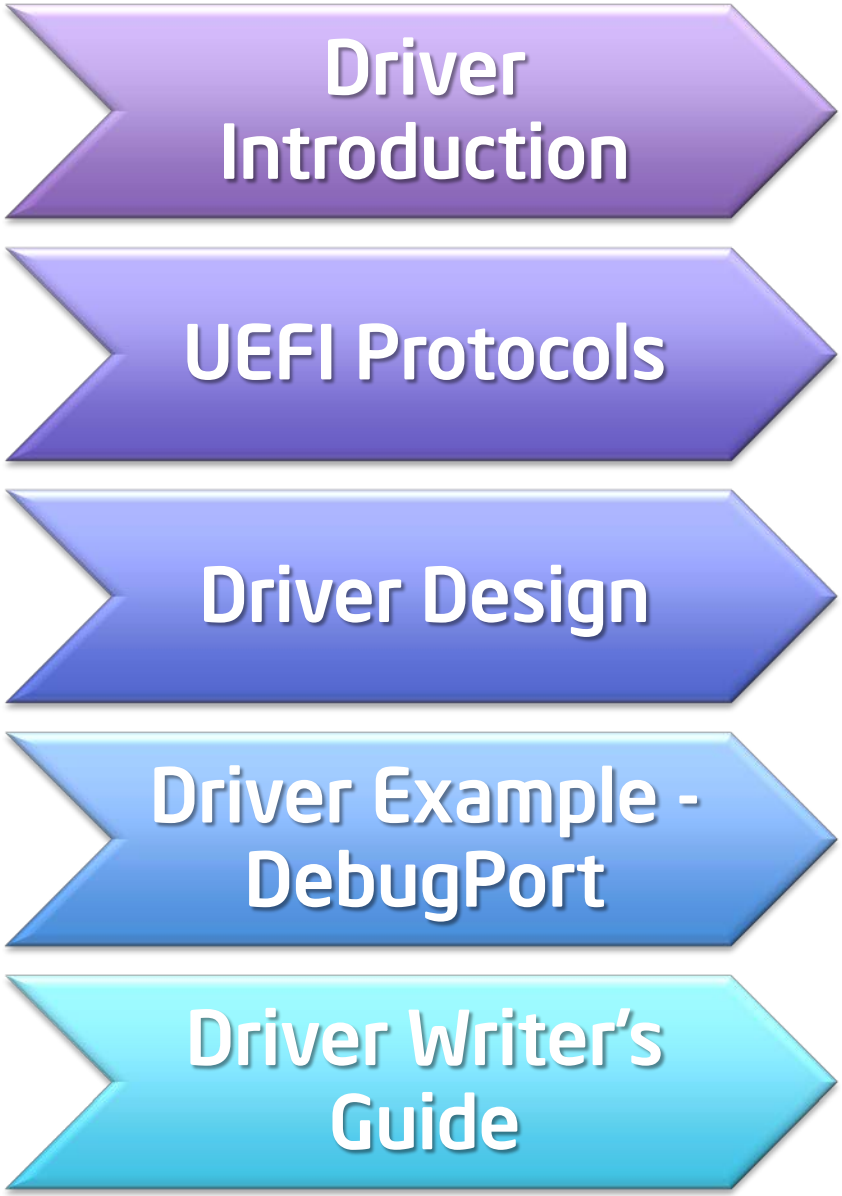
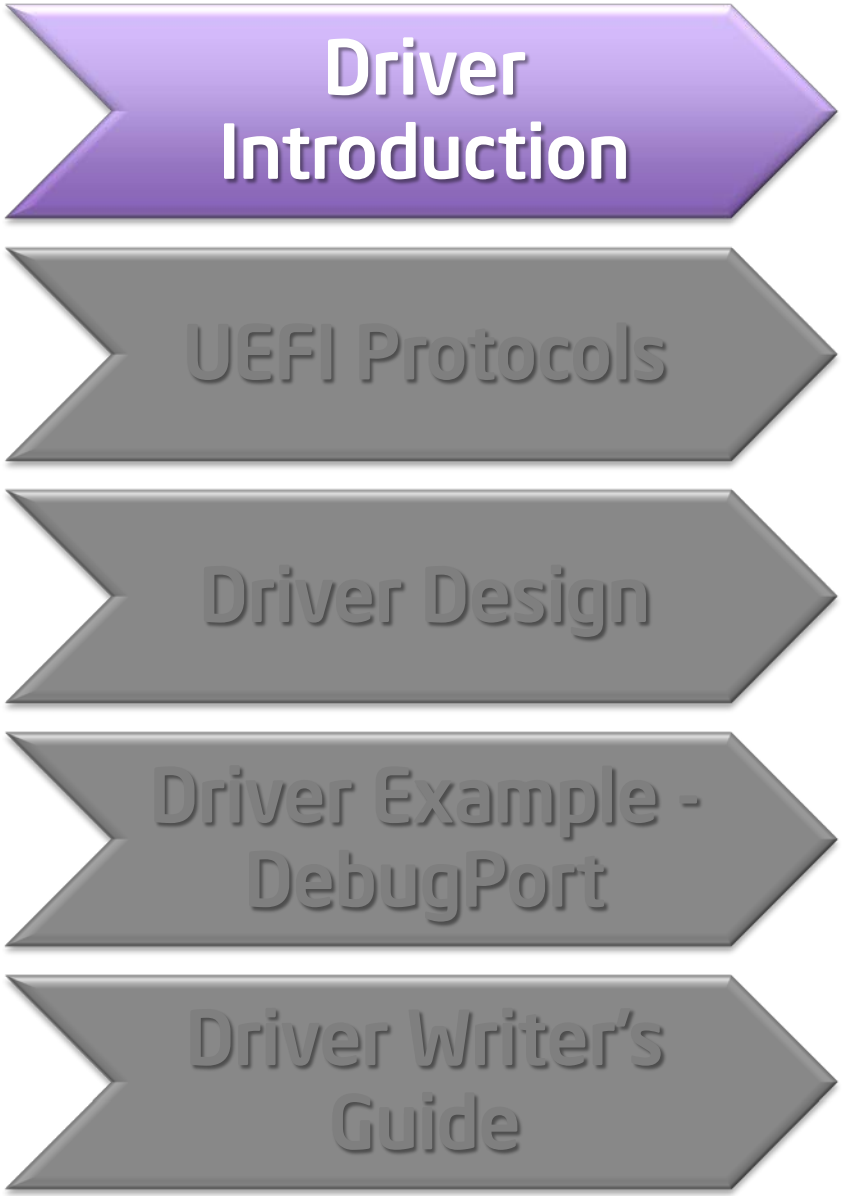
UEFI & EDK II Base Training
How to Write UEFI Drivers

Intel Corporation
Software and Services Group

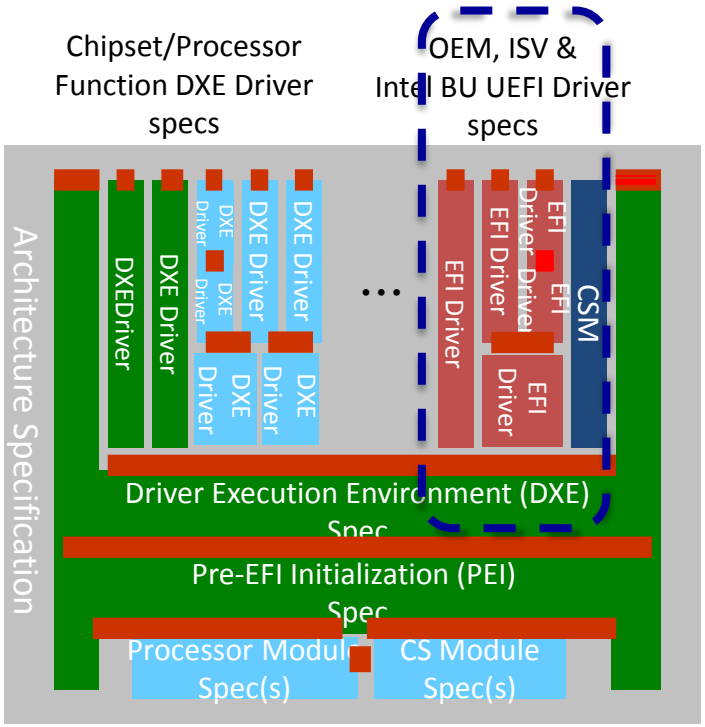
Agenda

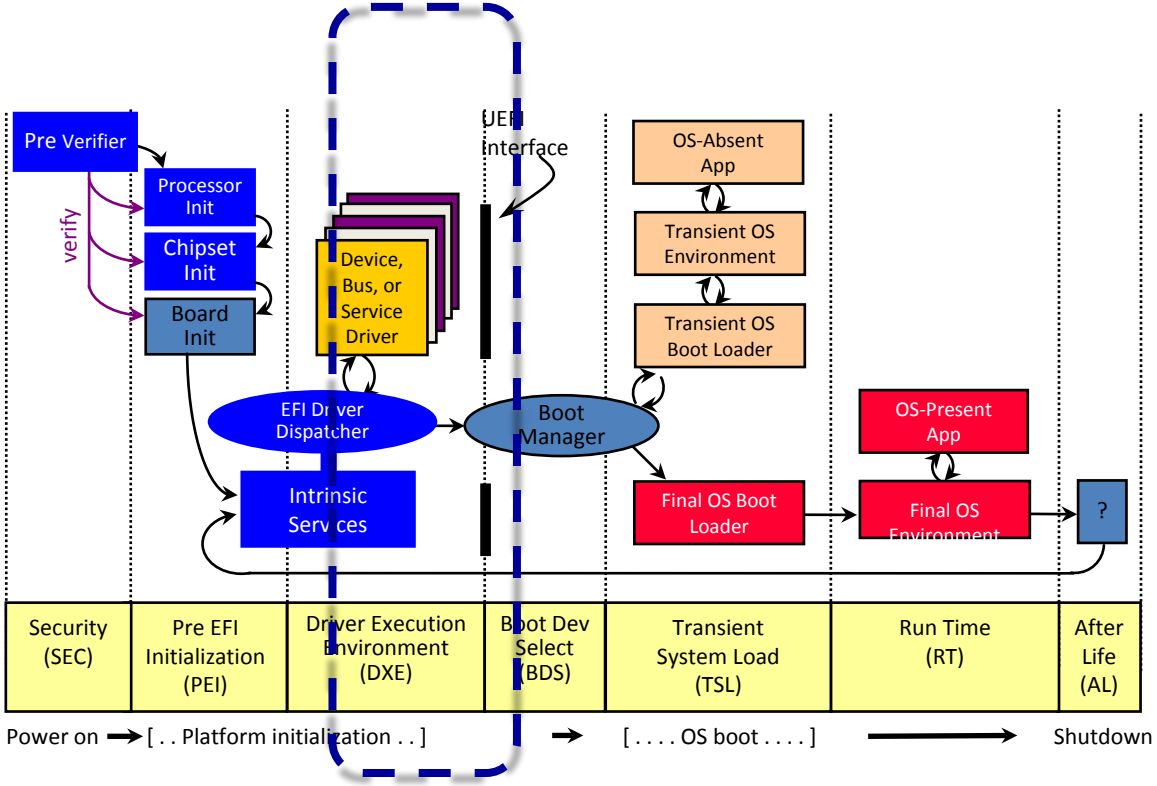

Driver Introduction
UEFI Protocols
Driver Design
Driver Example - DebugPort
Driver Writer's Guide

Agenda


Driver Introduction
UEFI Protocols
Driver Design
Driver Example - DebugPort
Driver Writer's Guide

UEFI Drivers

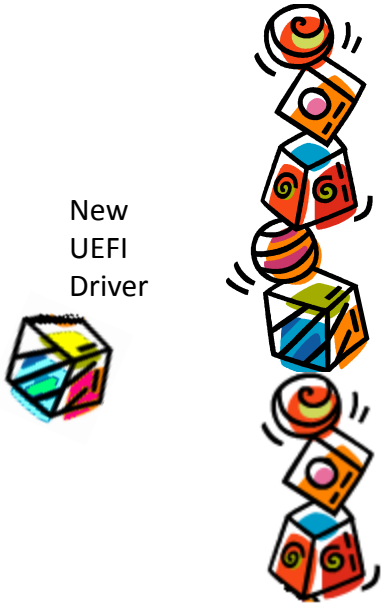




What do UEFI Drivers do?

- UEFI Drivers extend firmware
  - Add support for new hardware
  - No HW dependence
  - No OS dependence
- Portable across platforms
  - IA32, IA64, Intel-64/X64
- Enables rapid development
- Contents of a driver:
  - Entry Point
  - Published Functions
  - Consumed Functions
  - Data Structures

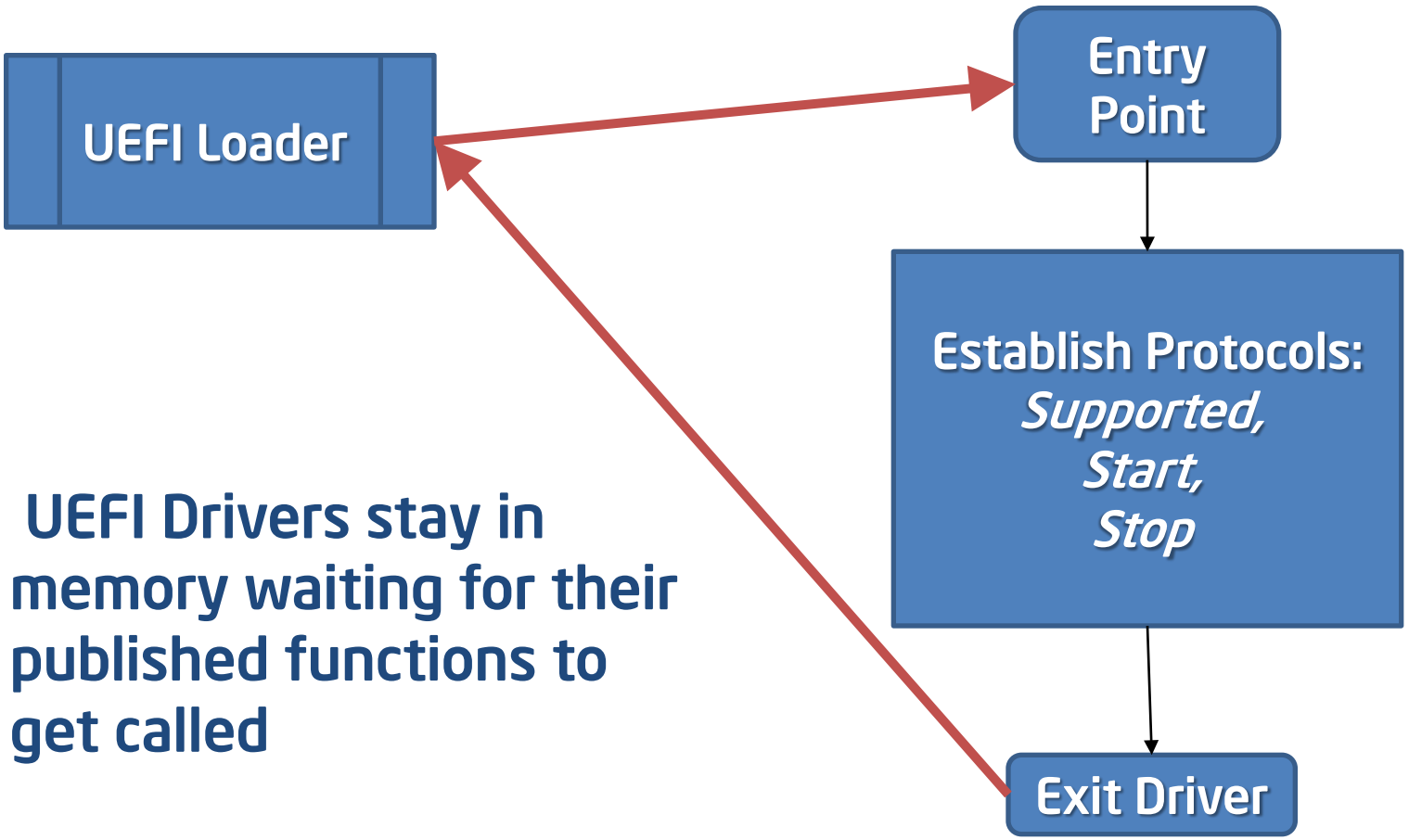
UEFI Driver Stack



What is an UEFI Driver?

