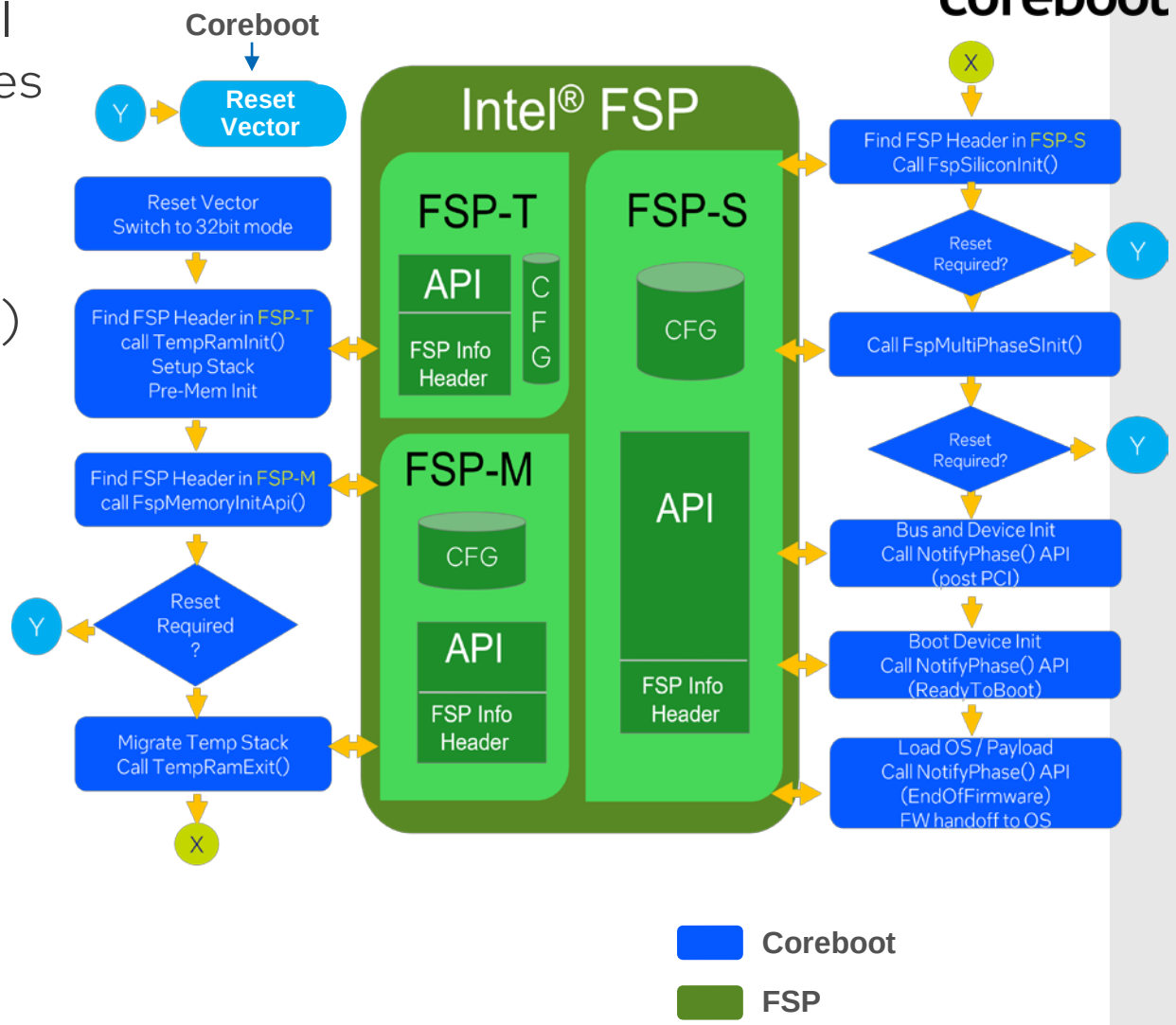
Coreboot boot flow using Intel® FSP v2.3

1. coreboot owns reset vector and contains the real mode reset vector handler code
2. Optionally coreboot can call FSP-T for CAR setup and create stack
3. coreboot to fill in required UPDs before calling FSP-M for memory init.
4. On exit of FSP-M, coreboot to tear down CAR and do the required Silicon programming including filling up UPDs for FSP-S before calling FSP-S to initialize Chipset
5. If supported by the FSP and the bootloader enables multi-phase silicon initialization by setting FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit to a non-zero value:
6. On exit of FSP-S, coreboot to perform PCI enumeration and resource allocation.

