UEFI Driver Development Training Protocols

Leon Li
UEFI Development
Intel

Agenda

- What is a protocol?
- Protocol vs. C++ class
- The “core” protocols
- Producing protocols
- Consuming protocols
- Custom protocols

What is a protocol?

- An interface
- Must be Produced by a driver
- May be Consumed by anyone
- A set of related functions and their associated data

- Examples:
  - Device Path, PCI I/O, Disk I/O, GOP, UNDI

PCI IO Protocol - Example

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 2.1 Spec.

Disk IO Protocol - Example

GUID

```
#define EFI_DISK_IO_PROTOCOL_GUID \
{0xCE345171,0xBA0B,0x11d2,0x8e,0x4F,0x00,0xa0,0xc9
  ,0x69,0x72,
0x3b}
```

Revision Number

```
#define EFI_DISK_IO_PROTOCOL_REVISION 0x00010000
```

Protocol Interface Structure

```
typedef struct _EFI_DISK_IO_PROTOCOL {
UINT64 Revision;
EFI_DISK_READ ReadDisk;
EFI_DISK_WRITE WriteDisk;
} EFI_DISK_IO_PROTOCOL;
```

See § 12.6 UEFI 2.1 Spec.

Device Path Protocol - Example

GUID

```
#define EFI_DEVICE_PATH_PROTOCOL_GUID \
{0x09576e91,0x6d3f,0x11d2,0x8e,0x39,0x00,0xa0,0xc9,0x69,0x72,
0x3b}
```

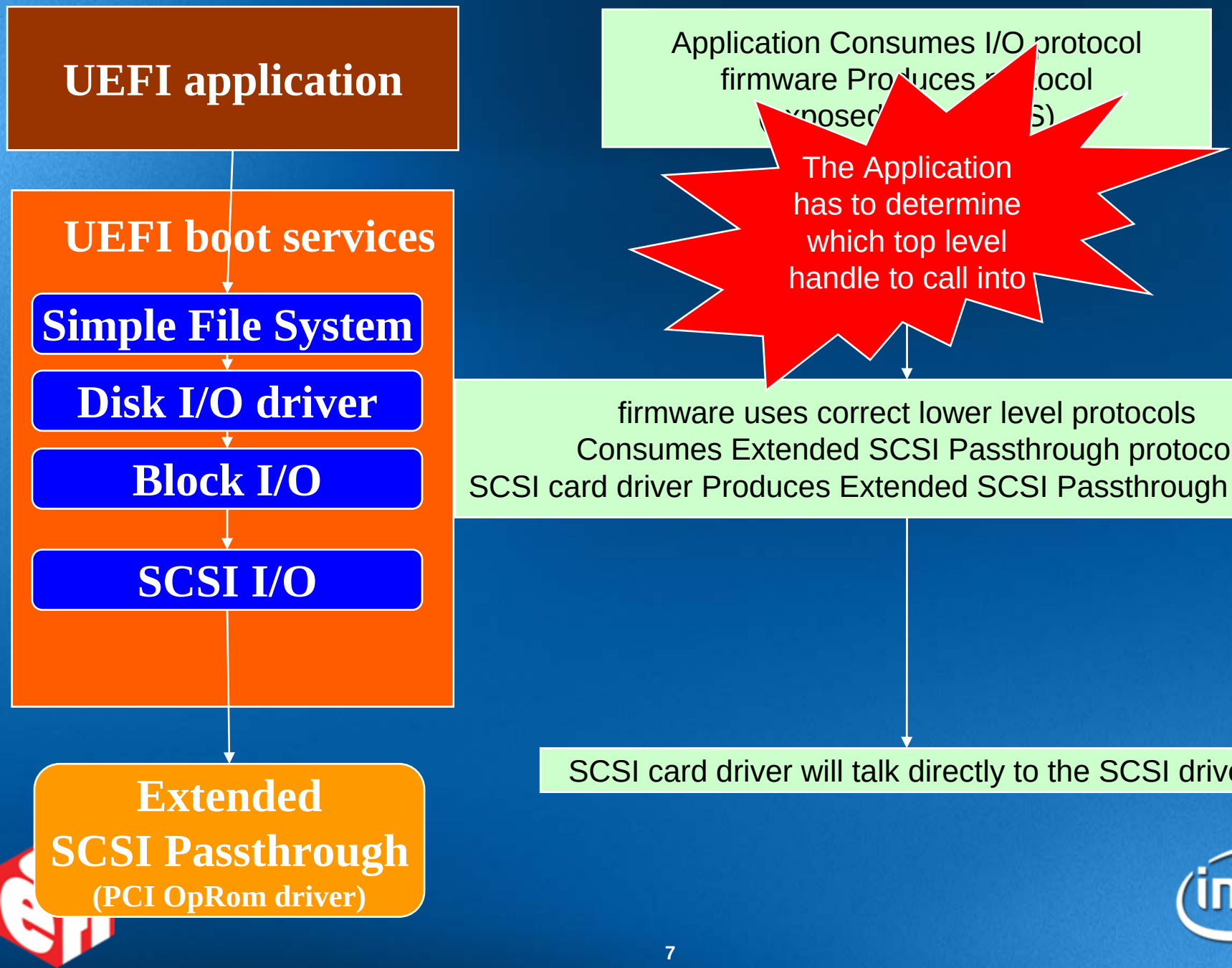
Protocol Interface Structure

```
typedef struct _EFI_DEVICE_PATH_PROTOCOL {
UINT8 Type;
UINT8 SubType;
UINT8 Length[2];
} EFI_DEVICE_PATH_PROTOCOL;
```

- The device path describes the location of the device the handle is for
- A UEFI driver may access only a physical device for which it provides functionality.