- An UEFI Loadable Image
  - Loaded by UEFI loader
  - May produce protocols
  - May consume protocols
  - Typically system driven
- Can be used for …
  - Supporting specific hardware
  - Overriding an existing driver

General Driver Execution (entry)



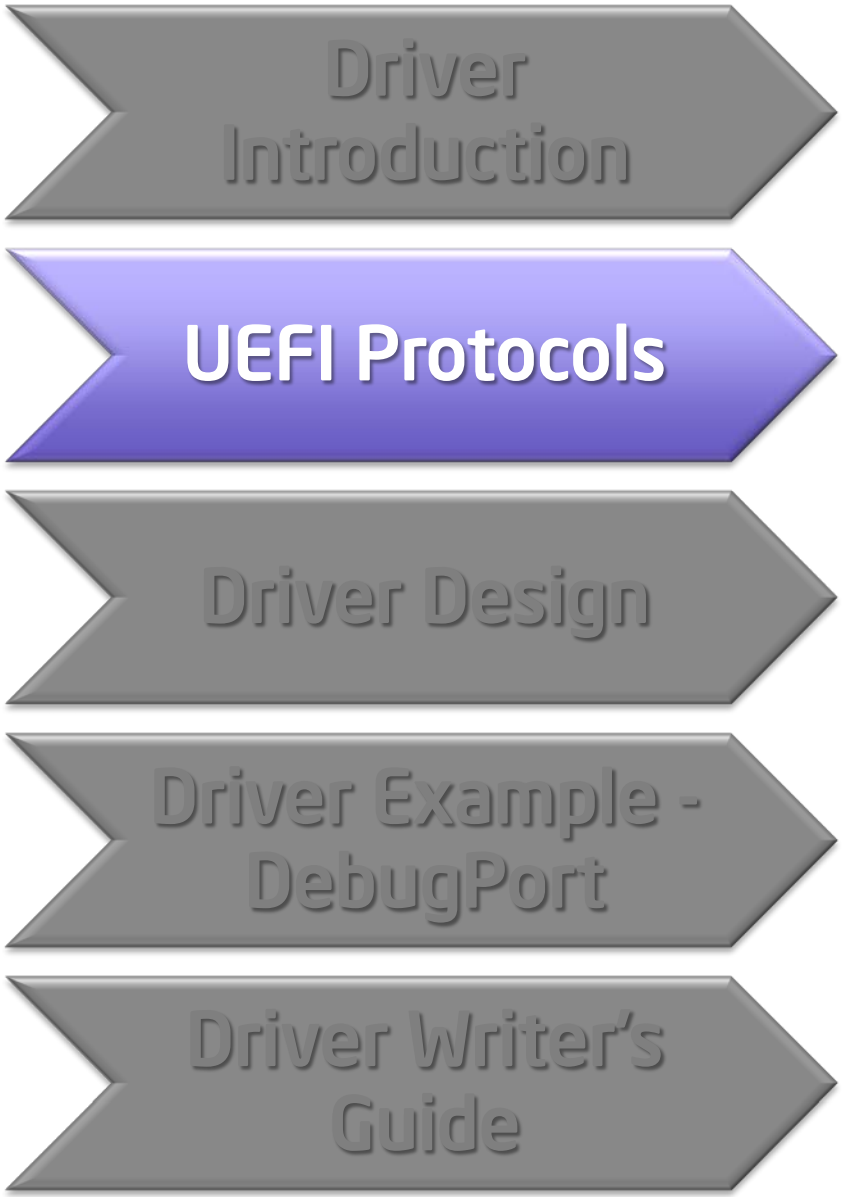
- UEFI Drivers stay in memory waiting for their published functions to get called

The UEFI Driver Model Specification

- Supports complex bus hierarchies
  - Follows the organization of physical/electrical architecture of the machine
- Driver Binding Protocol provides flexibility
  - Function to match drivers to devices
  - Driver version management
  - Hot-plug and unload support
- Drivers not tied to BIOS FLASH ROM
  - Can be loaded from UEFI System Partition
- Extensible - UEFI Drivers extend firmware(support future bus/device types)
  - Add support for new hardware
  - No dependence on OS or specific CPU architecture
  - Portable across platforms (IA32, x64, Itanium, …)
  - Reusable code leads to rapid platform development

See § 10 UEFI 2.X Spec.

Agenda

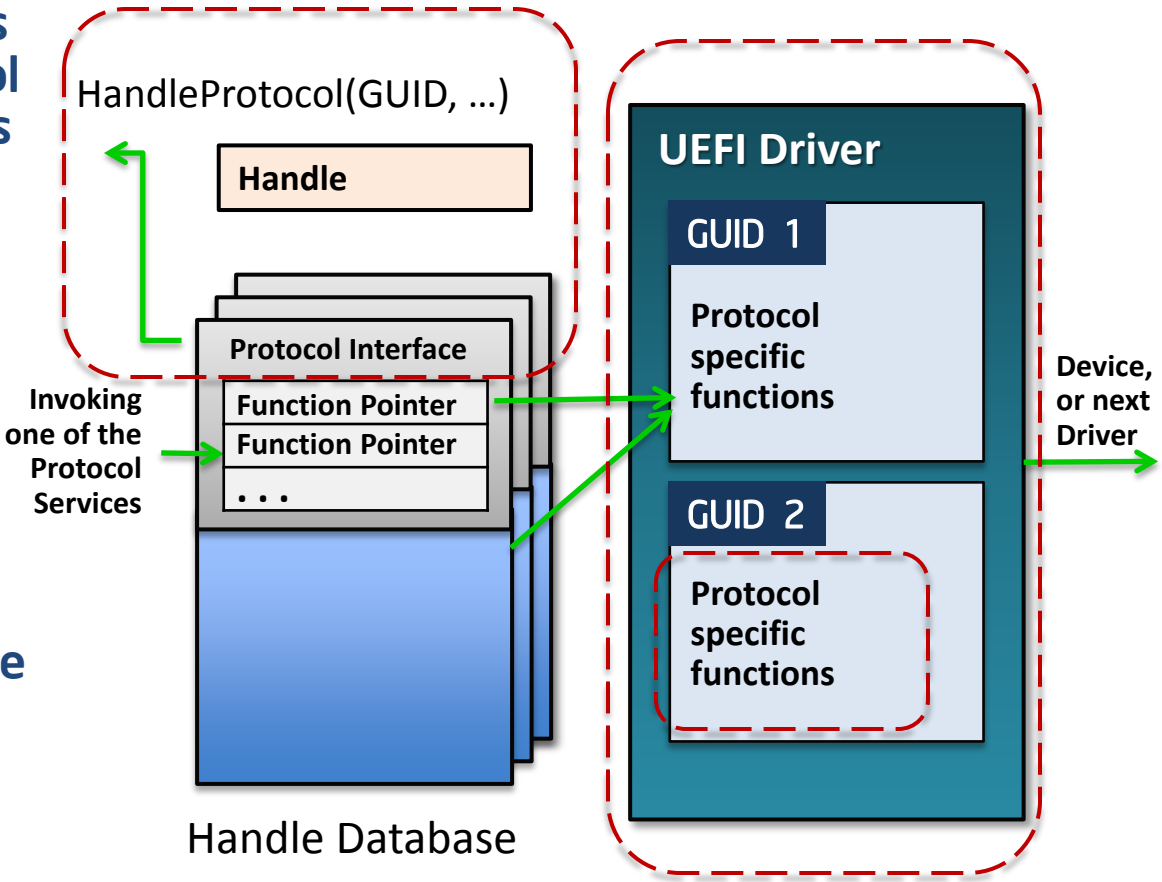

Driver Introduction
UEFI Protocols
Driver Design
Driver Example - DebugPort
Driver Writer's Guide

What is a protocol?

- A UEFI interface
  - Unique ID (GUID)
  - Protocol Interface structure
  - Protocol Services (functions)
- Must be Produced by a driver
- May be Consumed by anyone
- Set of related functions and associated data

Construction of a Protocol

- UEFI drivers contains functions specific to one or more protocol implementations, and registers them with the `InstallProtocolInterface()` Boot Service
- System firmware returns the Protocol Interface for the protocol that is then used to invoke the protocol specific service.
- The UEFI Driver keeps private, device specific context with the protocol interfaces.
- Examples:
  - Device Path, PCI I/O, Disk I/O, GOP, UNDI



See § 2.4 UEFI 2.x Spec.

Example: PCI I/O Protocol

GUID

```
#define EFI_PCI_IO_PROTOCOL_GUID \
{0x4cf5b200,0x68b8,0x4ca5,0x9e,0xec,0xb2,0x3e,0x3f,0x50,
0x2,0x9a}
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_IO_PROTOCOL {
    EFI_PCI_IO_PROTOCOL_POLL_IO_MEM PollMem;
    EFI_PCI_IO_PROTOCOL_POLL_IO_MEM PollIo;
    EFI_PCI_IO_PROTOCOL_ACCESS Mem;
    EFI_PCI_IO_PROTOCOL_ACCESS Io;
    EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS Pci;
    EFI_PCI_IO_PROTOCOL_COPY_MEM CopyMem;
    EFI_PCI_IO_PROTOCOL_MAP Map;
    EFI_PCI_IO_PROTOCOL_UNMAP Unmap;
    EFI_PCI_IO_PROTOCOL_ALLOCATE_BUFFER AllocateBuffer;
    EFI_PCI_IO_PROTOCOL_FREE_BUFFER FreeBuffer;
    EFI_PCI_IO_PROTOCOL_FLUSH Flush;
    EFI_PCI_IO_PROTOCOL_GET_LOCATION GetLocation;
    EFI_PCI_IO_PROTOCOL_ATTRIBUTES Attributes;
    EFI_PCI_IO_PROTOCOL_GET_BAR_ATTRIBUTES GetBarAttributes;
    EFI_PCI_IO_PROTOCOL_SET_BAR_ATTRIBUTES SetBarAttributes;
    UINT64 RomSize;
    VOID *RomImage;
} EFI_PCI_IO_PROTOCOL
```

See § 13.4 UEFI 2.x Spec.

Example: DebugPort Protocol

GUID

```
#define EFI_DEBUGPORT_PROTOCOL_GUID \
{0xEBA4E8D2,0x3858,0x41EC,0xA2,0x81,0x26,0x47,\
  0xBA,0x96,0x60,0xD0 }
```

Protocol Interface Structure

```
typedef struct _EFI_DEBUGPORT_PROTOCOL {
    EFI_DEBUGPORT_RESET Reset;
    EFI_DEBUGPORT_WRITE Write;
    EFI_DEBUGPORT_READ Read;
    EFI_DEBUGPORT_POLL Poll;
} EFI_DEBUGPORT_PROTOCOL
```

- The Debugport protocol is used for byte stream communication with a debugport device. The debugport can be a standard UART Serial port, a USB-based character device, or potentially any character-based I/O device.

See § 17.3 UEFI 2.x Spec.

UEFI Runs on Protocols …

EFI_GRAPHICS_OUTPUT_PROTOCOL

EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL EFI_BLOCK_IO_PROTOCOL

EFI_DRIVER_HEALTH_PROTOCOL

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL

EFI_DRIVER_DIAGNOSTICS2_PROTOCOL

EFI_COMPONENT_NAME2_PROTOCOL EFI_BLOCK_IO_PROTOCOL

EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL

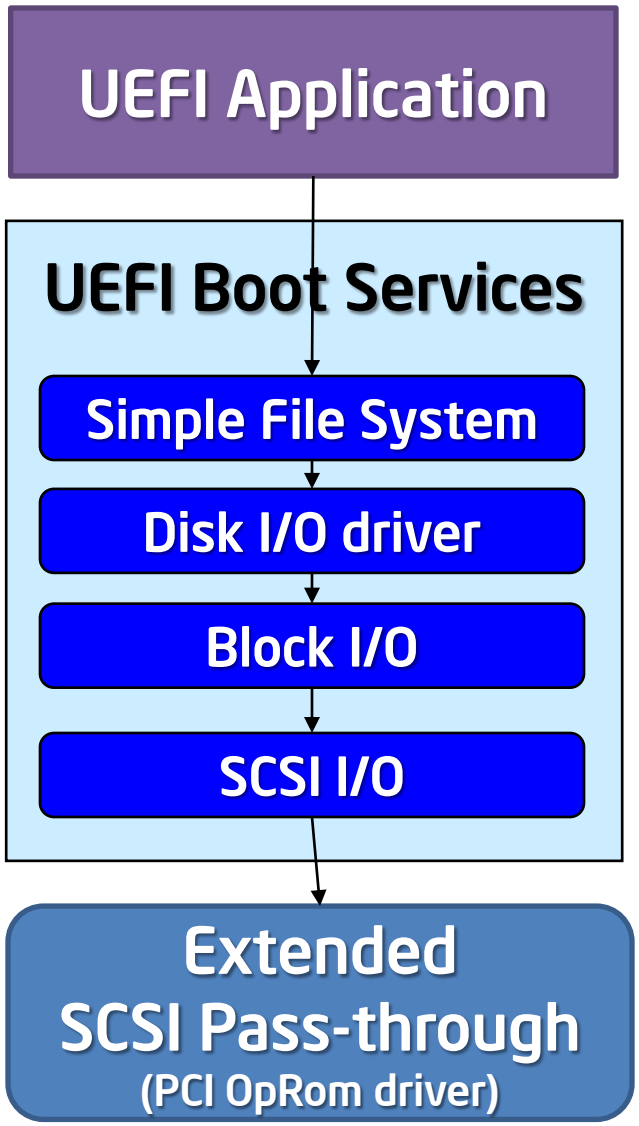
EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL

EFI_GRAPHICS_OUTPUT_PROTOCOL

EFI_DRIVER_BINDING_PROTOCOL EFI_ATA_PASS_THRU_PROTOCOL

EFI_EXT_SCSI_PASS_THRU_PROTOCOL

EFI_FIRMWARE_MANAGEMENT_PROTOCOL



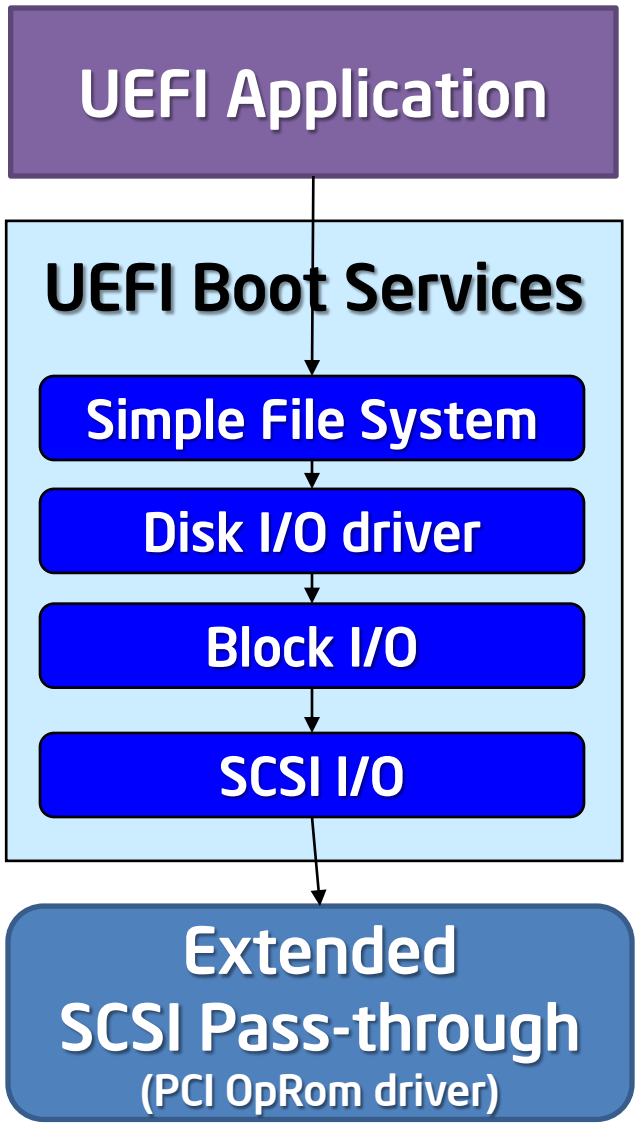
Protocol Example

Application consumes I/O protocol,
firmware produces protocol
(exposed through boot services)

