

Winning the Race to Bare Metal

~

UEFI Hypervisors

“He who gets there fastest with the mostest wins”

Summary

- There is a race to bare metal between black hats and white hats
- UEFI pre-OS capabilities provide the ability to launch tools BEFORE the OS bootloader is called
- Bare metal hypervisor technology is powerful, well understood and maturing
- Commodity support for UEFI 2.0 is emerging
- The combination of bare metal hypervisors and UEFI has great potential and implications for system exploitation and security

Our Custom Bare Metal Hypervisor

- Lightweight and pluggable by design
- Contains a custom hypervisor runtime debugger
- Capable of hosting x64 line of Windows Operating Systems
- Implements Intel[®] VMX (VT-x)

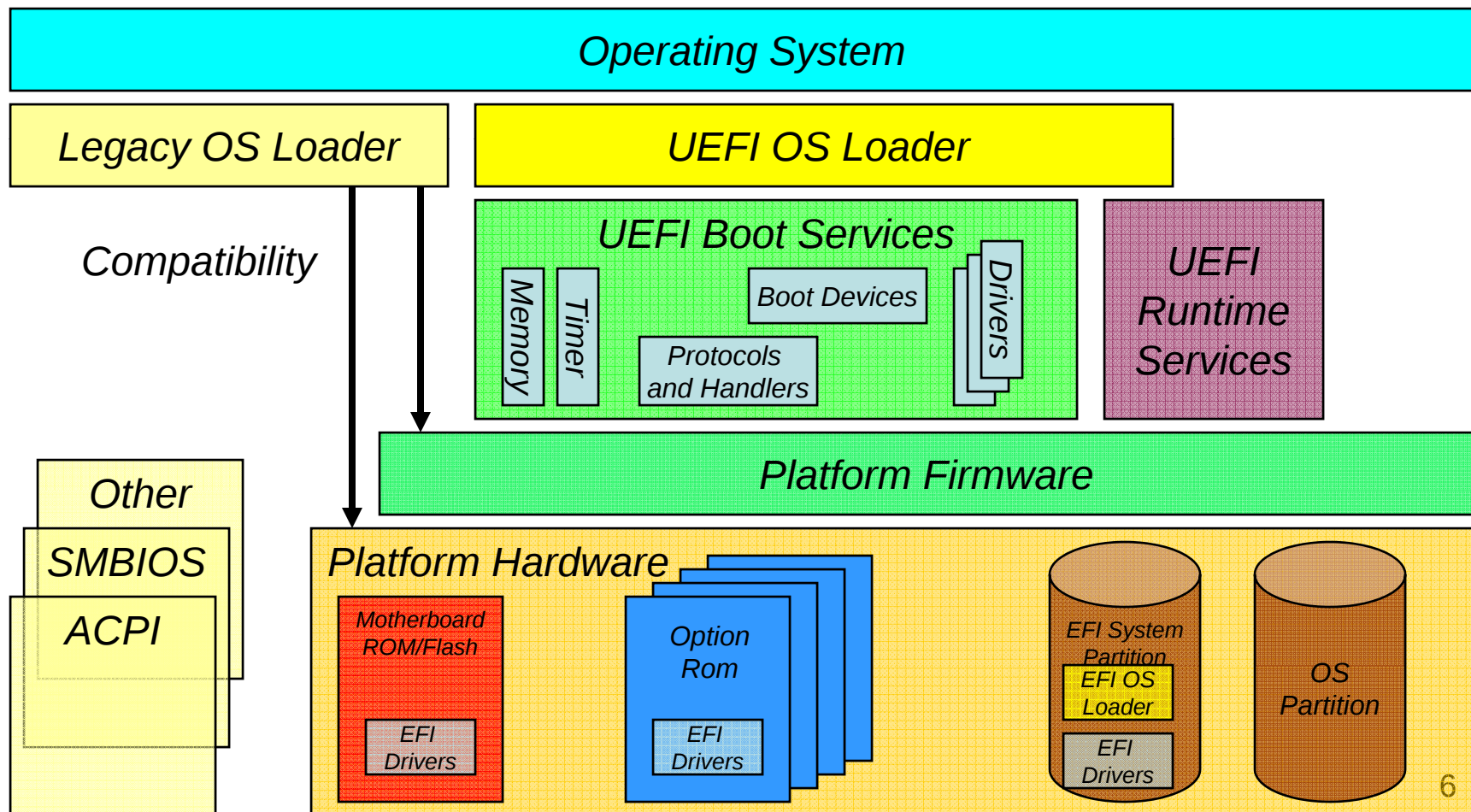
UEFI

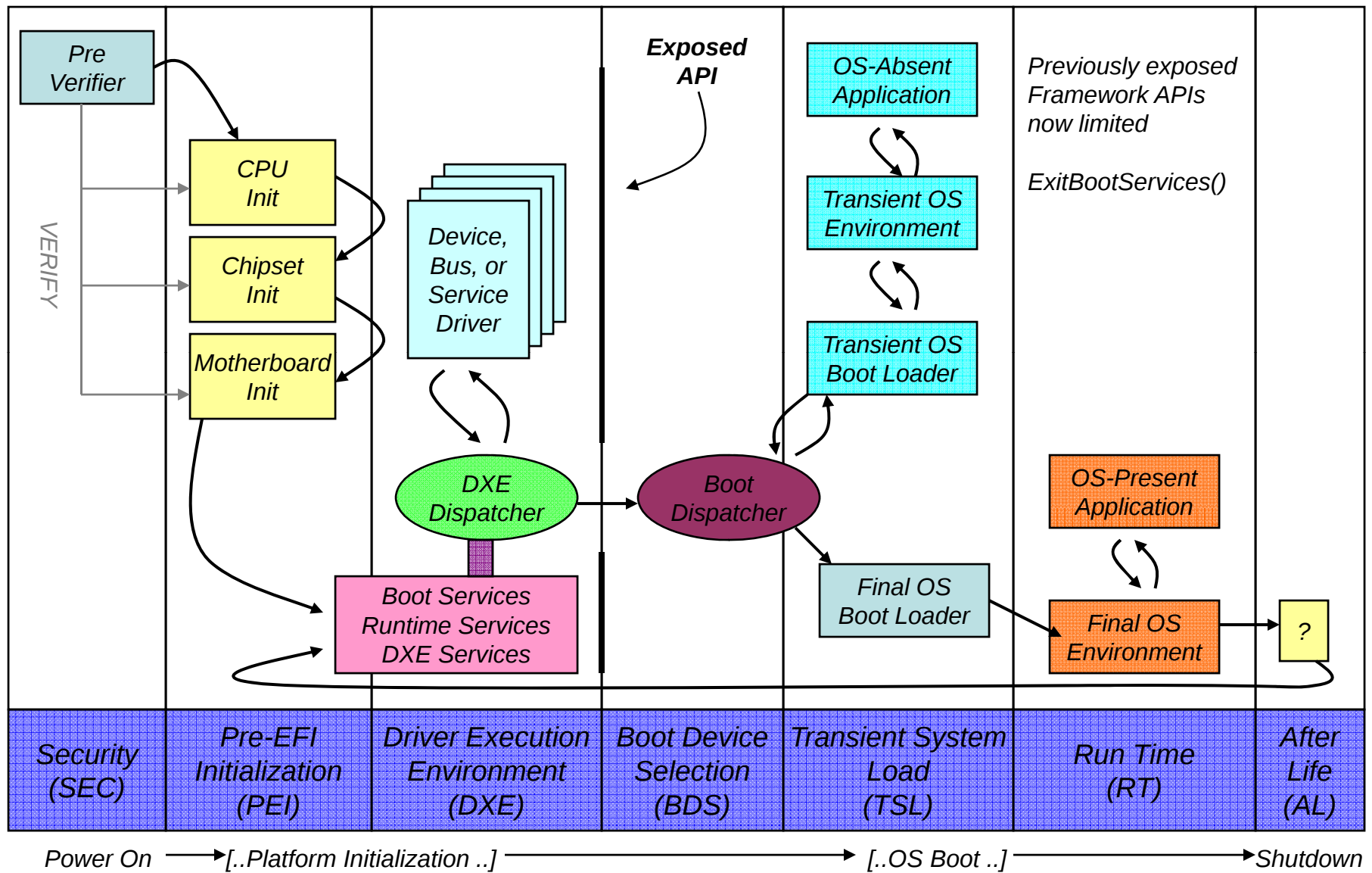
- **U**nified **E**xtensible **F**irmware **I**nterface
 - UEFI describes a programmatic interface between the platform firmware and the OS
 - UEFI replaces and extends the functionality of legacy BIOS
 - UEFI 2.0 is supported in Microsoft Windows Vista x64 SP1 and Linux distributions
 - Commodity motherboards are beginning to support UEFI 2.0
 - Some of the more interesting and powerful features of UEFI are the pre-OS capabilities