See § 9.2 UEFI 2.1 Spec.

Protocol example

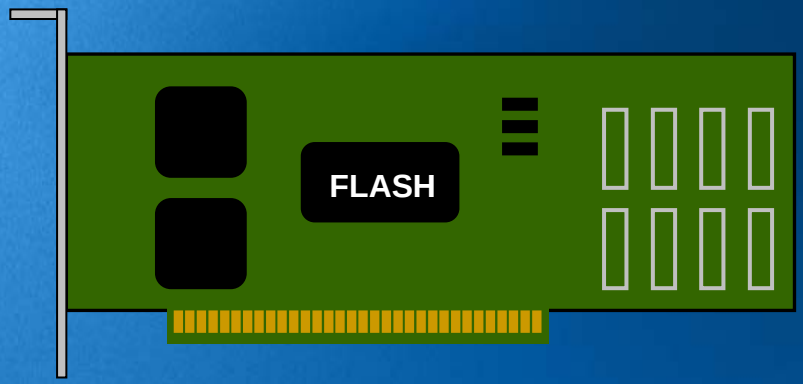


Protocol vs. C++ class

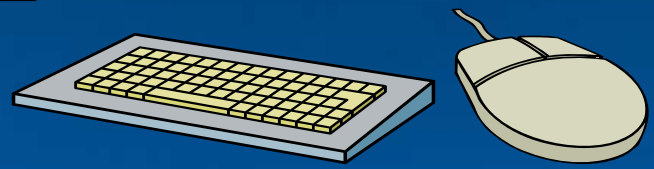
- An UEFI protocol is logically similar to a C++ class
  - With lower memory overhead
    - No virtual function table
  - Has private member variables
  - Has exposed functions
  - Has private functions
  - Has a ‘This’ pointer

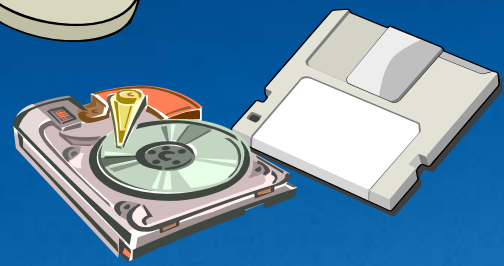
  - Look/feel very different

What I/O Protocols are Consumed?

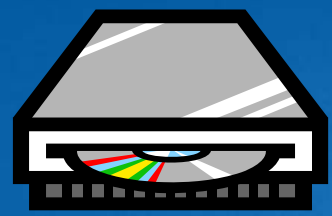


- PCI Adapters
  - –PCI I/O Protocol
  - –Device Path Protocol

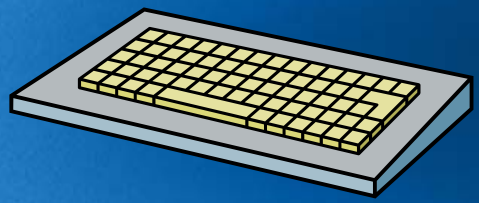




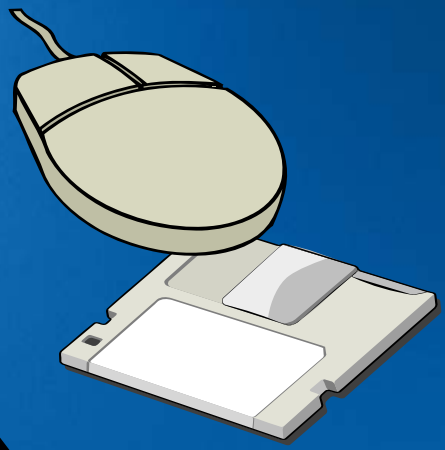
- USB Peripherals
  - –USB I/O Protocol
  - –Device Path Protocol



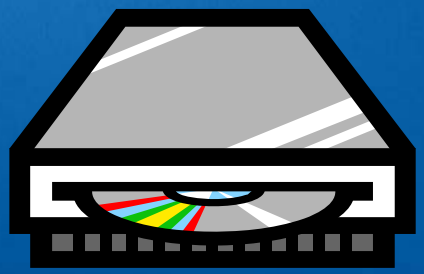
What I/O Protocols are Produced?