Firmware uses correct lower level protocols,
consumes extended SCSI pass-through protocol.
SCSI card driver produces extended SCSI pass-through protocol

SCSI card driver will talk directly to the SCSI drive

See back up for Example of UEFI 2.x Network stack



Protocol Example

Application consumes I/O protocol, firmware produces protocol (exposed through boot services)

The application has to determine which top level handle to call into

Firmware uses correct lower level protocols, consumes extended SCSI pass-through protocol. SCSI card driver produces extended SCSI pass-through protocol

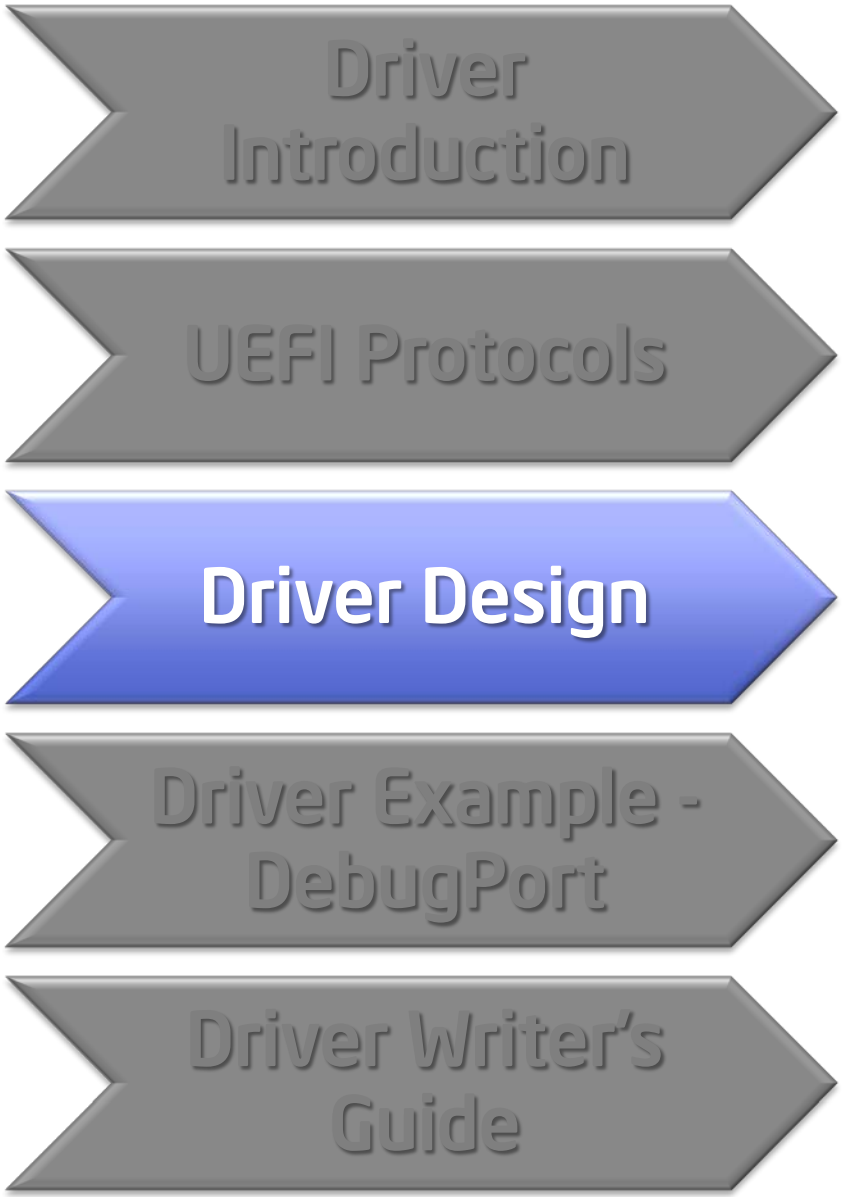
SCSI card driver will talk directly to the SCSI drive

See back up for Example of UEFI 2.x Network stack

UEFI Protocol vs. C++ Class

- *An UEFI Protocol is logically similar to a C++ Class, with similarities …*
  - Has private member variables
  - Has exposed functions
  - Has private functions
  - Has a 'This' pointer
- *Practical look & feel is very different between the UEFI Protocol and C++ Class*
- *Some differences*
  - Lower memory overhead
    - No virtual function table
  - Different code can produce the same protocol
  - SCSI driver, IDE driver can produce same abstraction
  - A piece of code can produce more than one protocol
  - Dynamically bound

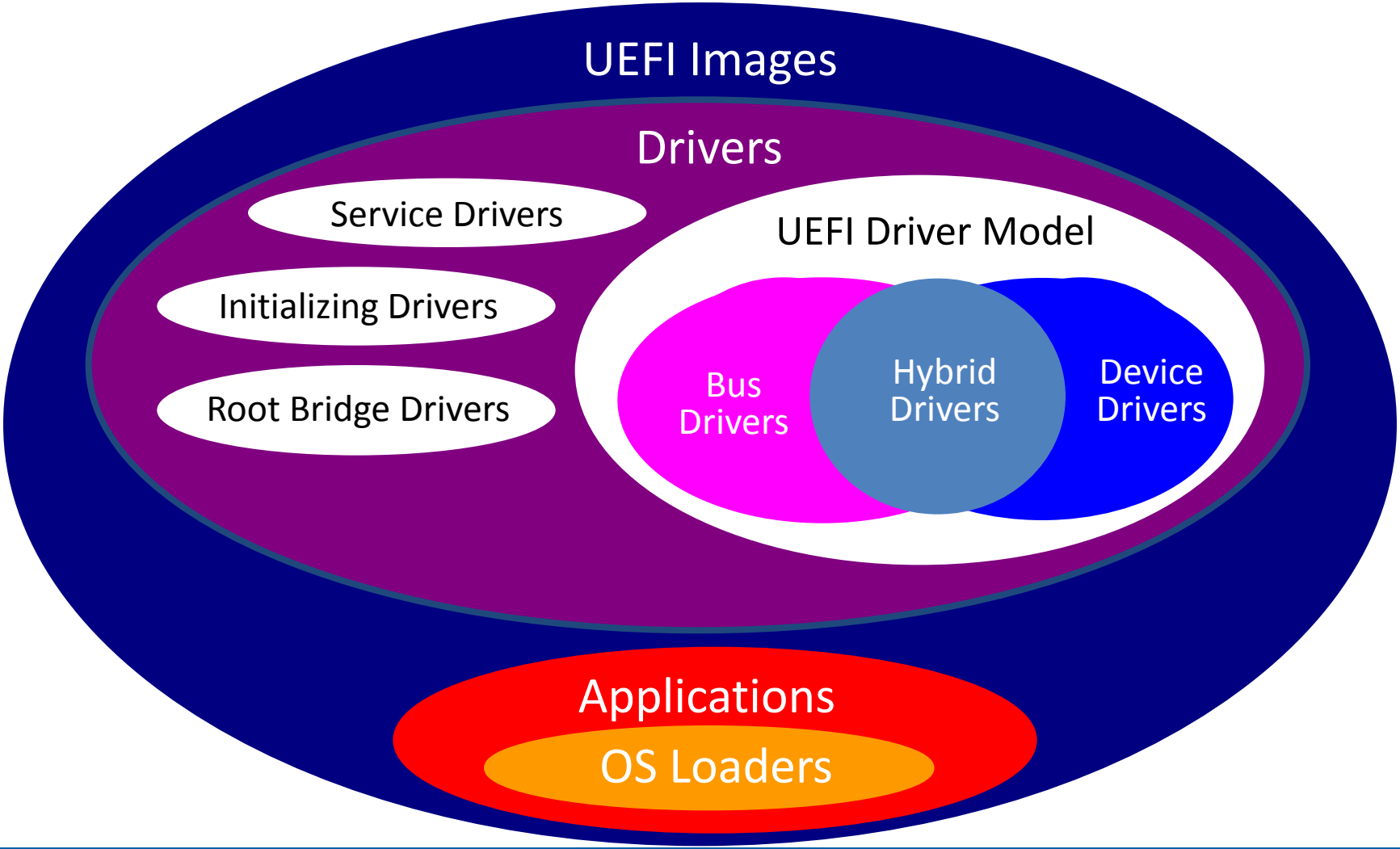
Agenda


Driver Introduction
UEFI Protocols
Driver Design
Driver Example - DebugPort
Driver Writer's Guide

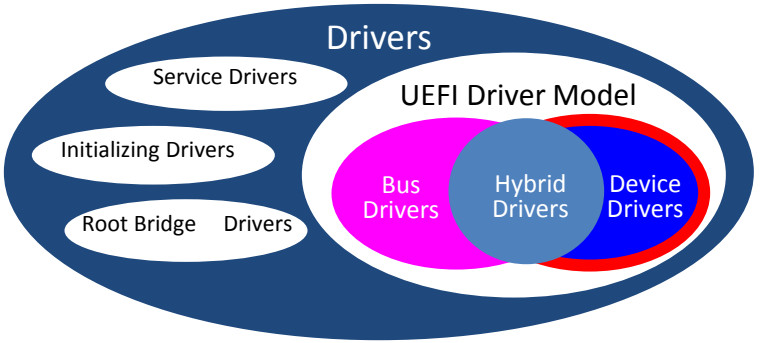
Steps to Design a UEFI Driver

1. Determine Driver Type
2. Identify Consumed I/O Protocols
3. Identify Produced I/O Protocols
4. Identify UEFI Driver Model Protocols
5. Identify Additional Driver Features
6. Identify Target Platforms
   - x86 (IA32)
   - Intel® 64 (x64)
   - Intel Itanium® Processor Family
   - EFI Byte Code (EBC)

What Type of Driver is Being Designed?



Device Drivers

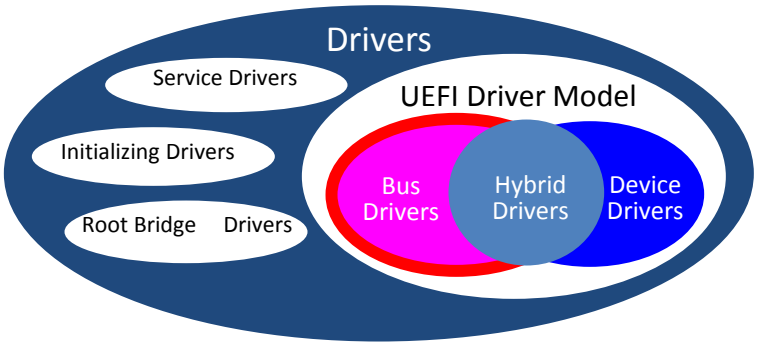


- Manages a Controller or Peripheral Device
- Start() Does Not Create Any Child Handles
- Start() Produces One or More I/O Protocols
  - Installed onto the Device's Controller Handle

Examples:

PCI Video Adapters
USB Host Controllers
USB Keyboards / USB Mice
PS/2 Keyboards / PS/2 Mice

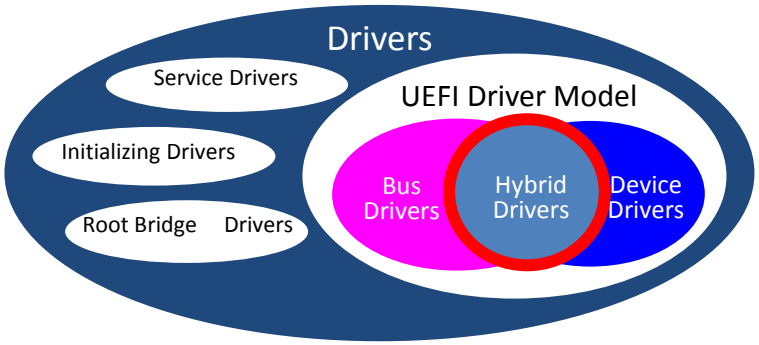
Bus Drivers



- Manages and Enumerates a Bus Controller
- Start() Creates One or More Child Handles
- Start() Produces Bus Specific I/O Protocols
  - Installed onto the Bus's Child Handles

Examples:
PCI Network Interface Controllers
Serial UART Controllers

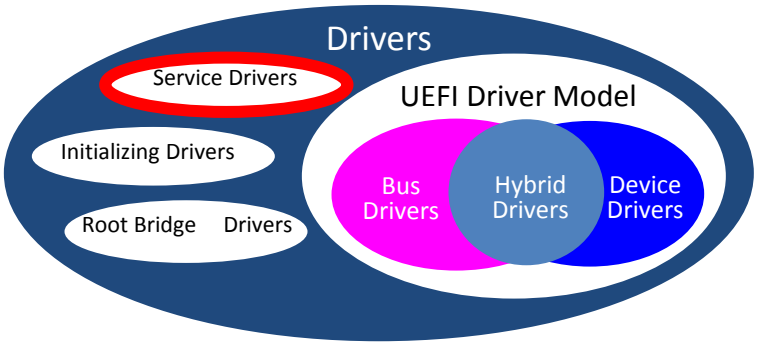
Hybrid Drivers



- Manages and Enumerates a Bus Controller
- Start() Creates One or More Child Handles
- Start() Produces Bus Specific I/O Protocols
  - Installed onto the Bus's Controller Handle
  - Installed onto Bus's Child Handles

Examples:
PCI SCSI Host Controllers
PCI Fiber Channel Controllers

Service Drivers



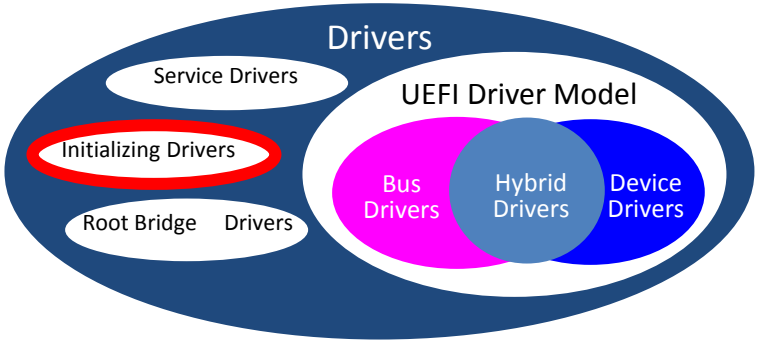
- Does Not Manage Hardware
- Provides Services to other Drivers
- Does not support Driver Binding Protocol
- Typically installs protocols in driver entry point
- Creates One or More Service Handles
- Produces Service Specific Protocols
  - Installed onto Service Handles

Examples:

UEFI Decompress Protocol

UEFI Byte Code Virtual Machine

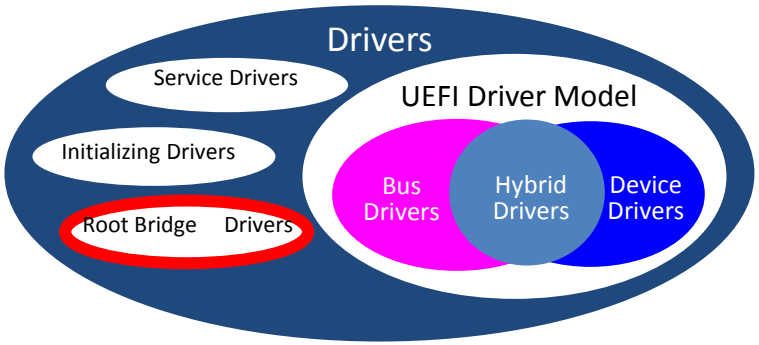
Initializing Drivers



- Typically Touches Hardware
- Performs One Time Initialization Operations
- Does Not Create Any Handles
- Does Not Produce Any Protocols
- Unloaded When Finished

Examples: UEFI None (PI PEI or DXE Driver)

Root Bridge Drivers



- Typically Manages Part of Core Chipset
- Directly Touches Hardware
- Creates One or More Root Bridge Handles
- Produces Root Bridge I/O Protocols
  - Installed onto new Root Bridge Handles

Examples: PCI Host Bridge

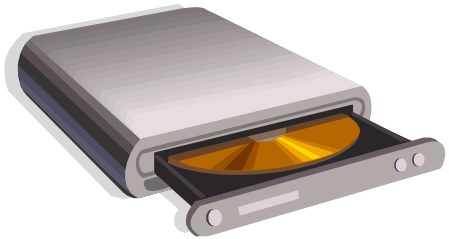
What I/O Protocols are Consumed?