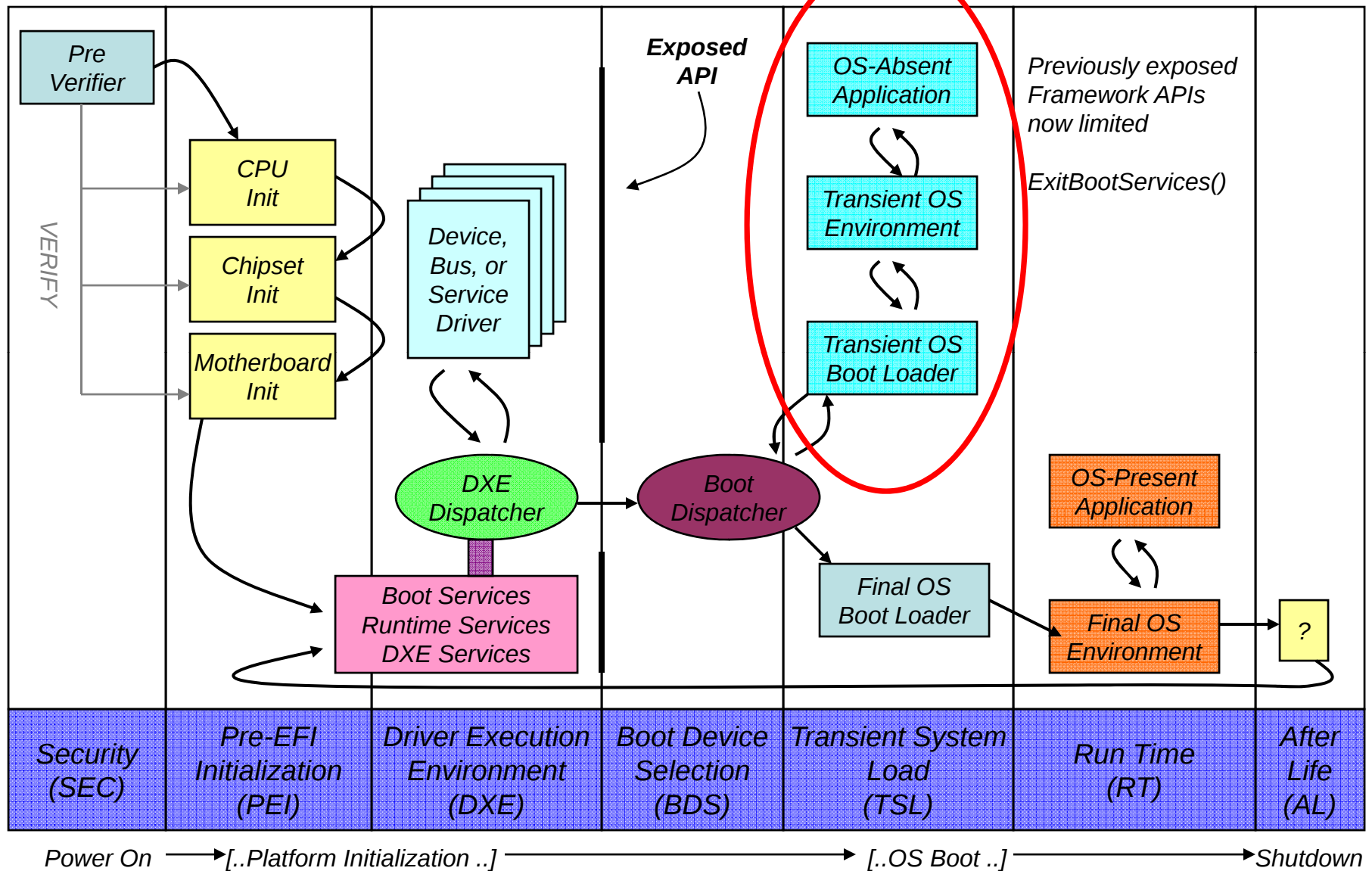
UEFI Goals

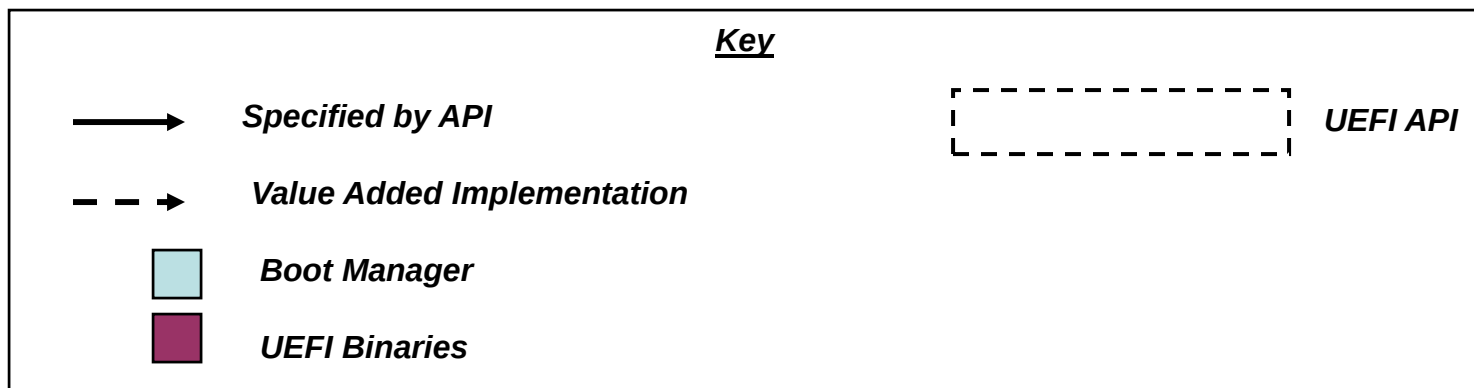
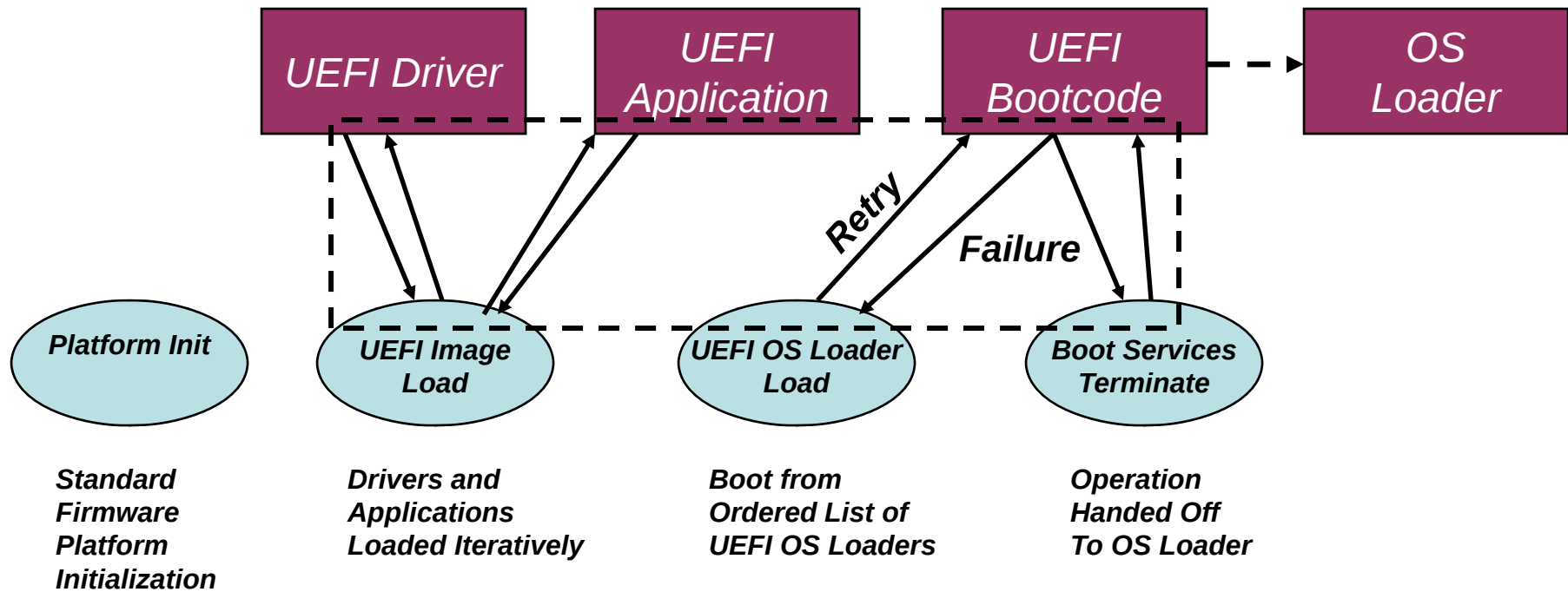
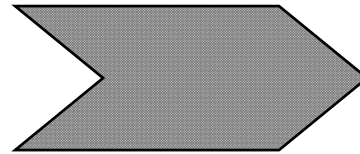
- OS neutrality
- Well defined, clean, and extensible interfaces
- Modularity
- HLL friendly
- Scalability
- Revolutionary Boot Manager
- Kick-Ass pre-OS capabilities