- Simple Input Protocol

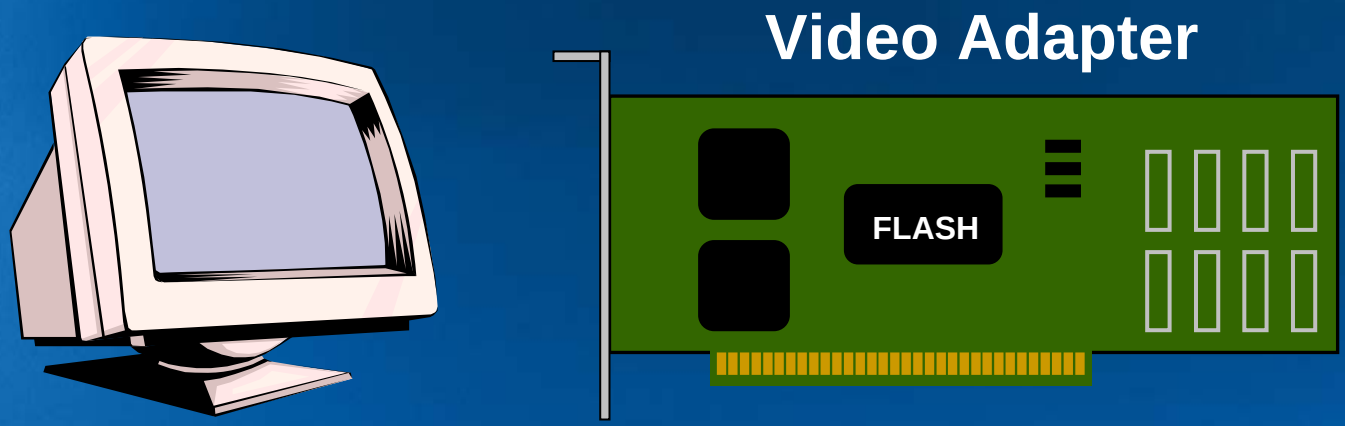


- Simple Pointer Protocol



- Block I/O Protocol

What I/O Protocols are Produced?

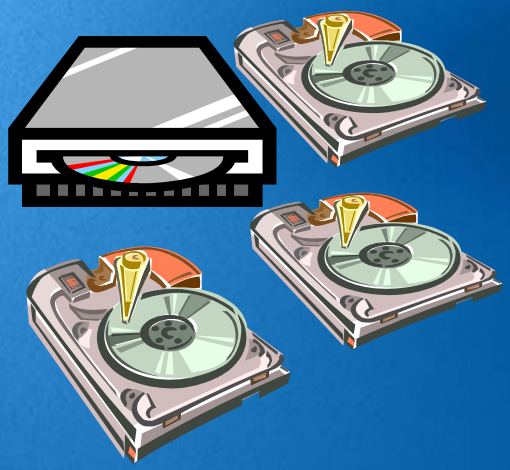


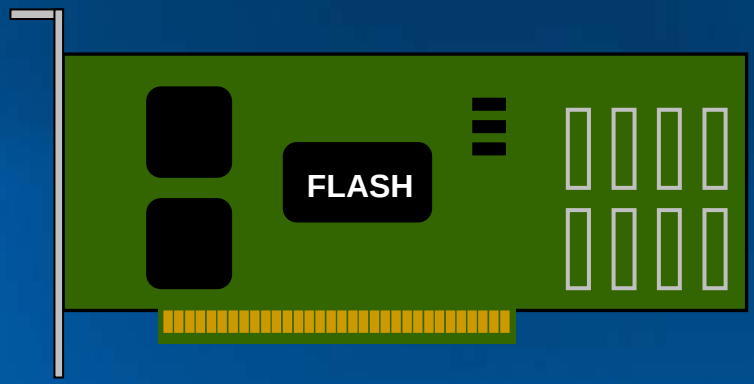
- UGA Draw Protocol and UGA I/O Protocol

or

- Simple Text Output Protocol

What I/O Protocols are Produced?





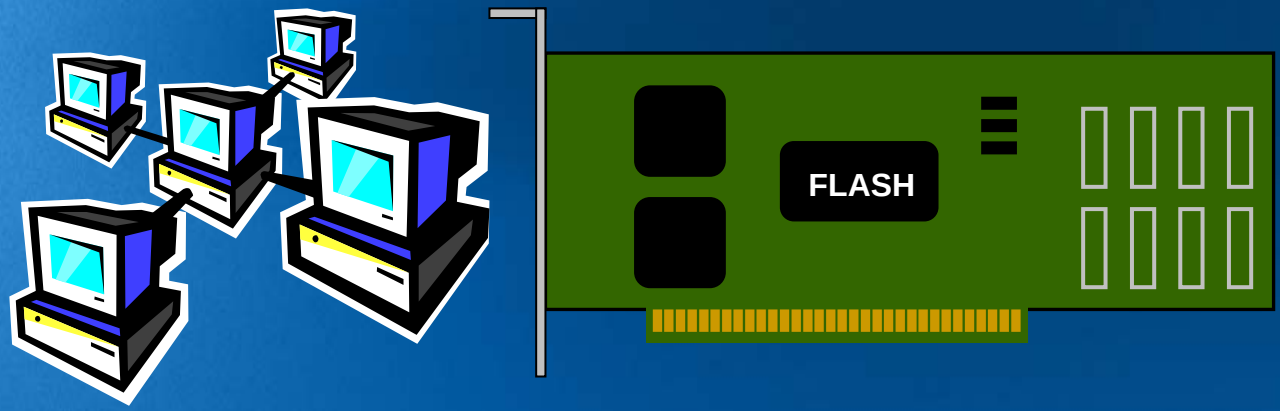
SCSI
SCSI RAID
Fiber Channel

- SCSI Pass Thru Protocol

and

- Block I/O Protocol

What I/O Protocols are Produced?