PCI Adapters

- PCI I/O Protocol
- Device Path Protocol

USB Peripherals

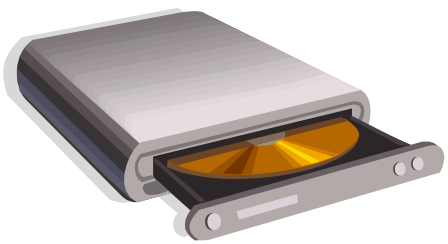
- USB I/O Protocol
- Device Path Protocol



What I/O Protocols are Produced?

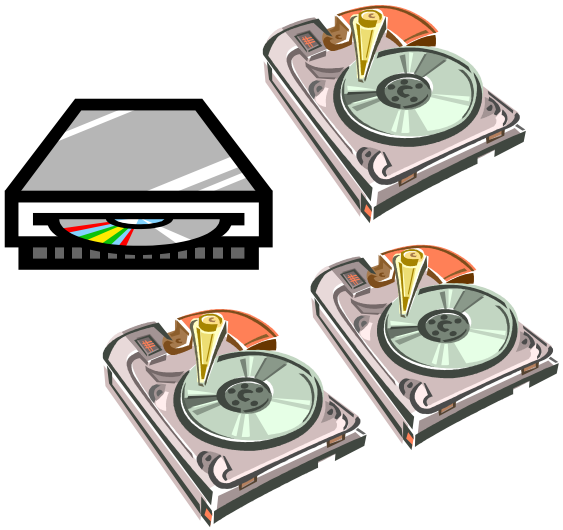
Simple Input Protocol

Simple Pointer Protocol



Block I/O Protocol

What I/O Protocols are Produced?



SCSI
SCSI RAID
Fiber Channel

- Extended SCSI Pass Thru Protocol and
- Block I/O Protocol

What I/O Protocols are Produced?

Network Interface Controller (NIC)

Depends on the NIC …

1. Universal Network Driver Interface (UNDI) & Network Interface Identifier Protocol (NII)

OR

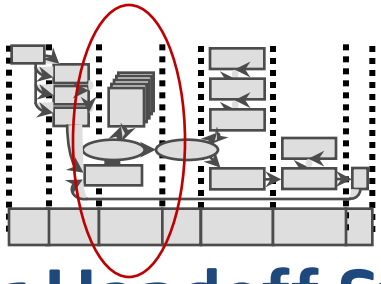
2. Managed Network Protocol (MNP)
3. Simple Network Protocol (SNP)

Examples: Intel UNDI (x64) GB drivers for EDK I found:
https://sourceforge.net/projects/efidevkit/files/Releases/Others/Other%20Contribution/
EDK II example: OptionRomPkg\UndiRuntimeDxe (32bit Driver)

See back-up for Network stack examples

How to Write a UEFI Driver

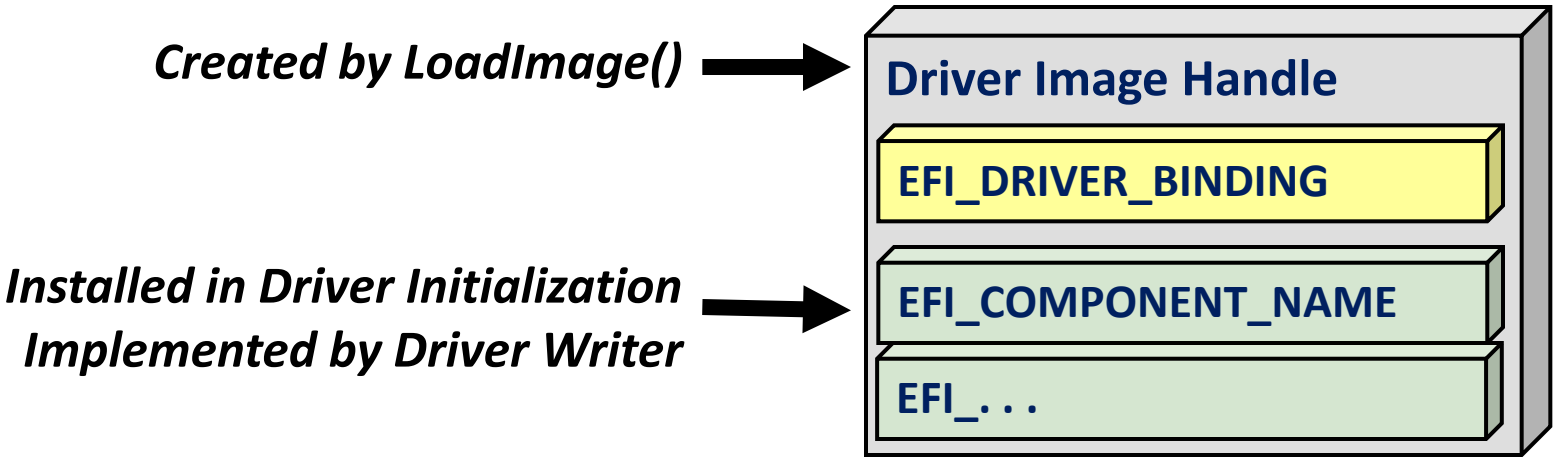
- Initialization
- Binding
  - Supported
  - Start
  - Stop
- Component name
- Configuration
- Diagnostic
- Driver Health
- Unload



Platform Initialization Spec

Driver Initialization

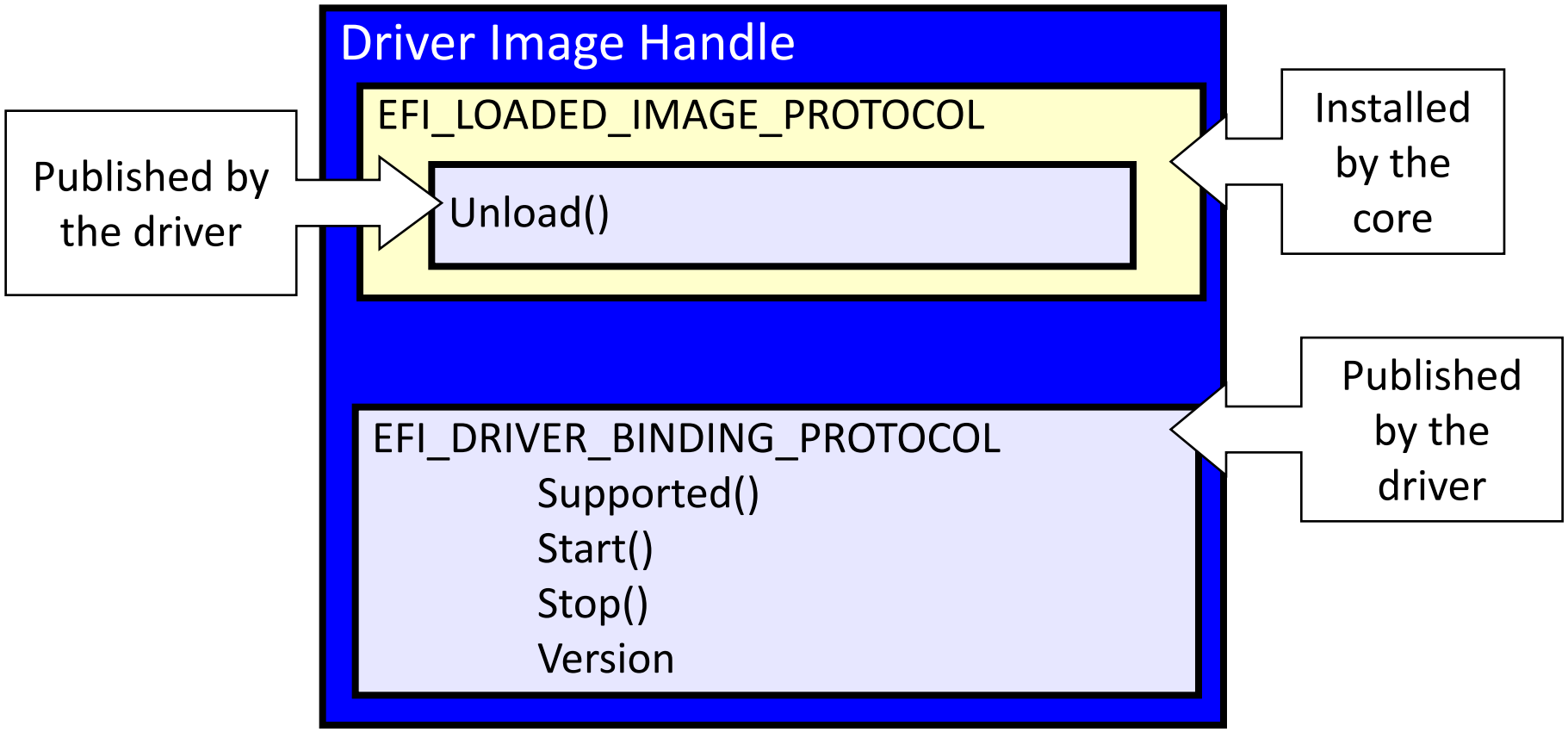
- UEFI Driver Handoff State
- Not Allowed to Touch Hardware Resources
- Installs Driver Binding on Driver Image Handle



Registers Driver for Later Use

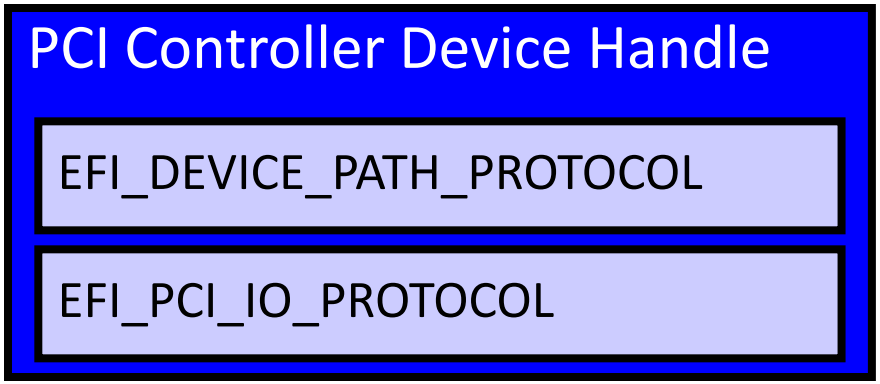
Responsibilities of Driver Writer

- **Driver Image Handle Required Protocols**



See § 2.5.2 UEFI 2.x Spec.

Supported - PCI Controller Device Handle

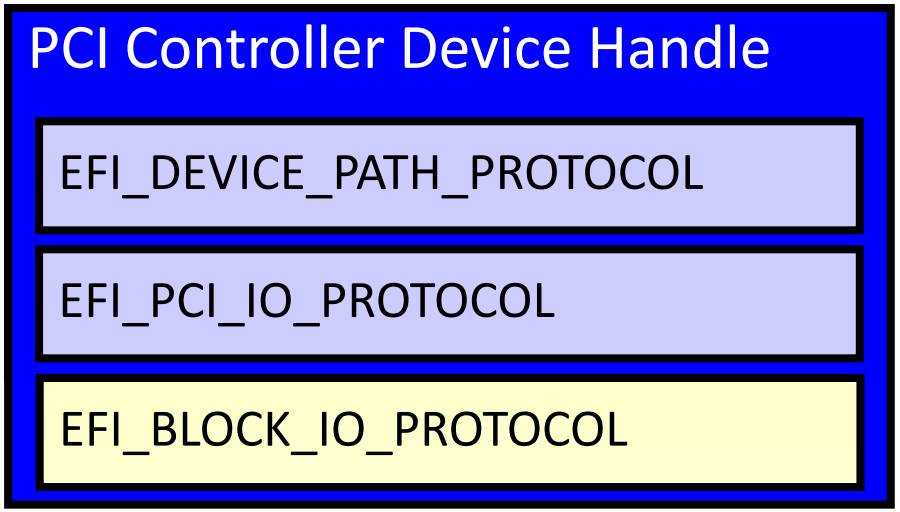


1. Opens PCI_IO Protocol
2. Checks
3. Closes PCI_IO Protocol
4. Returns: *Supported* or *Not Supported*

- Inputs:
  - "This"
  - Controller to manage
  - Remaining Device Path
- Supported()
  - Checks to see if a driver supports a controller
  - Check should not change hardware state of controller
  - Minimize execution time, move complex I/O to Start()
  - May be called for controller that is already managed
  - Child is optionally specified

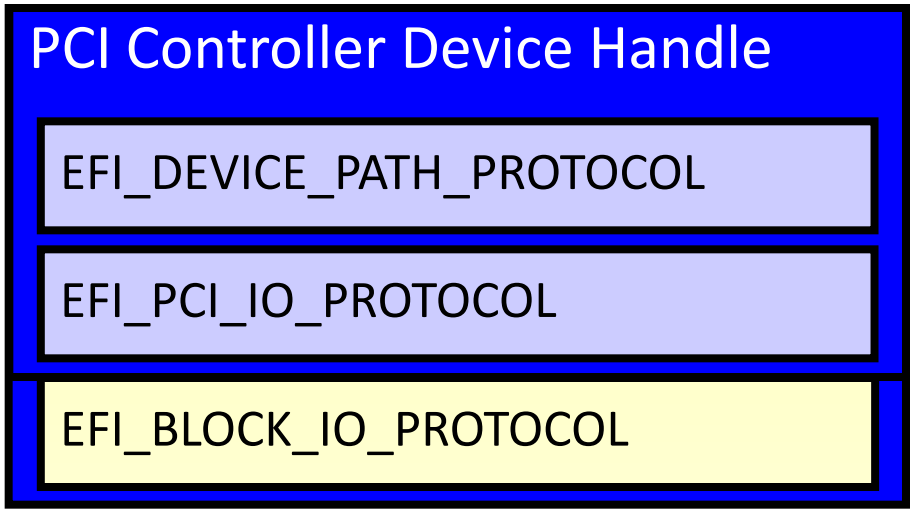
See § 10.1 UEFI 2.x Spec.