UEFI Framework

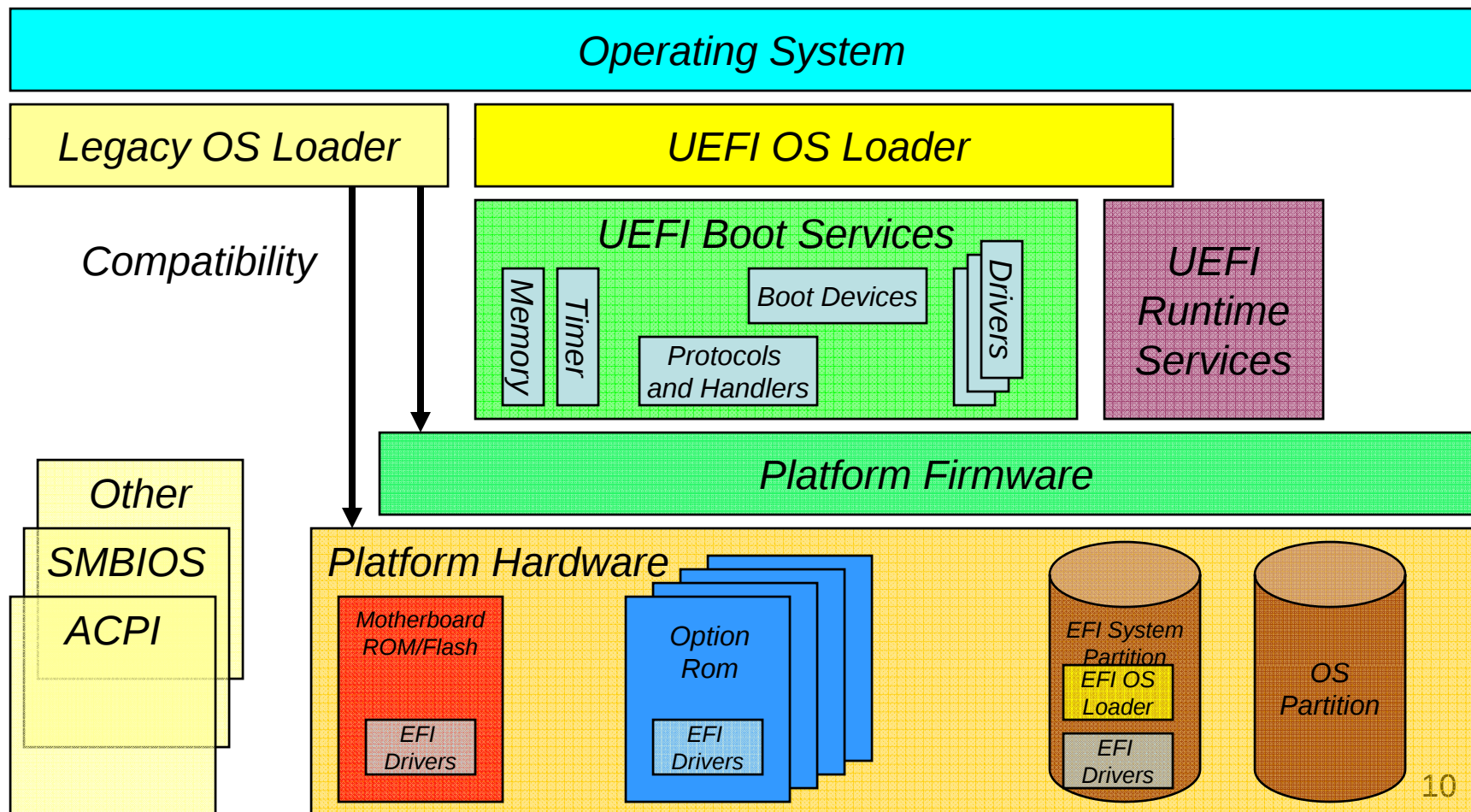




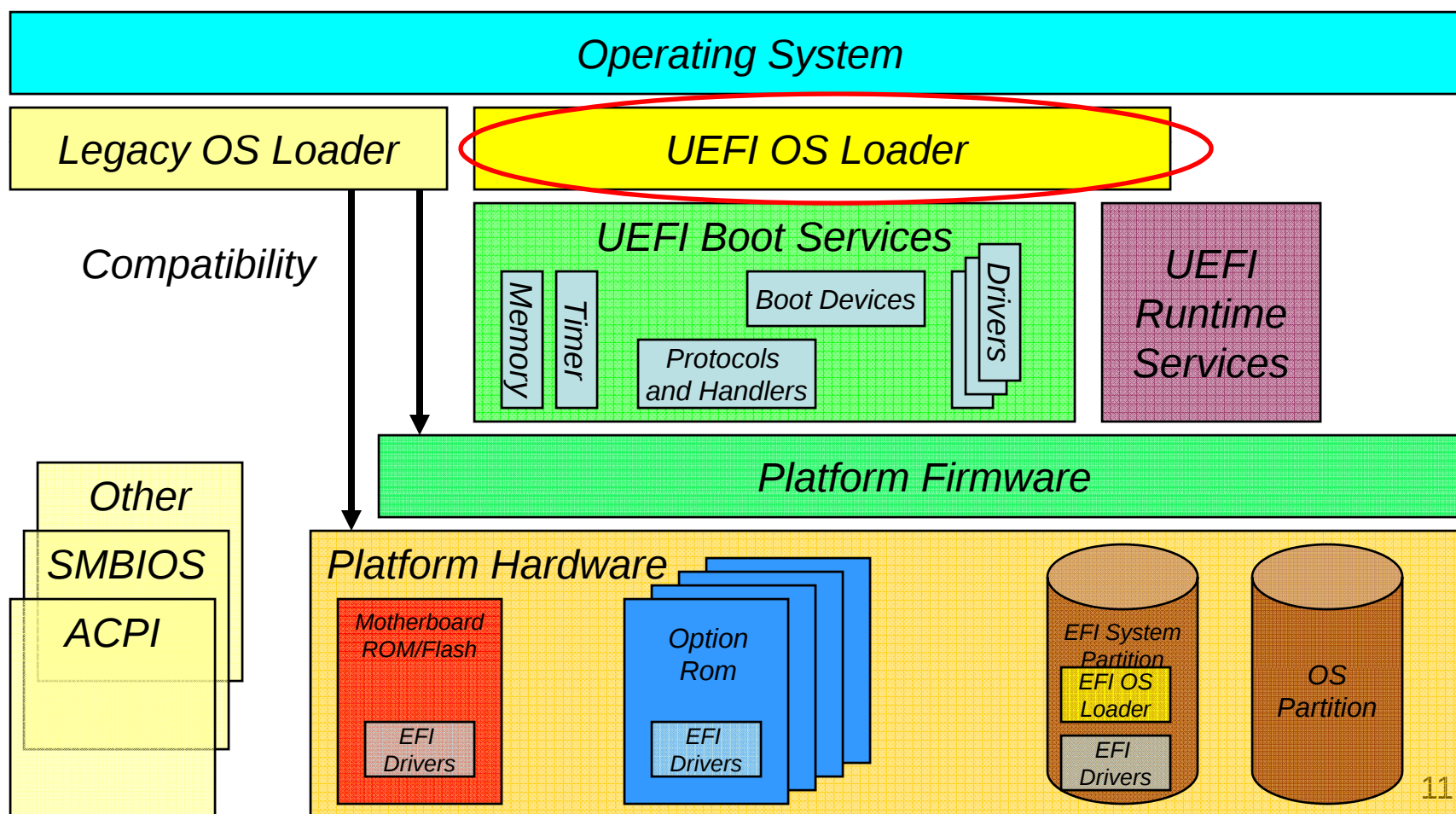




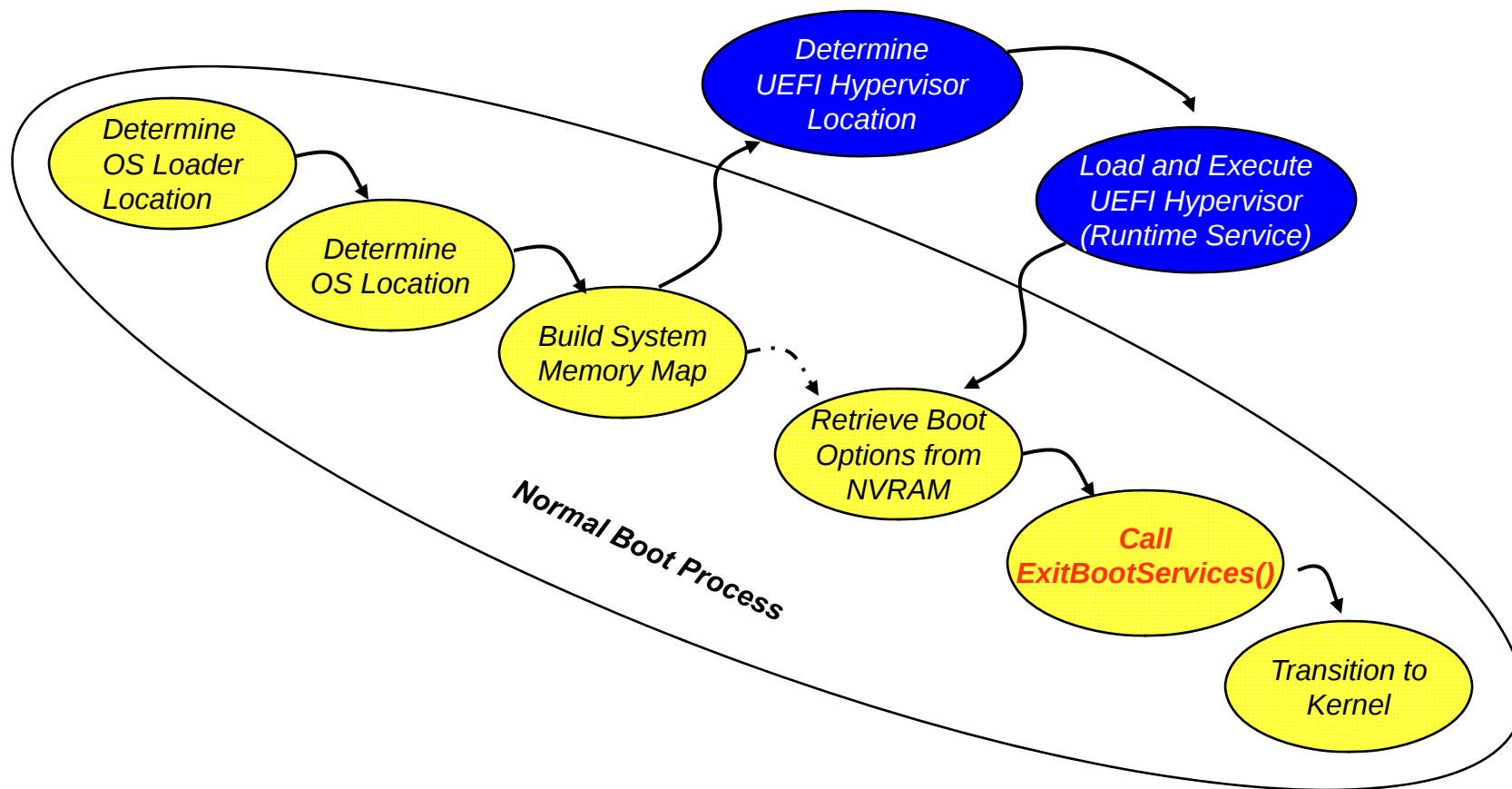
UEFI Framework



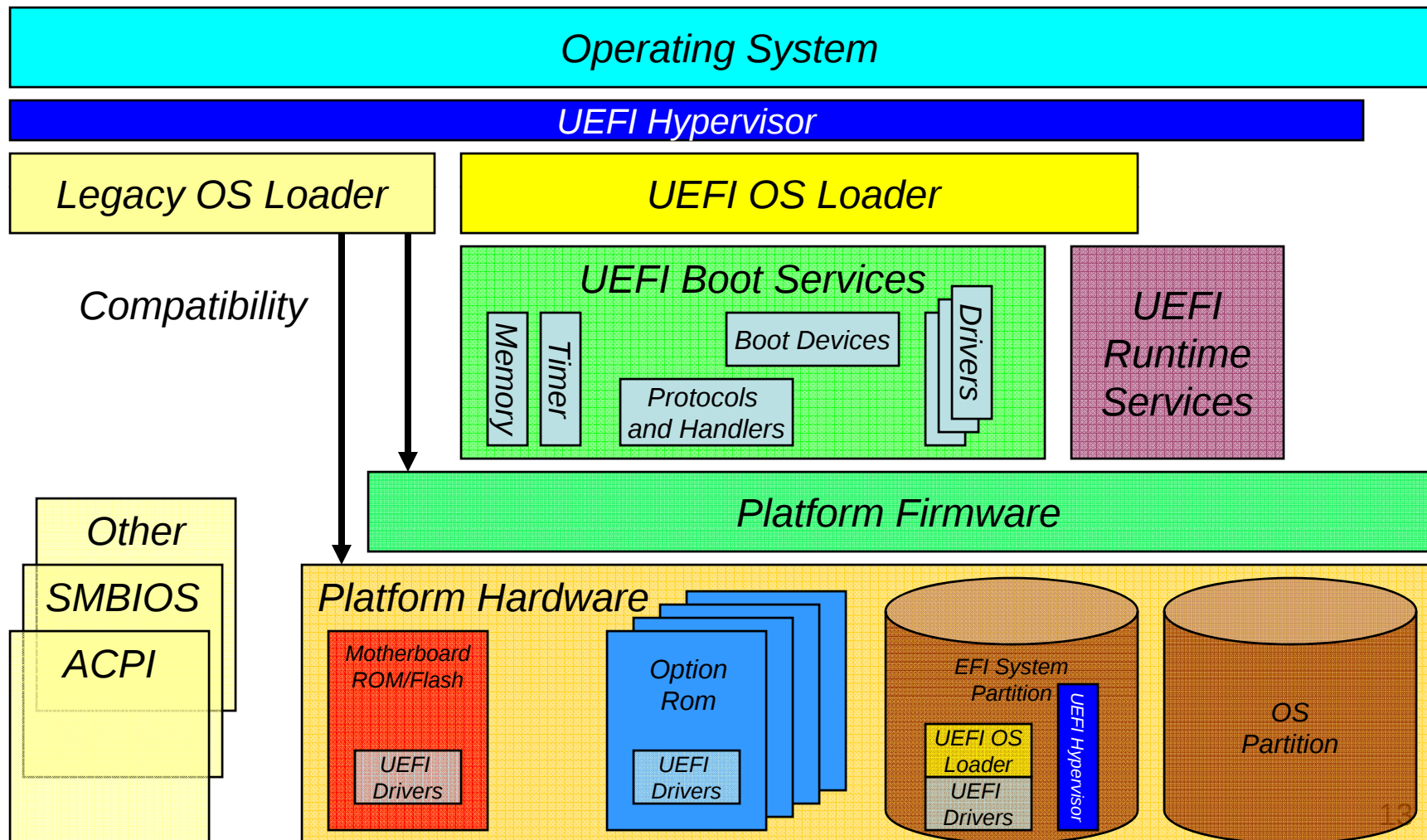
UEFI Framework



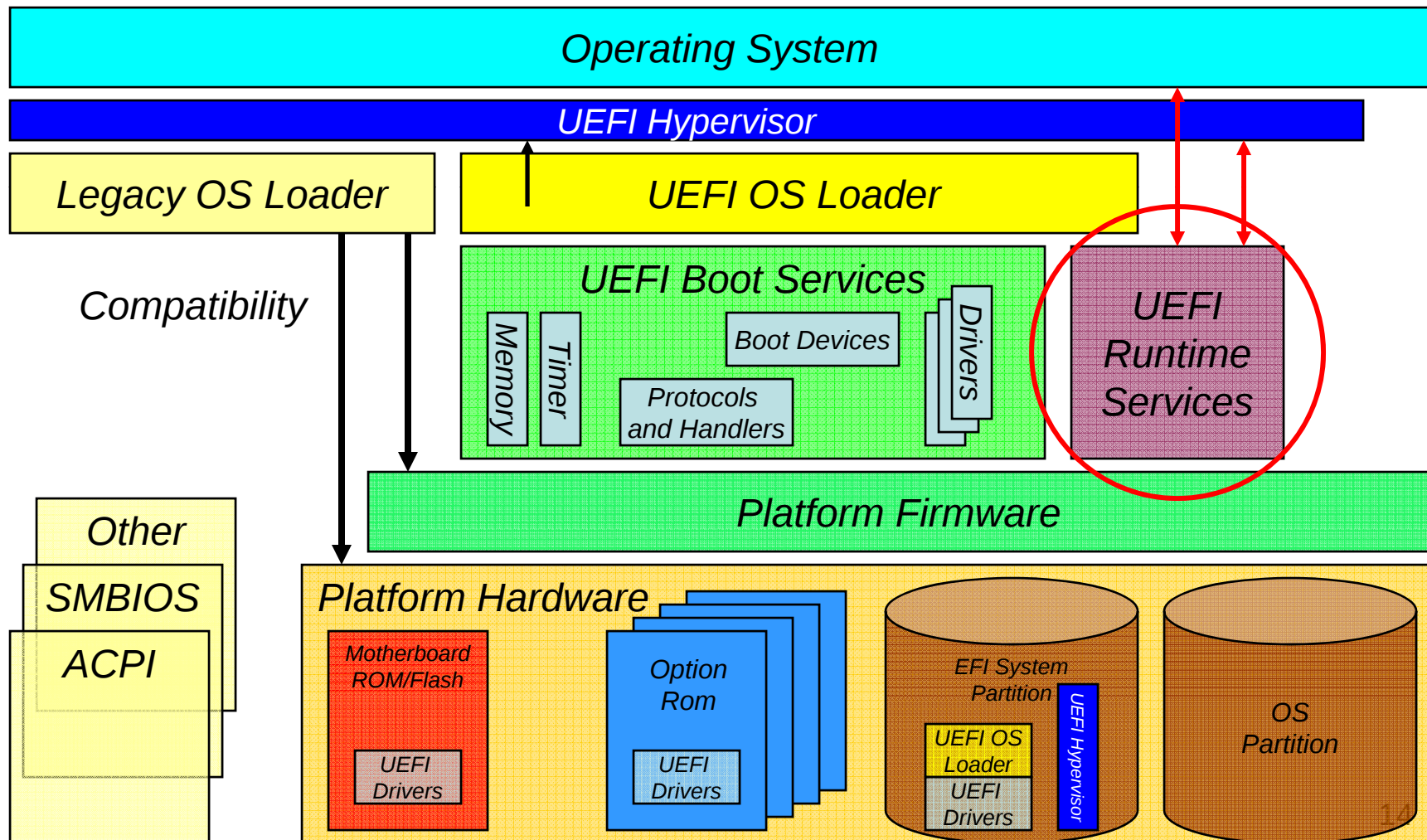
OS Loader Flow



UEFI Hypervisor Framework



UEFI Hypervisor Framework



Key Concepts of the Architecture

- Objects managed by UEFI firmware
- UEFI Images
- Handle Database and protocols
- UEFI System Table
- Events
- Device Paths
- Capsules

Key Concepts of the Architecture

- Objects managed by UEFI firmware
- **EFI Images**
- Handle Database and protocols
- UEFI System Table
- Events
- Device Paths
- Capsules

UEFI Image

- A class of files defined by the specification that contain executable code
- UEFI images contain the old familiar PE/COFF header that defines the format of the executable code
 - UEFI uses a subset of the PE32+ image format with a modified image signature
- The header defines processor type AND the image type
 - UEFI Application