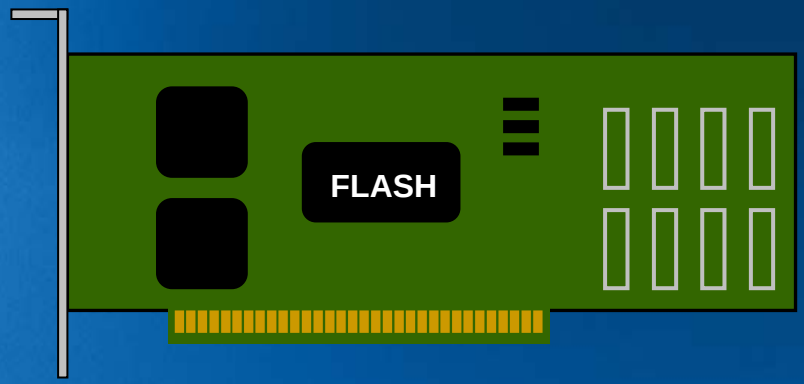


- UNDI

and

- Network Interface Identifier Protocol

What I/O Protocols are Produced?

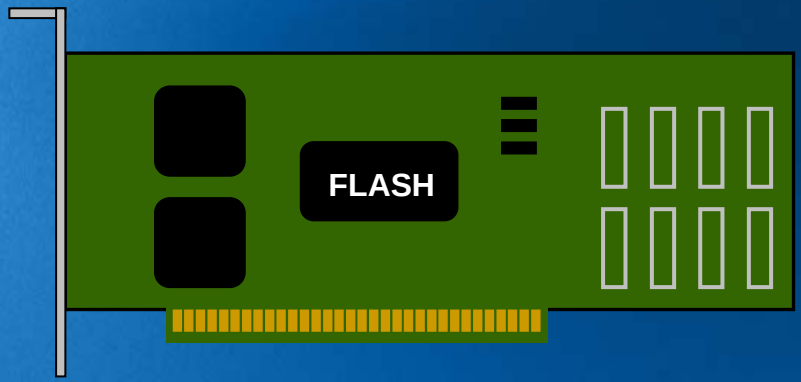


Serial Adapter
- **Single UART**
- **Multi-Port UART**
- **Modem**

Byte Stream Device

- Serial I/O Protocol
- Debug Port Protocol

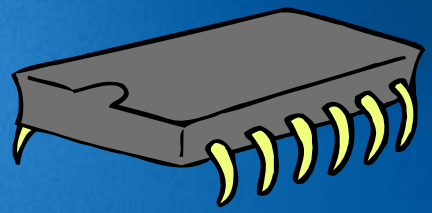
What I/O Protocols are Produced?



USB Host Controller
- UHCI
- OHCI
- EHCI

- USB Host Controller Protocol

What I/O Protocols are Produced?



Other Boot Devices

- FLASH
- ROM
- Tape Drives

- Load File Protocol

The “core” protocols

- Device Path protocol
  - UEFI Chapter 9
- Driver Binding protocol
  - UEFI Chapter 10.1
- Other UEFI Driver Model protocols
  - UEFI Chapter 10.2 onwards
- LoadedImage protocol

Device path protocol

- Description of location of the device
- Required for boot devices, logical devices and images
- 6 types:
  - Hardware
  - ACPI – UID/HID of device in AML
  - Messaging – LAN, Fiber Channel, ATAPI, SCSI, USB, and Vendor defined (like terminal type in console)
  - Media – ie HDD, FDD or CD-ROM
  - EDD 3.0 boot device – see EDD 3.0 spec int13 48
  - End of hardware – marks end of device path

Device paths

- Definition of where devices are physically Located
  - Utilize full UEFI device path to boot device
    - E.g. ACPI - PCI - SCSI - partition(SIG) – Filename
    - E.g. ACPI - PCI - MAC(MAC address)
  - Hard disk
    - short path option which only consists of partition ID and filename or:
    - GPT version is safer since it is unique.
  - Removable media (CDROM and LS120)
    - \EFI\BOOT\BOOTIA64.EFI or BOOTIA32.EFI
  - Required for locating console, boot device, etc…