Start - PCI Controller Device Handle



- Inputs:
  - "This"
  - Controller to manage,
  - Remaining Device Path
- Start()
  - *Opens* PCI I/O
  - Starts a driver on a controller
  - Can create ALL child handles or ONE child handle

Stop - PCI Controller Device Handle



- Inputs:
  - "This"
  - Controller to manage,
  - Remaining Device Path

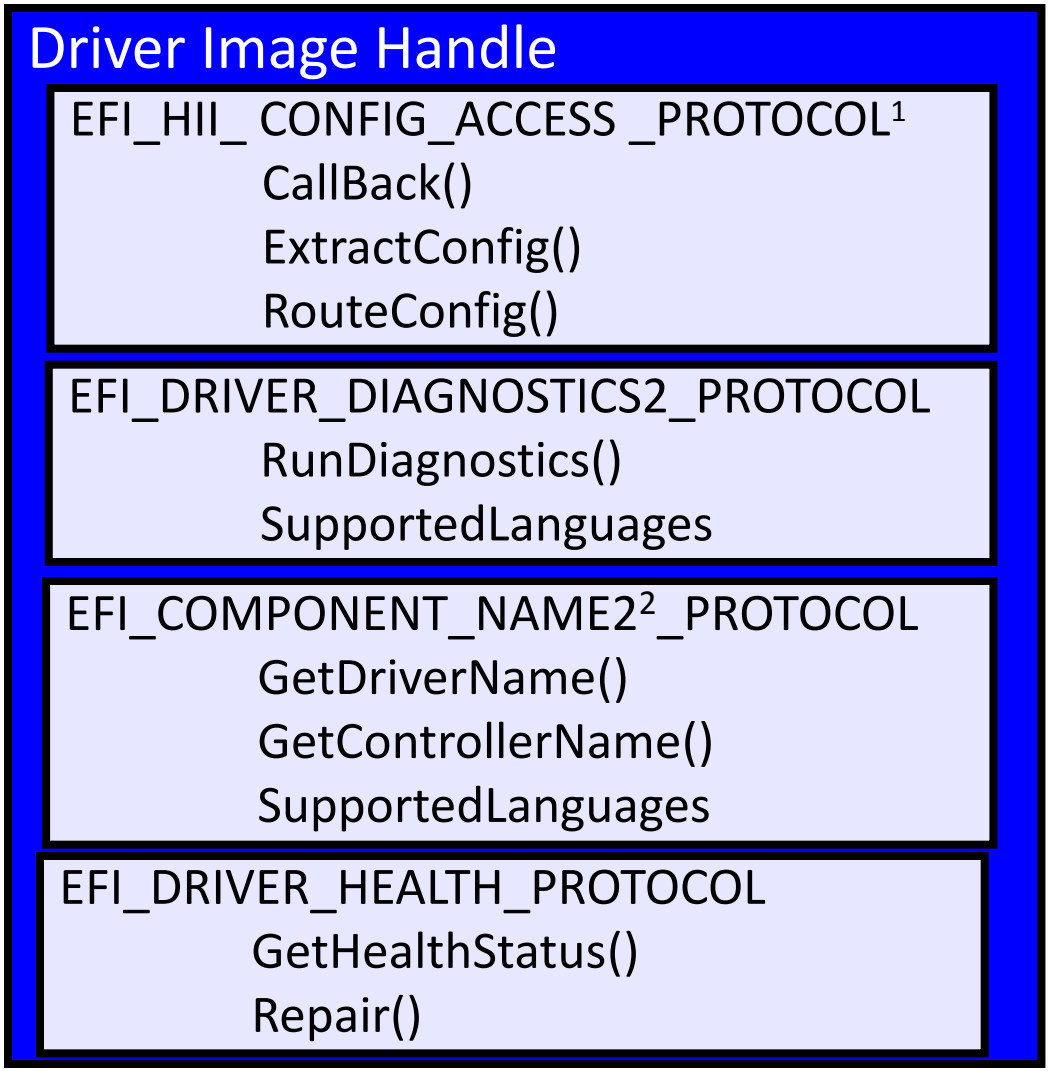
- Stop()
  - *Closes* PCI I/O
  - Stops a driver from managing a controller
  - Destroys all specified child handles
  - If no children specified, controller is stopped
  - Stopping a bus controller requires 2 calls
    - One call to stop the children. A second call to stop the bus controller itself

More Responsibilities of the Driver Writer

UEFI Driver Model

Optional

- Driver Image Handle Protocols



[1] EFI_HII_ CONFIG_ACCESS _PROTOCOL was added in the UEFI Spec 2.1 for better HII support

[2] EFI_COMPONENT_NAME2_PROTOCOL was added to UEFI Spec 2.1 for Localization support REC 4646 vs. ISO 639-2 language codes

See § 10 UEFI 2.x Spec.

Driver Design Checklist

	PCI Video	PCI RAID	PCI NIC
Driver Type	Device	Hybrid	Bus
I/O Protocols Consumed	PCI I/O	PCI I/O Device Path	PCI I/O Device Path
I/O Protocols Produced	GOP	EXT SCSI Pass Thru Block I/O	UNDI, NII[2]
Driver Binding	✓	✓	✓
Component Name	✓	✓	✓
Driver HII Function Configuration		✓	
Driver Diagnostics	✓	✓	✓
Unloadable	✓	✓	✓
Exit Boot Services Event			✓
Runtime			✓
Set Virtual Address Map Event[1]			✓

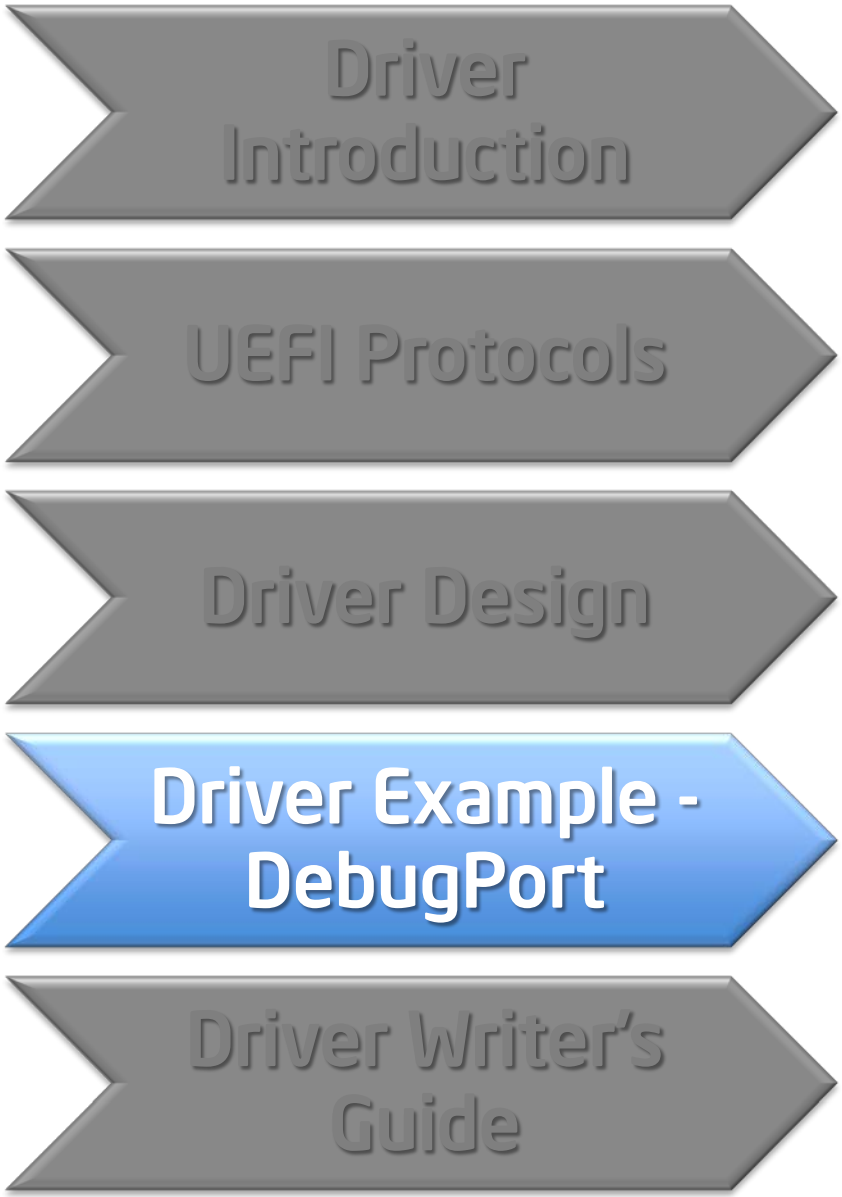
[1] See § 7.4 UEFI 2.x Spec

[2] Depends on the NIC Card Vendor

Running UEFI Drivers

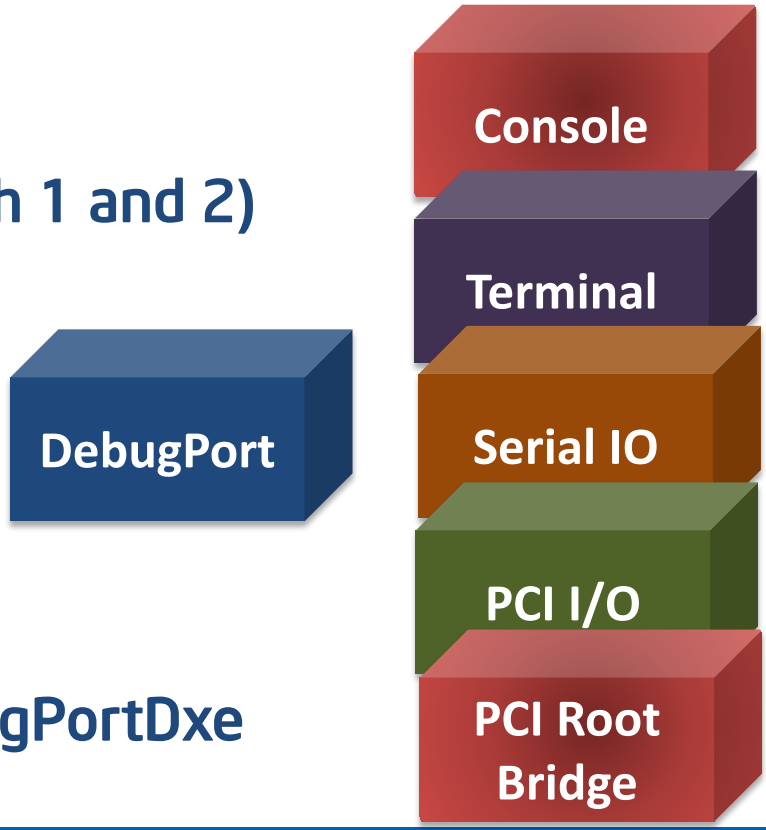
- `ConnectController()`
  - Called from Boot Manager or during load
  - Precedence rules are applied
    - Context override
    - Platform override
    - Bus override
    - Version number
  - Order of which drivers are installed into handle database is not deterministic
- `DisconnectController()`
  - Must test and implement `Stop()`

Agenda

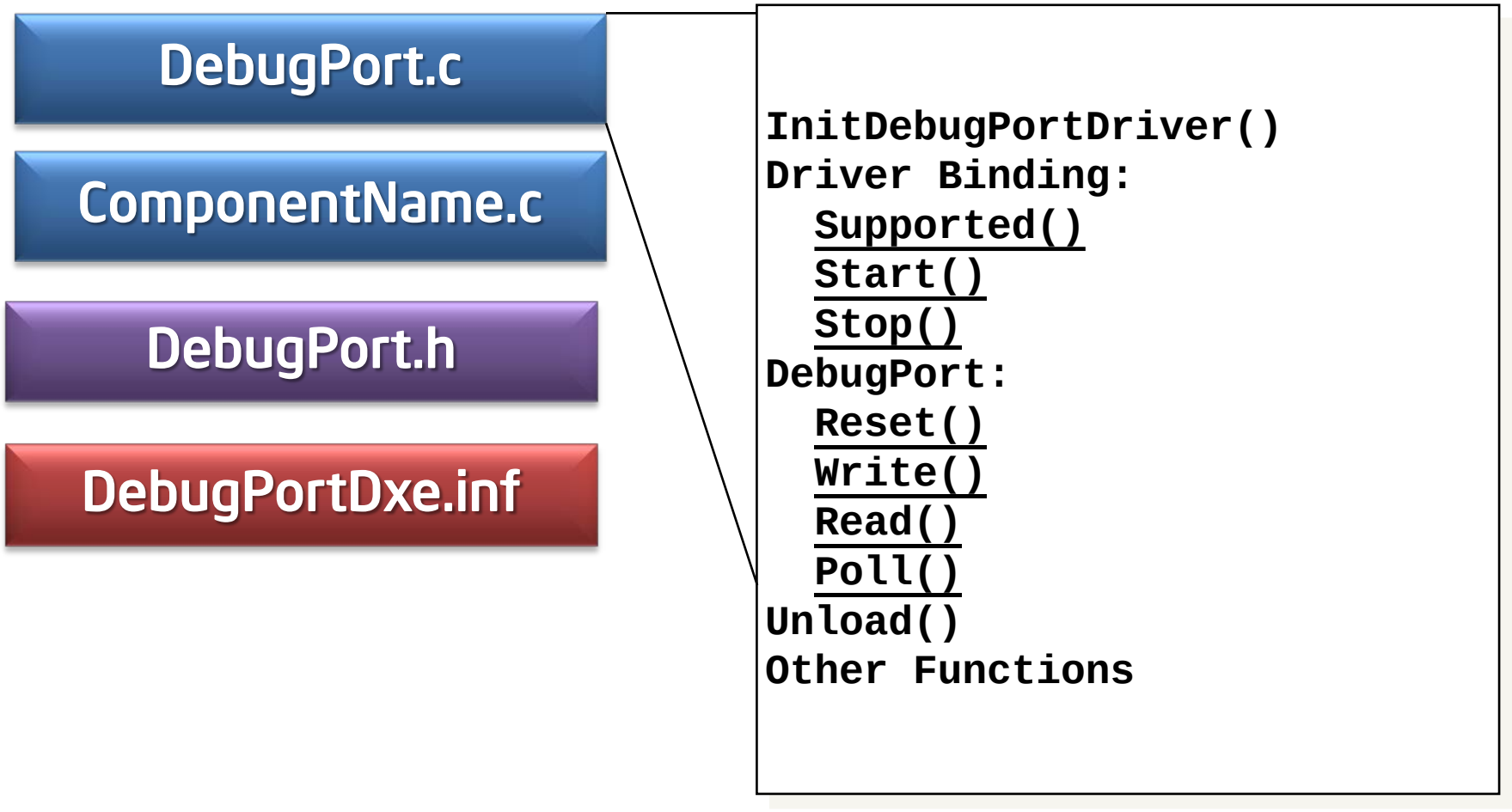

Driver Introduction
UEFI Protocols
Driver Design
Driver Example - DebugPort
Driver Writer's Guide