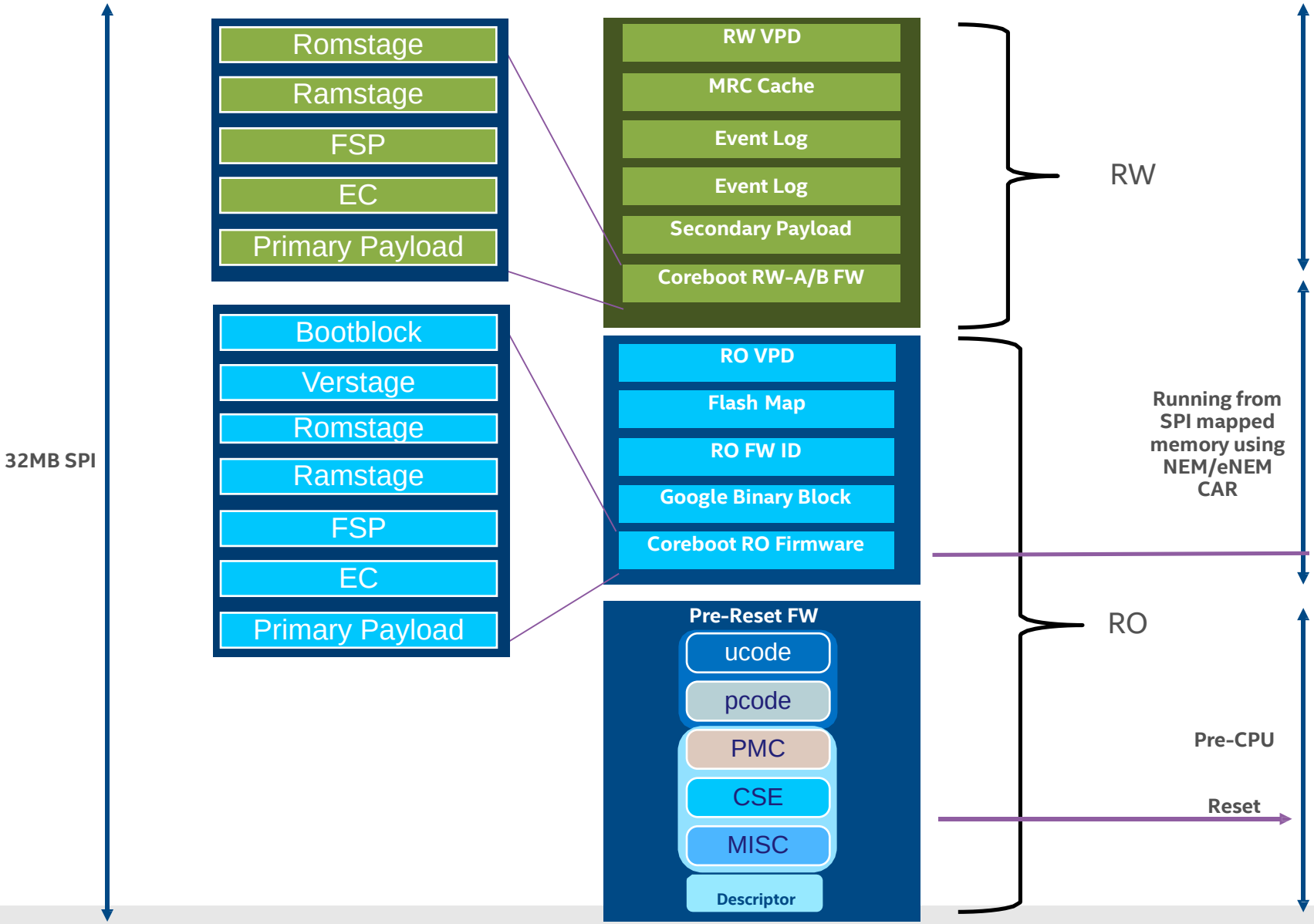


Coreboot boot flow using Intel® FSP v2.3

7. Bootloader calls the FspMultiPhaseSiInit() API with the EnumMultiPhaseGetNumberOfPhases parameter to discover the number of silicon initialization phases supported by the bootloader.
8. Bootloader must call the FspMultiPhaseSiInit() API with the EnumMultiPhaseExecutePhas parameter n times, where n is the number of phases returned previously. Bootloader may perform board specific code in between each phase as needed.
9. The number of phases, what is done during each phase, and anything the bootloader may need to do in between phases shall be described in the FSP Integration Guide.
10. coreboot calls NotifyPhase at proper stages before handing over to the payload.