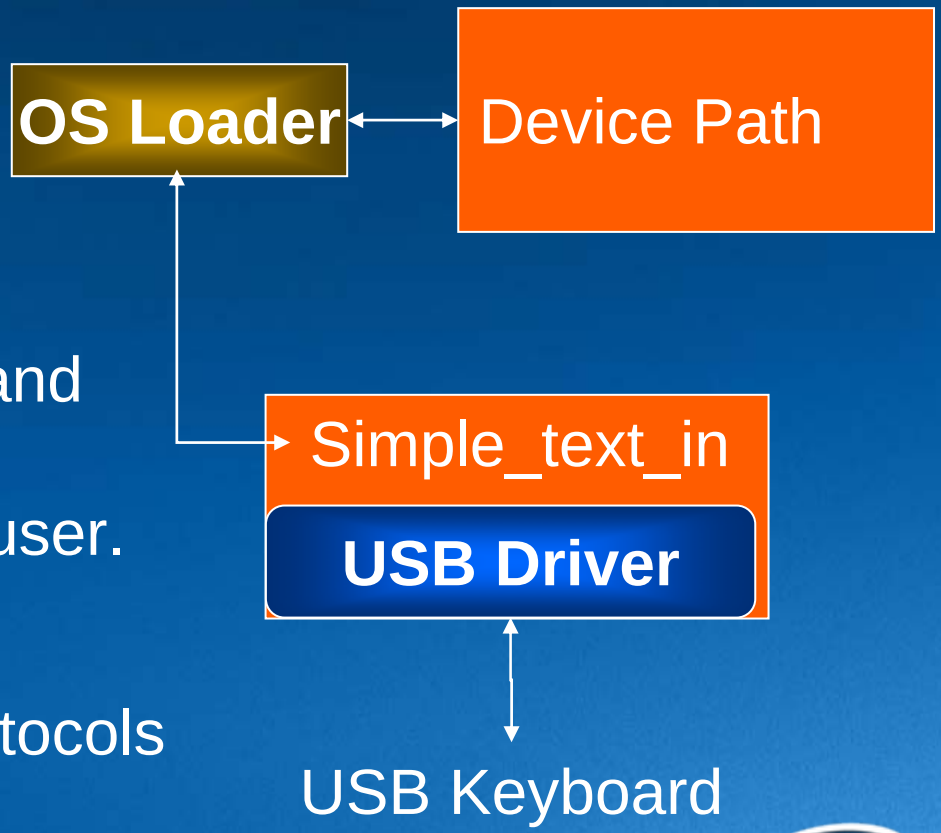
Device path examples

- Acpi(PNP0A03,0)/Pci(3|1)/Ata(Primary,Master)
  - Device path pointing to primary master Atapi device
- Acpi(PNP0A03,1)/Pci(0|0)/Scsi(Pun6,Lun0)/HD(Part1,Sig19772100-1DD2-100)
  - Device path pointing to part1 or a SCSI hard drive
- Acpi(PNP0A03,1)/Pci(0|0)/Scsi(Pun6,Lun0)/HD(Part2,Sig2562E300-1DD2-100)
- Acpi(PNP0A03,0)/Pci(3|1)/Ata(Secondary,Master)

Example

OS Loader wants to display choices for boot devices that it gets from Device Path protocol.

Use Device Path and find all devices that Produce it



Use Simple_Text and Simple_Output to interact with the user.

Loader Consumes protocols
Framework Produces the protocols

Driver Binding protocol

- *3 functions*
  - *Supported()*
  - *Start()*
  - *Stop()*
- *3 data blocks*
  - *Version*
  - *ImageHandle*
  - *DriverBindingHandle*

DriverBinding.Supported()

- ***Parameters:***
  - ***Current Controller***
  - ***Child (optional)***
- **Checks to see if a driver supports a controller**
- **Must 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**

DriverBinding.Start()

- ***Parameters:***
  - ***Current Controller***
  - ***Child (optional)***
- ***Starts the device specified by Controller***
- ***Child is optionally specified***
  - If NULL bus driver will start all children
  - If exists than starts that child of the child device

DriverBinding.Stop()

- ***Parameters:***
  - ***Current Controller***
  - ***Child Count***
  - ***ChildHandleBuffer (optional)***
- If Child Count is Zero or this is not a bus controller, stop the controller
- Otherwise stops Child Count number of children whose handles are in the buffer.
- Should be in reverse order from Start()

DriverBinding information

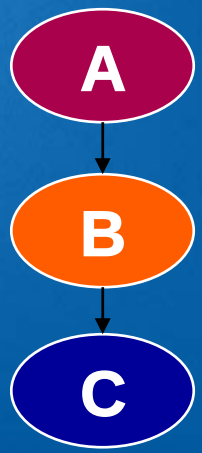
- Drivers cannot search for their hardware
- They only react to firmware questions for finding hardware
- Firmware makes all decisions about what driver controls what hardware.

Connect / Disconnect Controller interaction with DriverBinding

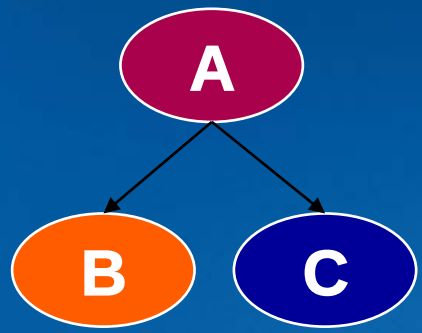
- When UEFI ConnectController() is called it will perform 2 actions.
  1. Create an ordered list of driver handles
     - DriverBinding->Supported()
  2. Connect drivers to controllers
     - DriverBinding->Start()
- When UEFI DisconnectController() is called it will perform 1 action.
  1. Disconnect drivers to controllers
     - DriverBinding->Stop()
- This can also be made to happen manually with UEFI shell using connect, disconnect, reconnect

EFI Services Binding
-UEFI *Driver Model*

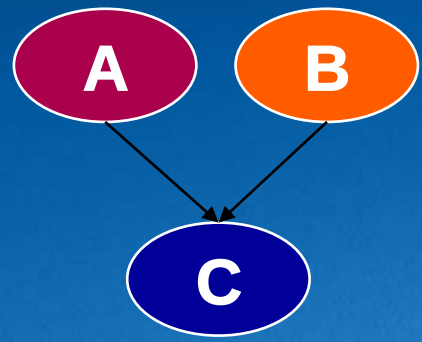
- Maps well - Hardware devices, Hardware bus controllers, Simple combinations of software services that layer on top of hardware devices
- NOT well - complex combinations of software services.