UEFI Debug Port Driver

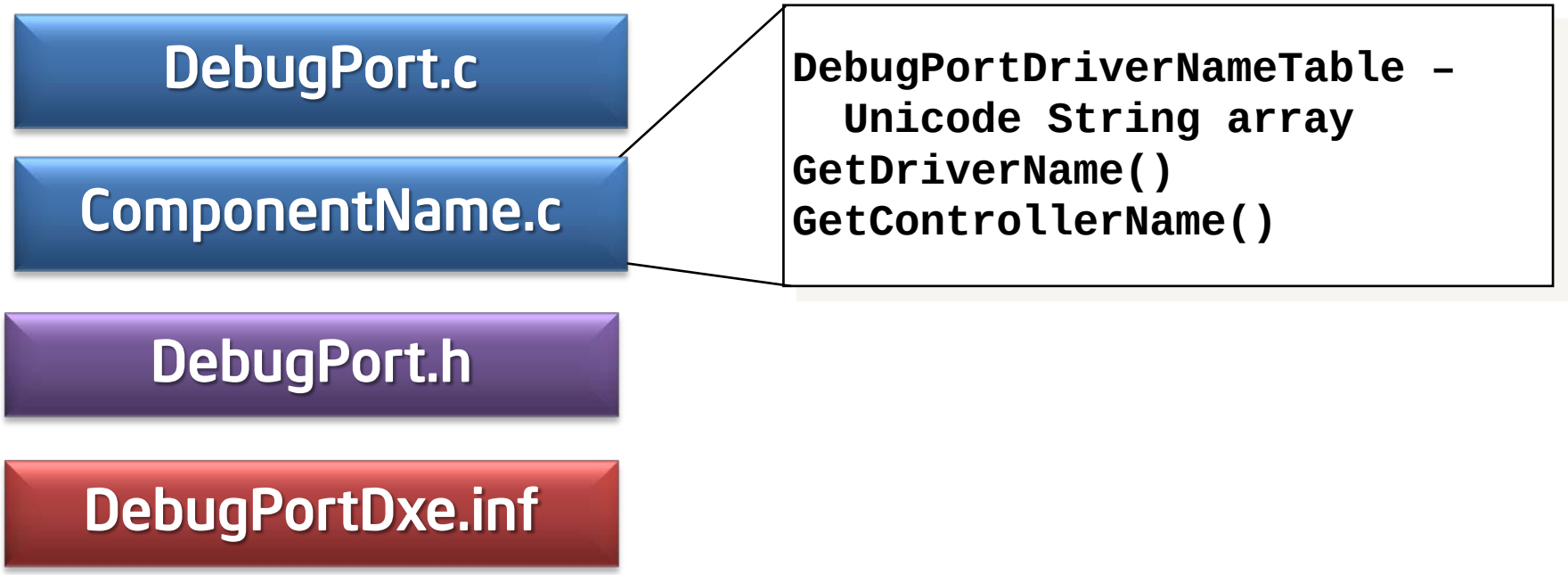
- The DebugPort Driver uses the Debugport protocol is for byte stream communication with a Debugport device. The Debugport driver example will use a standard UART Serial port
- UEFI Driver Produces
  - Driver Binding Protocol
  - Component Name Protocol (both 1 and 2)
  - DebugPort Protocol
  - Optional – Unload function
- UEFI Driver Consumes:
  - Serial IO Protocol
  - Device Path Protocol
- Example Location EDK II:
  - MdeModulePkg\Universal\DebugPortDxe



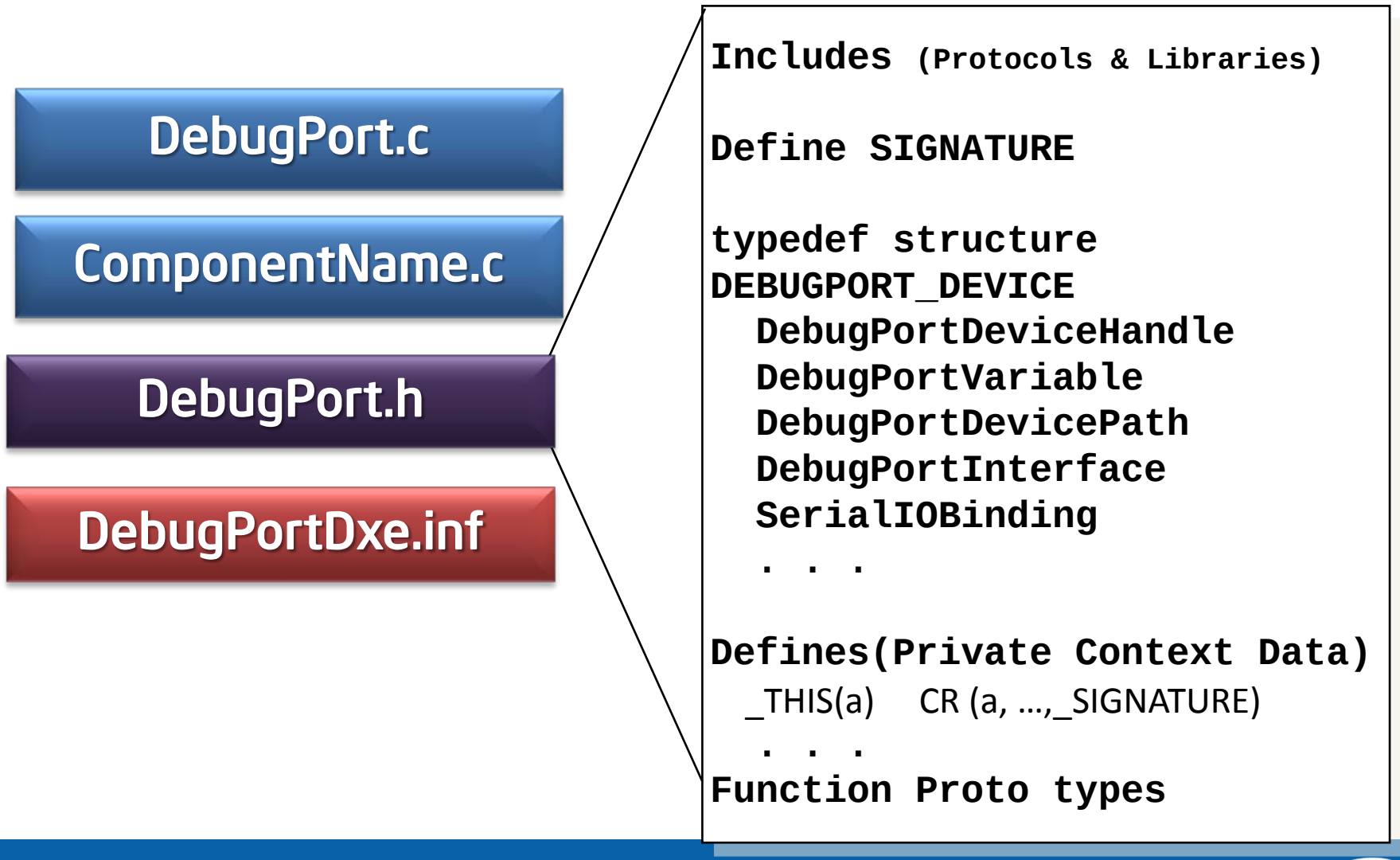
DebugPort Driver Function Layout



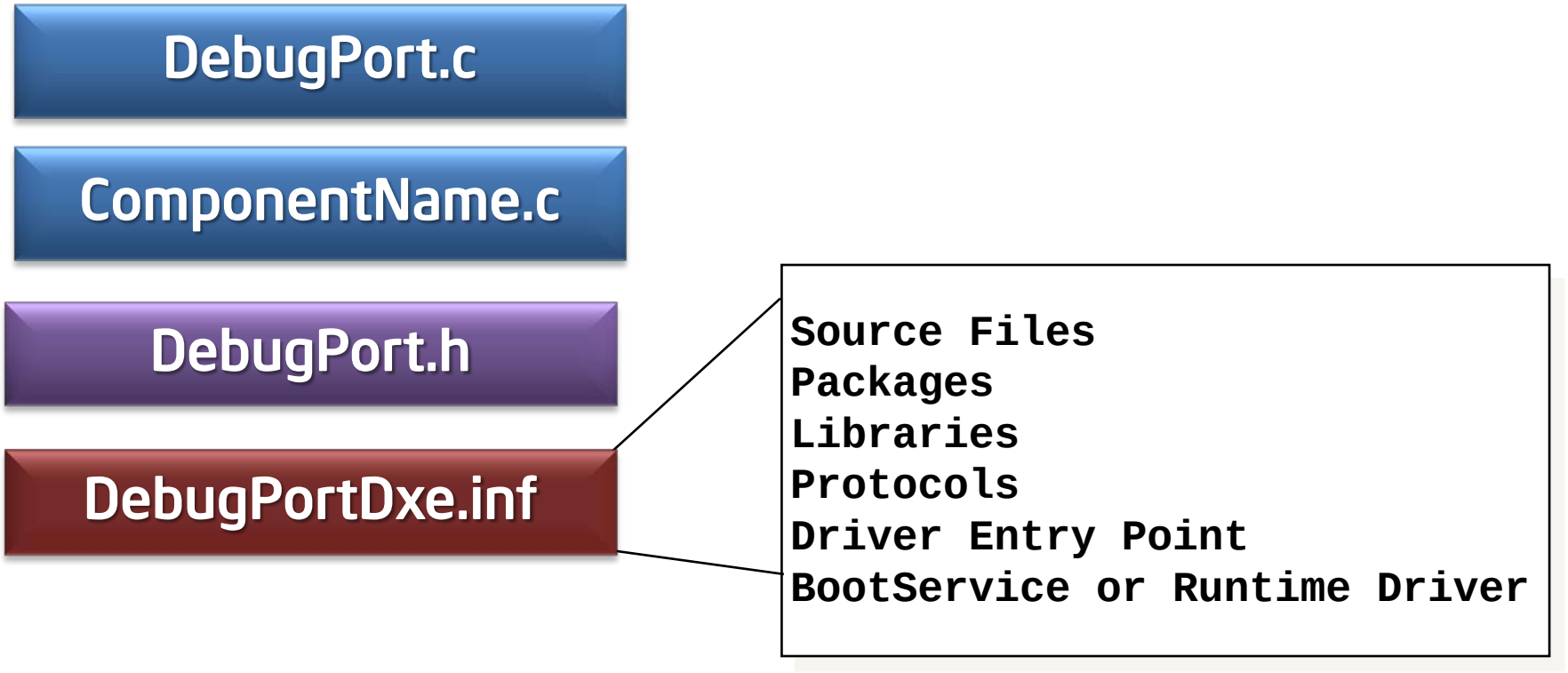
DebugPort Driver Function Layout



DebugPort Driver Function Layout



DebugPort Driver Function Layout



DebugPort.h Source

```
//  Include statements
#include <Uefi.h>
#include <Protocol/DevicePath.h>
#include <Protocol/ComponentName.h>
#include <Protocol/DriverBinding.h>
#include <Protocol/SerialIo.h>
#include <Protocol/DebugPort.h>
// . . .

// Signature
#define DEBUGPORT_DEVICE_SIGNATURE \
   SIGNATURE_32 ('D', 'B', 'G', 'P')
```

```
//
// Device structure used by driver
//
typedef struct {
  UINT32                Signature;
  EFI_HANDLE            DriverBindingHandle;
  EFI_HANDLE            DebugPortDeviceHandle;
  VOID                  *DebugPortVariable;

  EFI_DEVICE_PATH_PROTOCOL
       *DebugPortDevicePath;
  EFI_DEBUGPORT_PROTOCOL
       DebugPortInterface;
  EFI_HANDLE              SerialIoDeviceHandle;
  EFI_SERIAL_IO_PROTOCOL
      *SerialIoBinding;
  UINT64                  BaudRate;
  UINT32                  ReceiveFifoDepth;
  UINT32                  Timeout;
  EFI_PARITY_TYPE         Parity;
  UINT8                   DataBits;
  EFI_STOP_BITS_TYPE      StopBits;
} DEBUGPORT_DEVICE;
```

```
// Private Context Data
#define DEBUGPORT_DEVICE_FROM_THIS(a) CR (a, DEBUGPORT_DEVICE,
                                      DebugPortInterface, DEBUGPORT_DEVICE_SIGNATURE)
```

DebugPort.c Source- Variables

```
#include "DebugPort.h"
// Globals
EFI_DRIVER_BINDING_PROTOCOL
    gDebugPortDriverBinding = {
  DebugPortSupported,
  DebugPortStart,
  DebugPortStop,
  DEBUGPORT_DRIVER_VERSION,
  NULL,
  NULL
};

DEBUGPORT_DEVICE mDebugPortDevice = {
  DEBUGPORT_DEVICE_SIGNATURE,
   .  .  .
  (EFI_DEVICE_PATH_PROTOCOL *) NULL,
  { //DebugPort Protocol
    DebugPortReset,
    DebugPortWrite,
    DebugPortRead,
    DebugPortPoll
  },
   .  .  .
  (EFI_SERIAL_IO_PROTOCOL *) NULL,
   .  .  .
};
```

```
/**  Local function to obtain device path
  information from DebugPort variable.
  Records requested settings in DebugPort
  device structure.
  This gets called during the Supported
**/
VOID
GetDebugPortVariable ( VOID )
{
  UINTN      DataSize;
  EFI_DEVICE_PATH_PROTOCOL  DevicePath;
  EFI_STATUS                Status;
  DataSize = 0;
  Status = gRT->GetVariable (
   (CHAR16*) EFI_DEBUGPORT_VARIABLE_NAME,
    &gEfiDebugPortVariableGuid,
    NULL,
    &DataSize,
    mDebugPortDevice.DebugPortVariable);

 .  .  .
}
```

DebugPort.c Source - Init

- Entry point called by DXE Dispatcher or Shell Load Command
- Init simply installs the Driver Binding and Component Name Produced protocols

```
EFI_STATUS EFIAPI
InitializeDebugPortDriver (
  IN EFI_HANDLE                 ImageHandle,
  IN EFI_SYSTEM_TABLE           *SystemTable
  )
{
  EFI_STATUS    Status;
  //
  // Install driver model protocol(s).
  //
  Status =
   EfiLibInstallDriverBindingComponentName2 (
    ImageHandle,
          SystemTable,
          &gDebugPortDriverBinding,
          ImageHandle,
          &gDebugPortComponentName,
          &gDebugPortComponentName2
         );
  ASSERT_EFI_ERROR (Status);

  return EFI_SUCCESS;
}
```

DebugPort.c Source- Supported

- If there's a DEBUGPORT variable, the device path must match exactly.
- If there's no DEBUGPORT variable, then device path is not checked and does not matter.
- Checks to see that there's a serial I/O interface on the controller handle that can be bound BY_DRIVER | EXCLUSIVE.
- If all these tests succeed, then we return EFI_SUCCESS, otherwise, return EFI_UNSUPPORTED or other error returned by OpenProtocol.

```
EFI_STATUS EFIAPI
DebugPortSupported (
  IN EFI_DRIVER_BINDING_PROTOCOL
    *This,
  IN EFI_HANDLE ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL
                *RemainingDevicePath
  )
{
// Check to see that there's not
//debugport protocol already published,
  if (gBS->LocateProtocol
    (&gEfiDebugPortProtocolGuid, NULL,
    (VOID **) &DebugPortInterface) !=
    EFI_NOT_FOUND) {
     return EFI_UNSUPPORTED;
  }
// Read DebugPort variable
    GetDebugPortVariable ();

if (mDebugPortDevice.DebugPortVariable
    != NULL) {
// There's a DEBUGPORT variable
 if (Dp1 == NULL) {
      return EFI_OUT_OF_RESOURCES;
    }

    Dp2 = Dp1;
```

DebugPort.c

DebugPort.c Source- Supported

```
//check to see if the closest matching
//handle matches the controller handle
    Status = gBS->LocateDevicePath (
      &gEfiSerialIoProtocolGuid,
      &Dp2,
      &TempHandle
    );
    if (Status == EFI_SUCCESS
     && TempHandle != ControllerHandle) {
      Status = EFI_UNSUPPORTED;
    }
    if (Status == EFI_SUCCESS &&
        (Dp2->Type !=
        MESSAGING_DEVICE_PATH ||
        Dp2->SubType != MSG_VENDOR_DP ||
        *((UINT16 *) Dp2->Length) !=
        sizeof (DEBUGPORT_DEVICE_PATH))){
      Status = EFI_UNSUPPORTED;
    }
    if (Status == EFI_SUCCESS &&
        !CompareGuid
        (&gEfiDebugPortDevicePathGuid,
         (GUID *) (Dp2 + 1)))         {
      Status = EFI_UNSUPPORTED;
    }
```