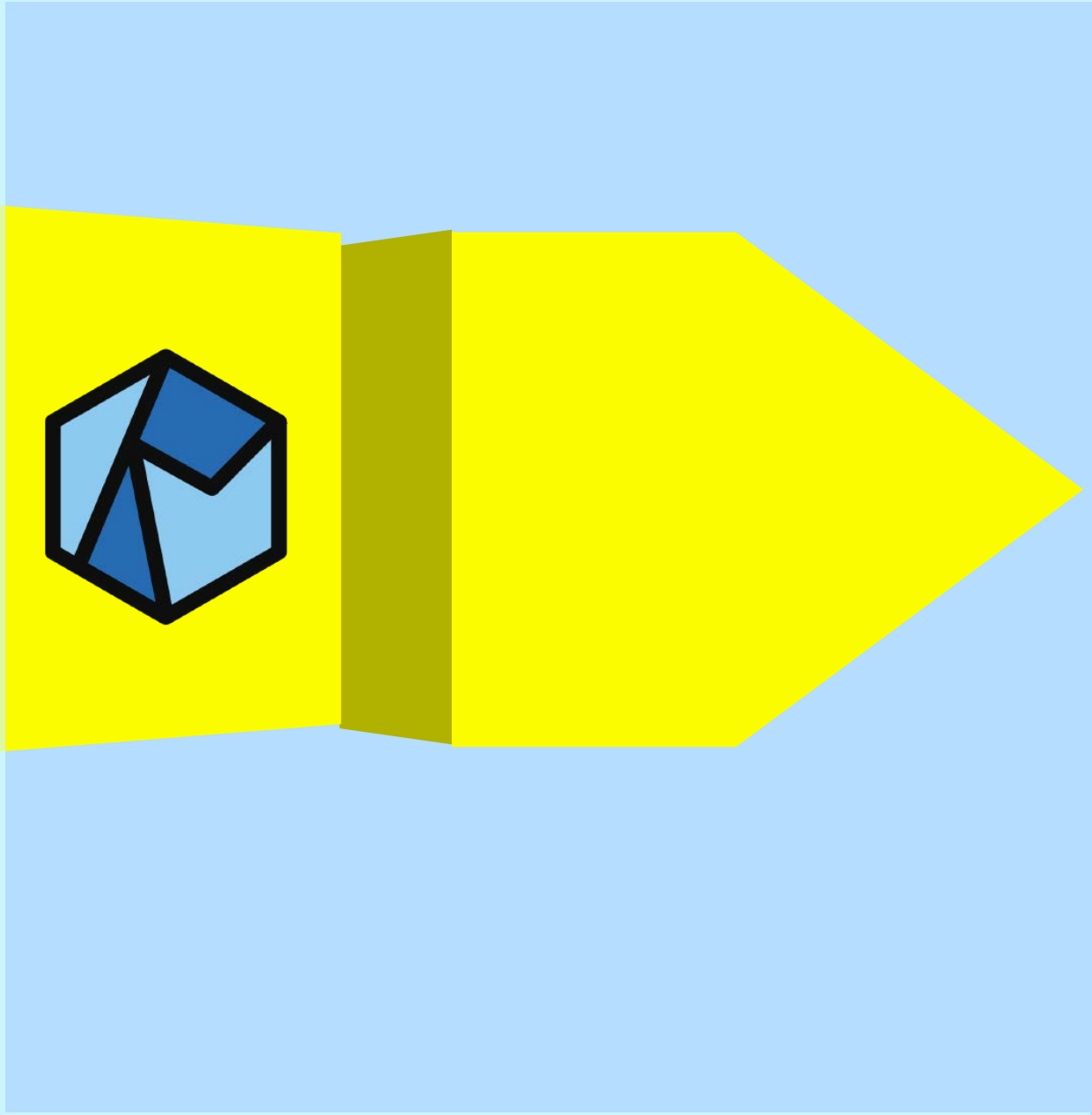


Integrated Firmware Image (IFWI) Layout



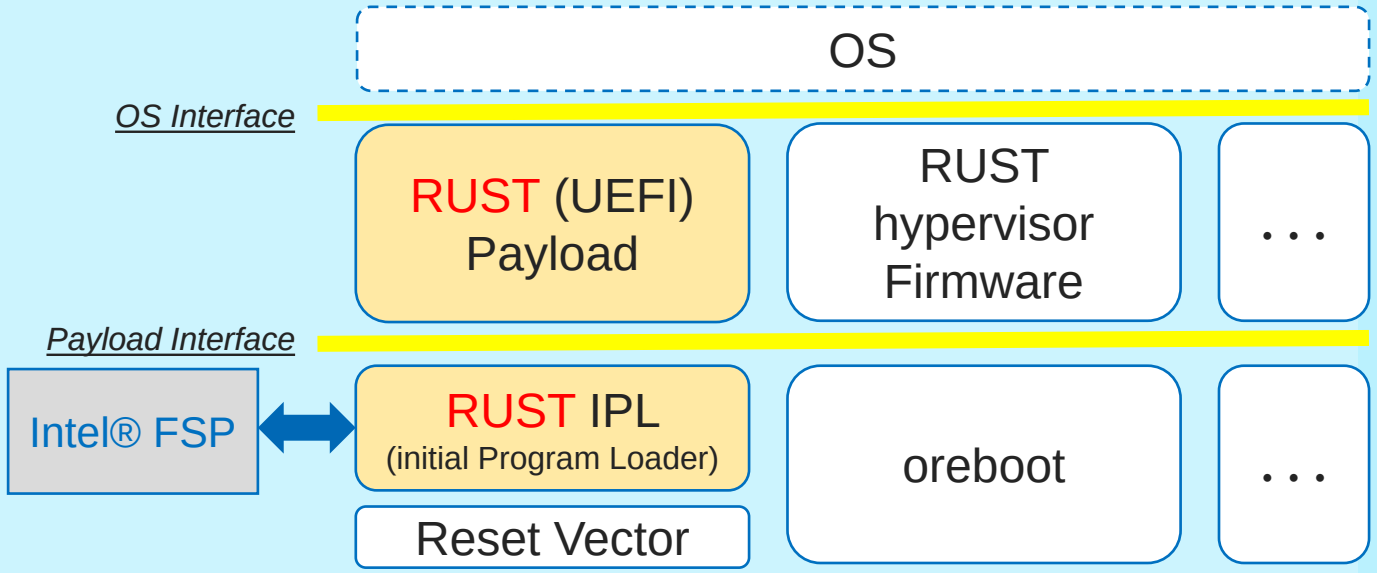
Platform Orchestration Layer (POL)

Future



Platform Orchestration Layer (POL) – Modern Language

- Modern Language – RUST-based firmware as either a payload or the platform layer implementation



- Oreboot https://github.com/oreboot/oreboot
- RUST architecture is based upon https://github.com/jyao1/rust-firmware
- The RUST API for FSP wrapper is at https://github.com/jyao1/rust-firmware/tree/master/rust-fsp-wrapper

Summary

- Platform Orchestration Layer (POL)
- Governs the Boot Flow
- Implementation of the Build Software


intel