#1 Liner Stack



#2 Mult. Dependencies



#3 Mult. Consumers

See § 2.5.8 UEFI 2.1 Spec.

UEFI Driver Model Protocols

- (Driver Binding protocol)
- Platform Driver Override protocol
- Bus Specific Driver Override protocol
- Driver Configuration protocol
- Driver Diagnostics protocol
- Component Name protocol
- Service Binding protocol

How are drivers selected?

- Precedence Rules for Driver Selection
  - Context Override
    - Person (normally using shell) forces a specific driver
  - Platform Driver Override Protocol
    - Platform firmware specifies one driver over another
  - Bus Specific Driver Override Protocol
    - The bus controller will help determine which driver to use
  - Version
    - Uses Driver Binding Protocol Version field
    - Always used

Platform Driver Override protocol

- Maps Controllers to Drivers
- OEM Value Add - Platform Management
- Used by UEFI Boot Manager to Load Drivers
- Provides Ordered List to ConnectController()
- Higher than Bus Specific Driver Overrides
  - If 2 add in cards appear identical but have different driver versions this could be used to specify which driver to use for each card.

Bus Specific Driver Override protocol

- Optional - Not supported by all bus types
  - Supported by PCI, ISA, etc.
  - Not supported by USB, SCSI, etc.
- Maps Controllers to Drivers
- Attached to Controller Handle by Bus Driver
- Provides an Ordered List to ConnectConroller()
- Overrides Driver Binding Protocol Search
  - The PCI bus driver will Produce this protocol for each peripheral card with a driver.

Driver Configuration protocol

- 3 functions
  - SetOptions – allows driver to control the screen to display options and get choices from the user
  - OptionsValid – validate whether the current options are valid
  - ForceDefaults – sets the default options
- 1 data structure
  - SupportedLanguages

- Note: Deprecated in UEFI 2.1 and replaced with Platform to Driver Configuration Protocol.

Driver Diagnostics protocol

- 1 function
  - RunDiagnostics – when this function gets called the driver will run the diagnostics on the child (represented by a device path) passed in.
- 1 data structure
  - SupportedLanguages

ComponentName protocol
ComponentName2 protocol

- 2 functions
  - GetDriverName() – returns the name of the driver producing the protocol
  - GetControllerName() – returns the name of the device being managed
- 1 data structure
  - SupportedLanguages

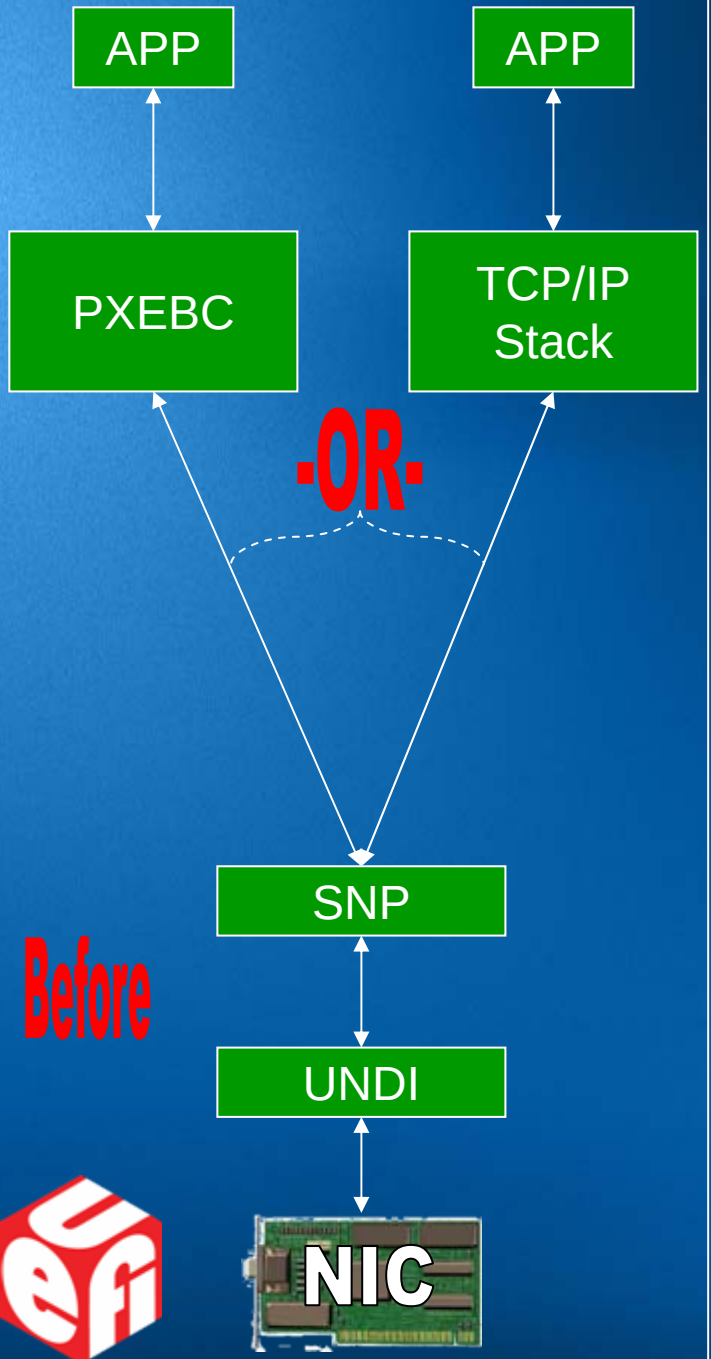
- Note errata for change of SupportedLanguages format (and GUID)
  - “eng” vs. “en-US”