Tests

```
FreePool (Dp1);
    if (EFI_ERROR (Status)) {
      return Status;
    }
  }
// no Status errors
Status = gBS->OpenProtocol (
          ControllerHandle,
          &gEfiSerialIoProtocolGuid,
          (VOID **) &SerialIo,
          This->DriverBindingHandle,
          ControllerHandle,
          EFI_OPEN_PROTOCOL_BY_DRIVER
          | EFI_OPEN_PROTOCOL_EXCLUSIVE
         );
// . . .
gBS->CloseProtocol (
        ControllerHandle,
        &gEfiSerialIoProtocolGuid,
        This->DriverBindingHandle,
        ControllerHandle
        );
  return EFI_SUCCESS;
}
```

UNSUPPORTED

SUPPORTED

DebugPort.c Source - Start

```
EFI_STATUS EFIAPI
DebugPortStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL
                *RemainingDevicePath
  )
{
  EFI_STATUS   Status;
//  . . .
  Status = gBS->OpenProtocol (
      ControllerHandle,
      &gEfiSerialIoProtocolGuid,
      (VOID **)
      &mDebugPortDevice.SerialIoBinding,
      This->DriverBindingHandle,
      ControllerHandle,
      EFI_OPEN_PROTOCOL_BY_DRIVER |
      EFI_OPEN_PROTOCOL_EXCLUSIVE
     );
  if (EFI_ERROR (Status)) {
    return Status;
  }
```

- Binds exclusively to serial I/O on the controller handle
- Initializes the Serial I/O interface
- Creates a Device Path on new handle
- Publish DebugPort protocol and Device Path Protocols

DebugPort.c Source- Start

```
mDebugPortDevice.SerialIoDeviceHandle =
      ControllerHandle;
// Initialize the Serial Io interface...
  Status =
    mDebugPortDevice.SerialIoBinding
     ->SetAttributes (
      mDebugPortDevice. . . // attributes
      );
  if (EFI_ERROR (Status)) {
    mDebugPortDevice. // attributes
    Status =
   mDebugPortDevice.SerialIoBinding
     ->SetAttributes (
      mDebugPortDevice.SerialIoBinding,
      mDebugPortDevice. // attributes
     );
   if (EFI_ERROR (Status)) {
      gBS->CloseProtocol (
         ControllerHandle,
         &gEfiSerialIoProtocolGuid,
         This->DriverBindingHandle,
         ControllerHandle
        );
      return Status;
    }
  }
```

```
 mDebugPortDevice.SerialIoBinding->Reset
   (mDebugPortDevice.SerialIoBinding);
// Create device path instance -DebugPort
  DebugPortDP.Header.Type =
                MESSAGING_DEVICE_PATH;
  DebugPortDP.Header.SubType =
                MSG_VENDOR_DP;
  SetDevicePathNodeLength
        (&(DebugPortDP.Header),
        sizeof (DebugPortDP));
  CopyGuid (&DebugPortDP.Guid,
        &gEfiDebugPortDevicePathGuid);
  Dp1 = DevicePathFromHandle
        (ControllerHandle);
  if (Dp1 == NULL) {
    Dp1 = &EndDP;
    SetDevicePathEndNode (Dp1);
  }

  mDebugPortDevice.DebugPortDevicePath =
    AppendDevicePathNode (Dp1,
    (EFI_DEVICE_PATH_PROTOCOL *)
   &DebugPortDP);
```

DebugPort.c Source- Start

```
 if (mDebugPortDevice.DebugPortDevicePath
   == NULL) {
    return EFI_OUT_OF_RESOURCES;
  }
// Publish DebugPort & Device Path
  Status =
   gBS->InstallMultipleProtocolInterfaces
    (
    &mDebugPortDevice.
        DebugPortDeviceHandle,
    &gEfiDevicePathProtocolGuid,
     mDebugPortDevice.
         DebugPortDevicePath,
     &gEfiDebugPortProtocolGuid,
     &mDebugPortDevice.
         DebugPortInterface,
      NULL
     );
  if (EFI_ERROR (Status)) {
    gBS->CloseProtocol (
          ControllerHandle,
          &gEfiSerialIoProtocolGuid,
          This->DriverBindingHandle,
          ControllerHandle
          );
    return Status;
  }
```

```
// Connect debugport child to serial io
  Status = gBS->OpenProtocol (
   ControllerHandle,
   &gEfiSerialIoProtocolGuid,
   (VOID **) &mDebugPortDevice.
             SerialIoBinding,
    This->DriverBindingHandle,
    mDebugPortDevice.
              DebugPortDeviceHandle,
     EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER
     );
  if (EFI_ERROR (Status)) {
    gBS->CloseProtocol (
       ControllerHandle,
       &gEfiSerialIoProtocolGuid,
       This->DriverBindingHandle,
       ControllerHandle
       );
    return Status;
  }

  return EFI_SUCCESS;
}
```

DebugPort.c Source - Stop

```
EFI_STATUS EFIAPI
DebugPortStop (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE ControllerHandle,
  IN UINTN      NumberOfChildren,
  IN EFI_HANDLE *ChildHandleBuffer
  )
{  EFI_STATUS  Status;
 if (NumberOfChildren == 0) {
// Close the bus driver
  gBS->CloseProtocol (
     ControllerHandle,
     &gEfiSerialIoProtocolGuid,
     This->DriverBindingHandle,
     ControllerHandle
     );
  mDebugPortDevice.SerialIoBinding = NULL;
  gBS->CloseProtocol (
     ControllerHandle,
     &gEfiDevicePathProtocolGuid,
     This->DriverBindingHandle,
     ControllerHandle
     );
  FreePool
    (mDebugPortDevice.DebugPortDevicePath);
  return EFI_SUCCESS;
```

- Close the bus driver when Number of Children is 0 then return
- Stop this driver on ControllerHandle by removing Serial I/O protocol on the ControllerHandle
- Uninstall our protocols DevicePath and DebugPort

DebugPort.c Source - Stop

```
} else {
// Disconnect SerialIo child handle
 Status = gBS->CloseProtocol (
     mDebugPortDevice.SerialIoDeviceHandle,
     &gEfiSerialIoProtocolGuid,
     This->DriverBindingHandle,
     mDebugPortDevice.DebugPortDeviceHandle
    );

  if (EFI_ERROR (Status)) {
    return Status;
  }
// Unpublish our protocols DevicePath,
//         DebugPort
  Status = gBS
   ->UninstallMultipleProtocolInterfaces (
     mDebugPortDevice.DebugPortDeviceHandle,
     &gEfiDevicePathProtocolGuid,
     mDebugPortDevice.DebugPortDevicePath,
     &gEfiDebugPortProtocolGuid,
     &mDebugPortDevice.DebugPortInterface,
     NULL
    );
```

```
 if (EFI_ERROR (Status)) {
   gBS->OpenProtocol (
     ControllerHandle,
     &gEfiSerialIoProtocolGuid,
     (VOID **)
        &mDebugPortDevice.SerialIoBinding,
     This->DriverBindingHandle,
     mDebugPortDevice.
         DebugPortDeviceHandle,
     EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER
    );
  } else {
    mDebugPortDevice.DebugPortDeviceHandle
        = NULL;
  }
 }
 return Status;
}
```


DebugPort.c Source- DebugPort Protocol Functions

- Reset() ⟺ Calls Seriallo:GetControl to flush buffer
- Read() ⟺ Calls Seriallo:Read() after setting if it's different than the last Seriallo access
- Write() ⟺ Calls Seriallo:Write() Writes 8 bytes at a time and does a GetControl between 8 byte writes to help insure reads are interspersed
- Poll() ⟺ Calls Seriallo:Write() after setting if it's different than the last Seriallo access

ComponentName.c

ComponentName.c Source

```
//
// EFI Component Name Protocol
//
EFI_COMPONENT_NAME_PROTOCOL
  gDebugPortComponentName = {
    DebugPortComponentNameGetDriverName,
    DebugPortComponentNameGetControllerName,
    "eng"
};

//
// EFI Component Name 2 Protocol
//
EFI_COMPONENT_NAME2_PROTOCOL
  gDebugPortComponentName2 = {
   (EFI_COMPONENT_NAME2_GET_DRIVER_NAME)
    DebugPortComponentNameGetDriverName,
   (EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME)
    DebugPortComponentNameGetControllerName,
     "en"
};
```

```
EFI_UNICODE_STRING_TABLE
  mDebugPortDriverNameTable[] = {
   {
     "eng;en",
     (CHAR16 *) L"DebugPort Driver"
   },
   {
    NULL,
    NULL
   }
};
EFI_STATUS EFIAPI
DebugPortComponentNameGetDriverName
(  // . . .
 )
{
  return LookupUnicodeString2 (
      Language,
      This->SupportedLanguages,
      mDebugPortDriverNameTable,
      DriverName,
      (BOOLEAN)(This ==
          &gDebugPortComponentName)
    );
}
```

DebugPortDxe.inf

DebugPortDxe.inf Source

[Defines]
[Sources]
  ComponentName.c
  DebugPort.c
  DebugPort.h

[Packages]
  MdePkg/MdePkg.dec

[LibraryClasses]
  DevicePathLib
  UefiRuntimeServicesTableLib
  UefiBootServicesTableLib
  MemoryAllocationLib
  BaseMemoryLib
  UefiLib
  UefiDriverEntryPoint
  DebugLib

[Guids]
  gEfiDebugPortVariableGuid
  gEfiDebugPortDevicePathGuid

[Protocols]
  gEfiSerialIoProtocolGuid
  gEfiDevicePathProtocolGuid
  gEfiDebugPortProtocolGui

Example of EDK II Inf file

DebugPort Demo Nt32

- Load DebugPortDxe.efi
- Check loaded driver

```
Shell> fsnt0:

fsnt0:\> load DebugPortDxe.efi
load: Image fsnt0:\DebugPortDxe.efi loaded at 5E78000 - Success

fsnt0:\> _
```

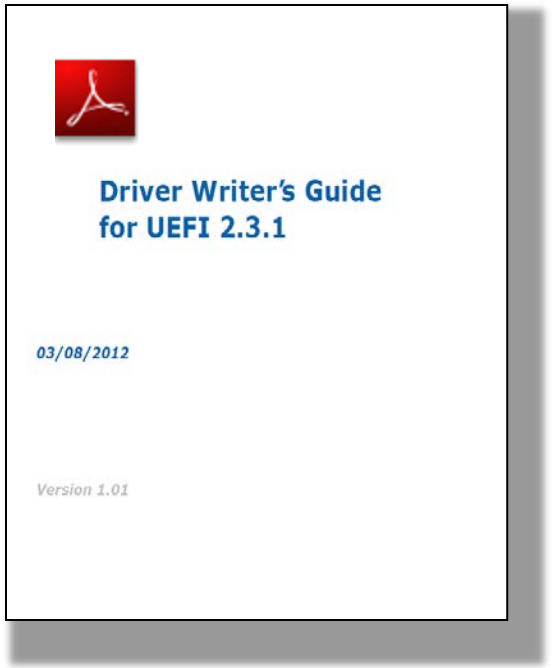
```
fsnt0:\> dh -d 7a
 7A: Image(\/DebugPortDxe.efi) ImageDevPath (..00000000)/\/DebugPortDxe.efi)Driv
erBinding ComponentName ComponentName2
     Driver Name     : DebugPort Driver
     Image Name      : \/DebugPortDxe.efi
     Driver Version  : 00000005
     Driver Type     : BUS
     Configuration   : NO
     Diagnostics     : NO
     Managing        :
       Ctrl[6D] : COM1
         Child[7B] : VenHw(58C518B1-76F3-11D4-BCEA-0080C73C8881)/VenHw(0C95A93D-
A006-11D4-BCFA-0080C73C8881,00000000)/Uart(115200,8,N,1)/DebugPort()
```

Agenda


Driver Introduction
UEFI Protocols
Driver Design
Driver Example - DebugPort
Driver Writer's Guide

UEFI Driver Writer's Guide

- Captures Practical Experiences
- Use as a Recipe Book
- Must Read for all UEFI Driver Developers
- Living Document
  - Content Based on Industry Feedback
  - Updated as Techniques are Refined
  - Updated as New Technologies are Introduced
- Updated to UEFI 2.3.1 Mar 2012
  - Driver Development web page



General Topics

- Overview of UEFI Concepts
- UEFI Services
  - Commonly Used by UEFI Drivers
  - Rarely Used by UEFI Drivers
  - Should Not Be Used by UEFI Drivers
- General Driver Design Guidelines
- Classes of UEFI Drivers
- Driver Entry Point
- Private Context Data Structures
- UEFI Driver Model Protocols

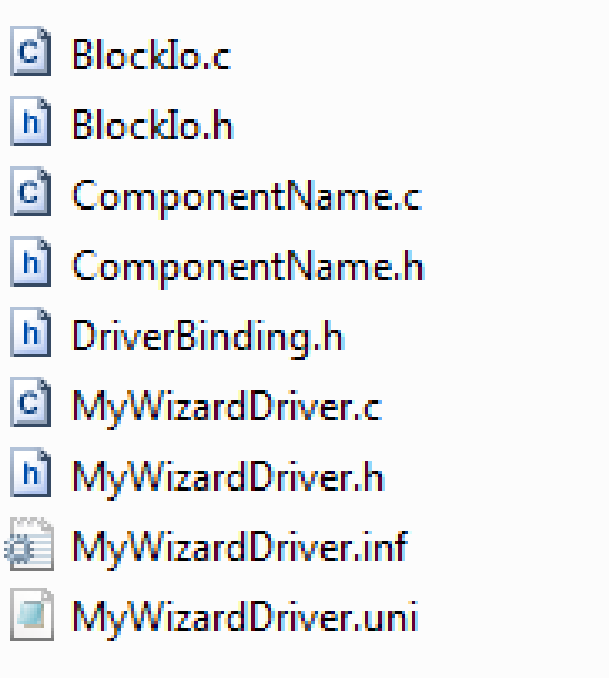
Additional Driver Development Guides

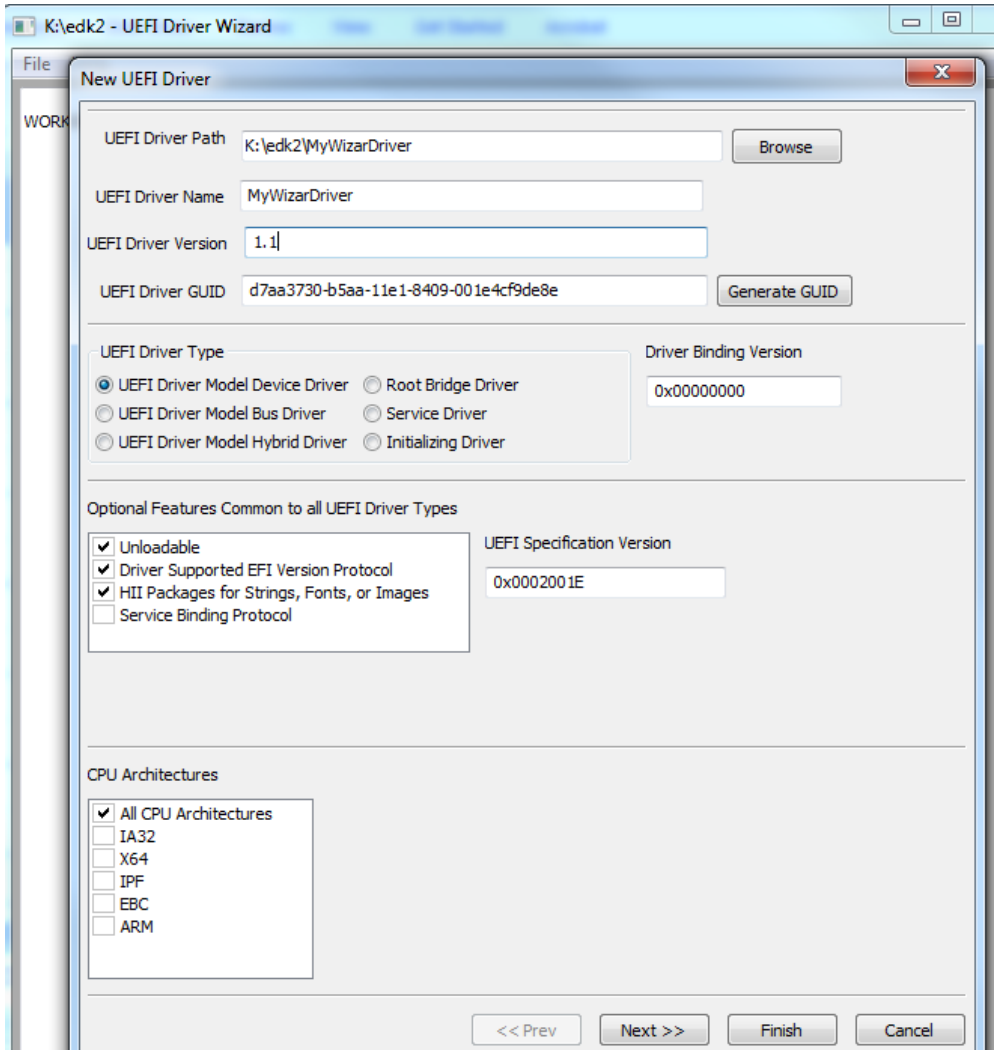
- New driver development guides are being created for specific device types
- Short documents (8-12 pages) point developers to proper resources
- Visit the intel.com "UEFI Driver and Application Tool Resources" page to find resources for UEFI driver developers.