 - UEFI Boot Services Driver
 - UEFI Runtime Driver

UEFI Image

- A class of files defined by the specification that contain executable code
- UEFI images contain the old familiar PE/COFF header that defines the format of the executable code
 - UEFI uses a subset of the PE32+ image format with a modified image signature
- The header defines processor type AND the image type
 - UEFI Application
 - UEFI Boot Services Driver
 - **UEFI Runtime Driver**

Overview of Runtime Services

- Available before, during and after the OS is booted; after **ExitBootService()** is called
- UEFI Runtime Drivers are loaded in memory marked as *EfiRuntimeServiceCode*
- UEFI Runtime data structures are marked as *EfiRuntimeServiceData*
- UEFI Runtime Drivers coexist with and can be invoked by a UEFI-aware OS

UEFI Sample Code

(Note the lack of 16-bit assembly language code!!)

```
/*++
Copyright (c) 1998 Intel Corporation
Module Name:
    rtdriver.c
Abstract:
    Test runtime driver
Revision History
--*/
#include "efi.h"
#include "efilib.h"
//
EFI_STATUS
TestRtUnload (
    IN EFI_HANDLE      ImageHandle
);
//
CHAR16  *RtTestString1 = L"This is string #1";
CHAR16  *RtTestString2 = L"This is string #2";
CHAR16  *RtTestString3 = L"This is string #3";
EFI_GUID RtTestDriverId = { 0xcc2ac9d1, 0x14a9, 0x11d3, 0x8e, 0x77, 0x0, 0xa0,
    0xc9, 0x69, 0x72, 0x3b };
```

```
EFI_STATUS
InitializeTestRtDriver (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_LOADED_IMAGE    *Image;
    EFI_STATUS          Status;

    // Initialize the Library.

    InitializeLib (ImageHandle, SystemTable);
    Print(L"Test RtDriver loaded\n");

    // Add an unload handler

    Status = BS->HandleProtocol (ImageHandle, &LoadedImageProtocol,
    (VOID*)&Image);
    ASSERT (!EFI_ERROR(Status));
    Image->Unload = TestRtUnload;
```

```
// Add a protocol so someone can locate us
```

```
Status = LibInstallProtocolInterfaces (&ImageHandle, &RtTestDriverId, NULL,  
NULL);  
ASSERT (!EFI_ERROR(Status));
```

```
// Modify one pointer to verify fixups don't reset it
```

```
Print(L"Address of RtTestString3 is %x\n", RtTestString3);  
Print(L"Address of RtTestString3 pointer is %x\n", &RtTestString3);  
RtTestString3 = RtTestString2;  
return EFI_SUCCESS;
```

```
}
```

```
EFI_STATUS
```

```
TestRtUnload (  
    IN EFI_HANDLE      ImageHandle  
)
```

```
{
```

```
    DEBUG ((D_INIT, "Test RtDriver unload being requested\n"));  
    LibUninstallProtocolInterfaces (ImageHandle, &RtTestDriverId, NULL,  
NULL);  
    return EFI_SUCCESS;
```

UEFI Hypervisor

- Our challenge was to convert our x64 Windows hypervisor device driver to a UEFI Runtime device driver
- We identified the MSI P45 Platinum motherboard as UEFI 2.0 compliant
- We used the Tianocore UEFI 2.0.0.1 SDK

DEMO

Resources

- UEFI 2.0 Specification – <http://www.uefi.org>
- “*Beyond BIOS – Implementing the Unified Extensible Firmware Interface with Intel’s Framework*” by Zimmerman, Rothman, and Hale - Intel® Press - <http://www.intel.com/intelpress/>

Questions ?

Email: don AT hypervista-tech.com