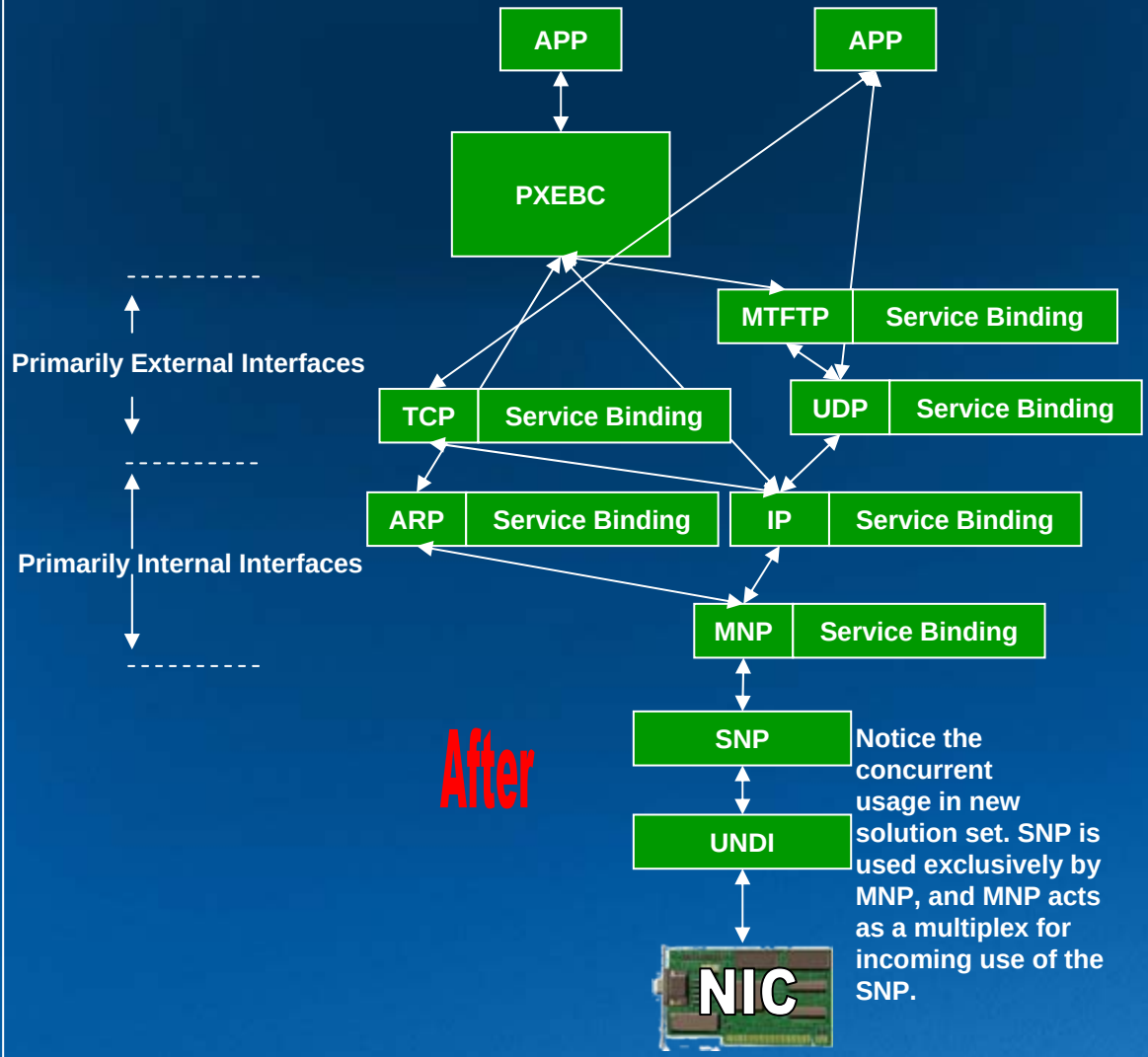
ComponentName protocol

- Limit Lengths of Names to 40 Unicode Characters
- Include Driver Name and Version Number
- UNDI Driver (Network Interface Controller)
  - Typically the Name of the PCI Controller
- MAC Node Produced by an UNDI Driver
  - Identify Location of Physical Connector on NIC
- PCI Slots
  - Identify Physical Location of PCI Slots in the System
- SCSI / SCSI RAID / Fiber Channel
  - Controller - Typically name of the PCI Controller
  - Channel - Identify Physical Location of the SCSI Channel
  - Disk - Use Results from INQUIRY Command

Service Binding protocol

- This protocol is used to create and destroy children that have protocols installed on their handles.
  - CreateChild()
  - DestroyChild()
  - The driver is responsible for adding and removing the protocols.
- This is used when a single driver wants to exclusively Consume a specific protocol. So a given driver Produces one of these to give a fake exclusive access to a protocol.