http://intel.com/go/uefi-ihv

Driver Wizard

- MyWizardDriver
  - Files created





- http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=UEFI_Driver_Wizard

Summary

- Good Designs Save Time and Money
- Many Tools Available to Test and Debug
- Using Driver Guidelines Improves Portability
- Compile in EBC to have one driver image to support x86, x64 and Itanium.

Further Information

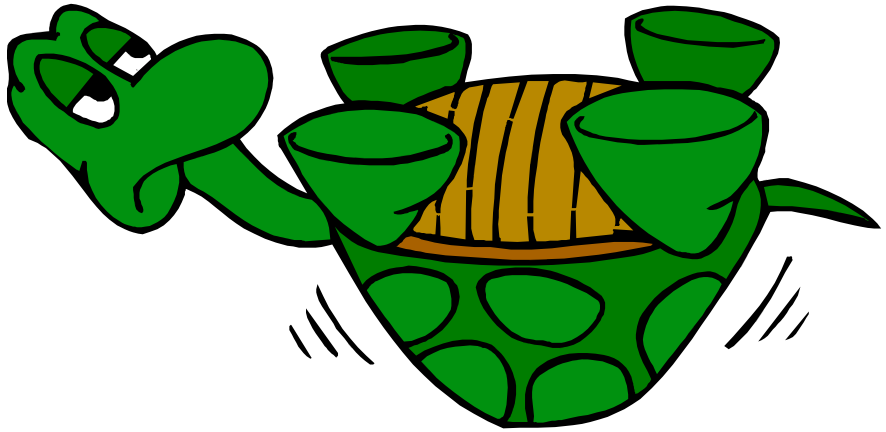
- http://www.intel.com/udk
  - Intel site for UEFI information & IDF presentations
- http://www.uefi.org
  - UEFI Forum, Specifications and Self Certification Test
- https://www.TianoCore.org
  - Website for UEFI open source resources
  - UEFI Developer Kit (UDK) and EDK II
  - Driver Wizard
  - Driver Development web page

Q & A


intel®
Software

Back Up

- Required Materials for IHVs
- Optional Materials for OEMs

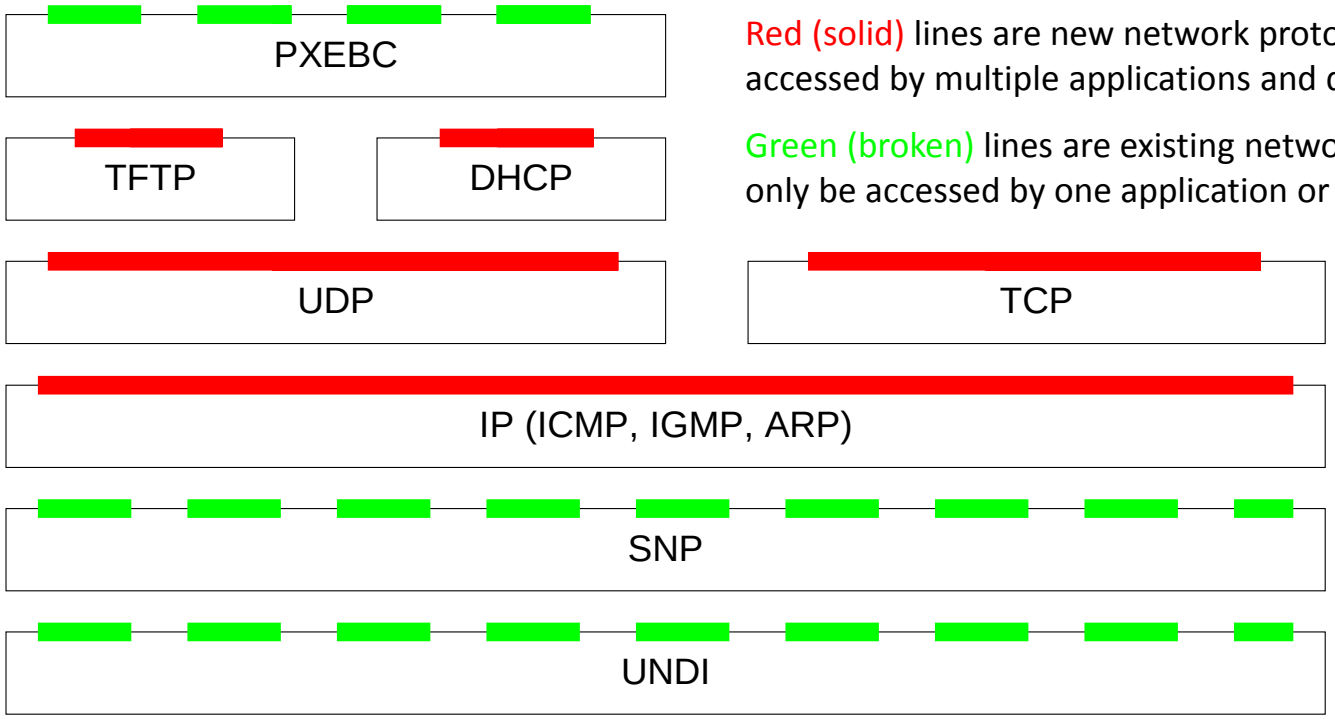


See Backup Presentation "*UEFI-Drivers-Backup Only*"

Network UEFI 2.1

Back Up Materials

Protocol Diagram



Red (solid) lines are new network protocol APIs that can be accessed by multiple applications and drivers at the same time.

Green (broken) lines are existing network protocol APIs that can only be accessed by one application or driver at a time.

Refer to the "UEFI Driver Development Guide for Network Boot Devices" @ intel.com for more information

Internet Protocol (IP)

- One IP instance per SNP instance.
- Can be used by multiple applications and drivers at the same time.
- Implements a subset of IP (RFC 791).
  - No support for IP options in alpha code.
  - Minimal support for ICMP
    - ping works
    - errors can be routed up to applications and drivers
  - Just enough to make UDP and TCP layers work

User Datagram Protocol (UDP)

- One UDP instance per IP instance.
- Can be used by multiple applications and drivers at the same time.
- Implements all of UDP (RFC 768).

Transmission Control Protocol (TCP)

- One TCP instance per IP instance.
- Can be used by multiple applications and drivers at the same time.
- Implements a subset of TCP (RFC 793).
  - No application control of window or segment sizes.
  - Minimal data returned for socket status.
  - Just enough to make implementing a Berkeley Socket interface possible.

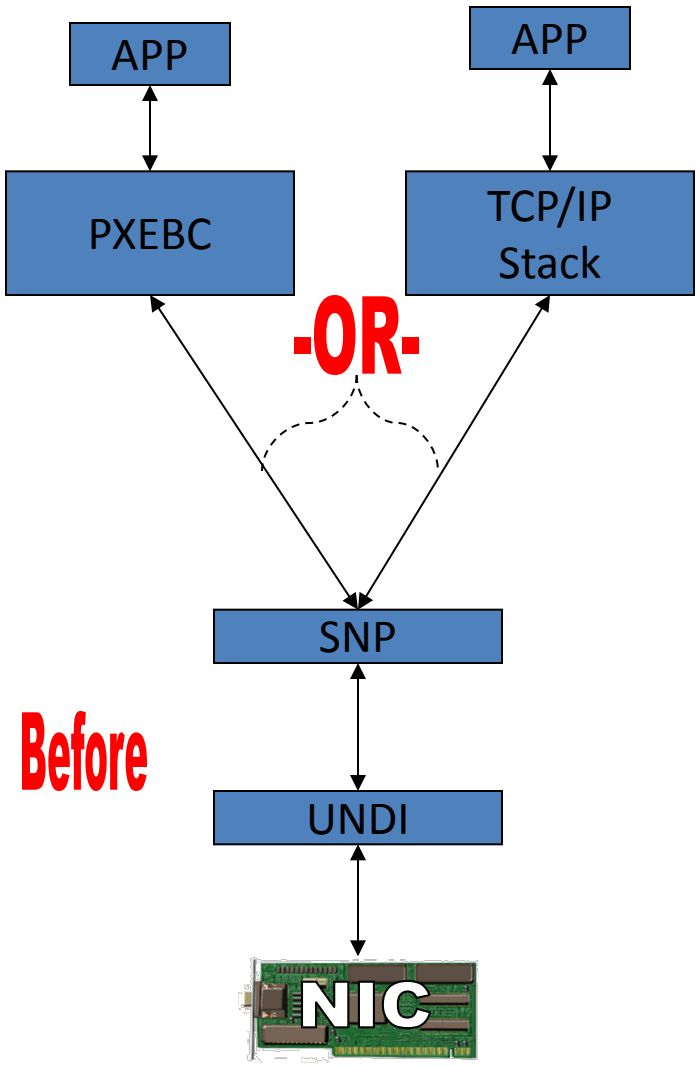
Multicast Trivial File Transfer Protocol (Mtftp)

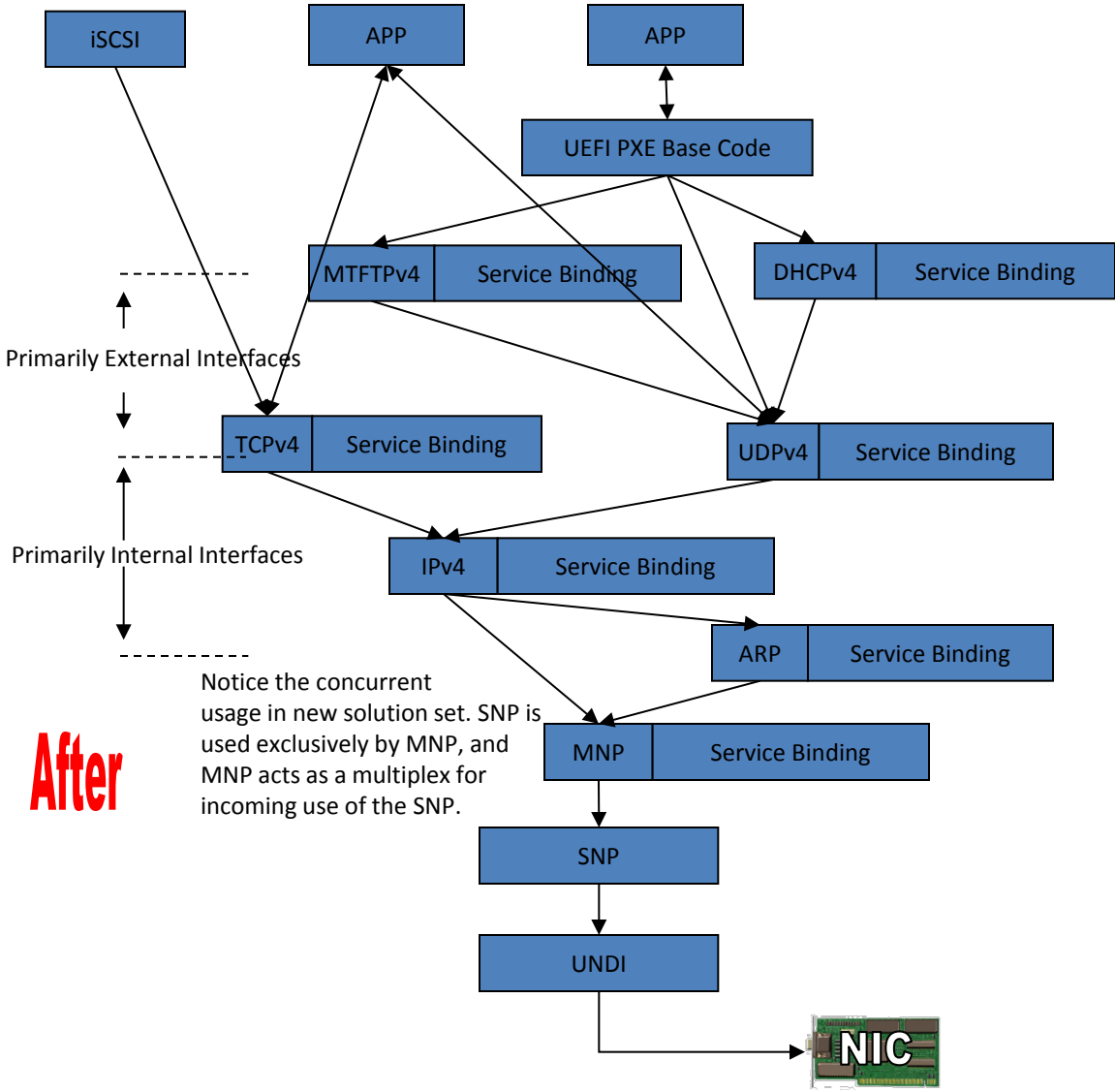
- One MTFTP instance per UDP/IP instance.
- Can be used by multiple applications and drivers at the same time.
- Implements all of TFTP (RFC 1350) + Options (RFCs 2347, 2348, 2349) + Internet-Drafts covering BIG extensions.

Dynamic Host Configuration Protocol (DHCP)

- One DHCP instance per UDP/IP instance.
- Can be used by multiple applications and drivers at the same time.
- Only provides network configuration information.
  - Does not manage network connections.
  - If DHCP data changes, application must update network connection accordingly.
- Implements all of DHCP (RFC 2131 and 2132)

Current Network Stack Layout






intel®
Software

Disclaimer

THIS INFORMATION COTNAINED IN THIS DOCUMENT, INCLUDING ANY TEST RESULTS ARE PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT OR BY THE SALE OF INTEL PRODUCTS. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel retains the right to make changes to its specifications at any time, without notice.

Recipients of this information remain solely responsible for the design, sale and functionality of their products, including any liability arising from product infringement or product warranty.

Intel may make changes to specifications, product roadmaps and product descriptions at any time, without notice.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

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