Notice the concurrent usage in new solution set. SNP is used exclusively by MNP, and MNP acts as a multiplex for incoming use of the SNP.

LoadedImage protocol

- Protocol that handles loading and unloading of images
- Device drivers generally do not use LoadImage(), StartImage(), or Exit()
- They do use UnloadImage()
  - Allows for control when driver unloads
  - Verifies that driver is disconnected (like Stop())
  - Releases all events and memory allocated

See §7.1.2 of Driver Writers Guide.

Producing Protocols

- Drivers Produce protocols
- In your driver header:
  - Include the header from the protocol directory
    - #include EFI_PROTOCOL_DEFINITION (DriverBinding)
  - Add the protocol your driver structure
  - Add a handle to your driver structure
  - Add function prototypes to the file
- In your driver implementation file
  - Set all the pointers to your function bodies
  - Install the protocols on the handles

Consuming protocols

- Drivers and applications Consume protocols
- In your header:
  - Include the header from the protocol directory
    - #include EFI_PROTOCOL_CONSUMER (DriverBinding)
  - Add the protocol pointer your driver structure
- In your driver implementation file
  - Open the protocol
  - Call the protocol functions

Custom Protocols

1. Producing custom protocols
   - See § “Using the EDK” for information on creating custom protocols
2. Publishing custom protocols
   - Send all files to consumers

Tips and Tricks for protocols

- DevicePath protocol has a lot of utility functions to help with DevicePath manipulations
- You can restrict consumers to a single one
  - Certain ‘bit-banging’ operations require this
  - Ex. UNDI or SNP (see the service binding flowchart)
- Produce a custom debug protocol
  - Your custom protocol can be protected
    - Don’t publish its interface
    - Only Produce it under specific conditions
- Write an UEFI application to test your driver

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	UGA Draw UGA I/O	SCSI Pass Thru Block I/O	UNDI, NII
Driver Binding	✓	✓	✓
Component Name	✓	✓	✓
Driver Configuration		✓	
Driver Diagnostics	✓	✓	✓
Unloadable	✓	✓	✓
Exit Boot Services Event			✓
Runtime			✓
Set Virtual Address Map Event			✓


intel®
Leap ahead™

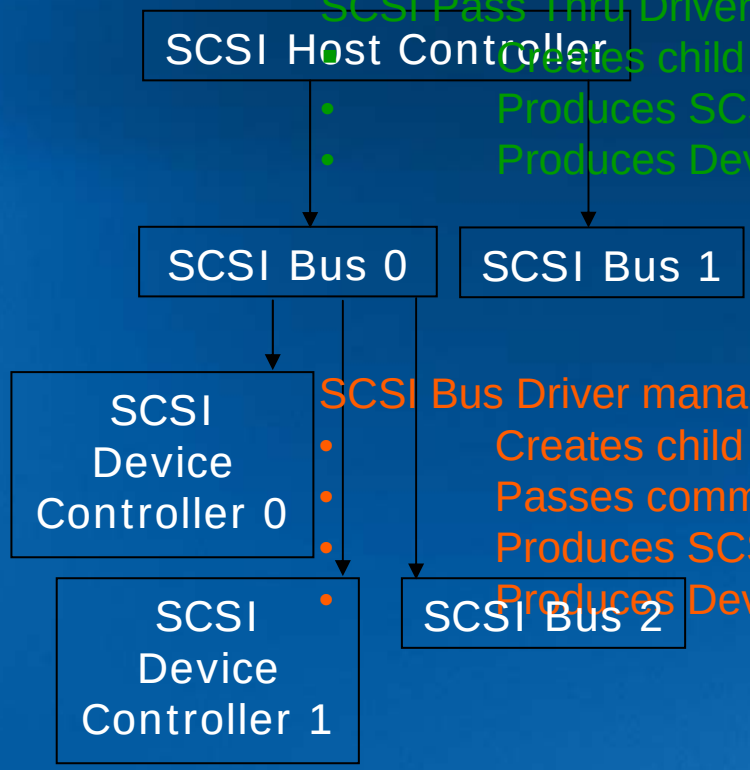
SCSI Driver Stack

- Usable by any device that utilizes SCSI commands
  - SCSI devices
  - ATAPI devices
  - FibreChannel devices
- Abstracts the implementation from UEFI
  - SCSI version
  - Protocol
- Compatibility

SCSI Driver Stack

- Why Hybrid Driver
  - Produce protocols on child handles AND onto own handle
    - On child channel handles
      - Produce Ext SCSI Pass Thru
      - Produces DevicePath
    - On self
      - Produces DevicePath

SCSI Driver Stack



SCSI Pass Thru Driver manages the host controller.

- Creates child handles for each bus
- Produces SCSI Pass Thru on bus handles
- Produces Device Path on bus handles

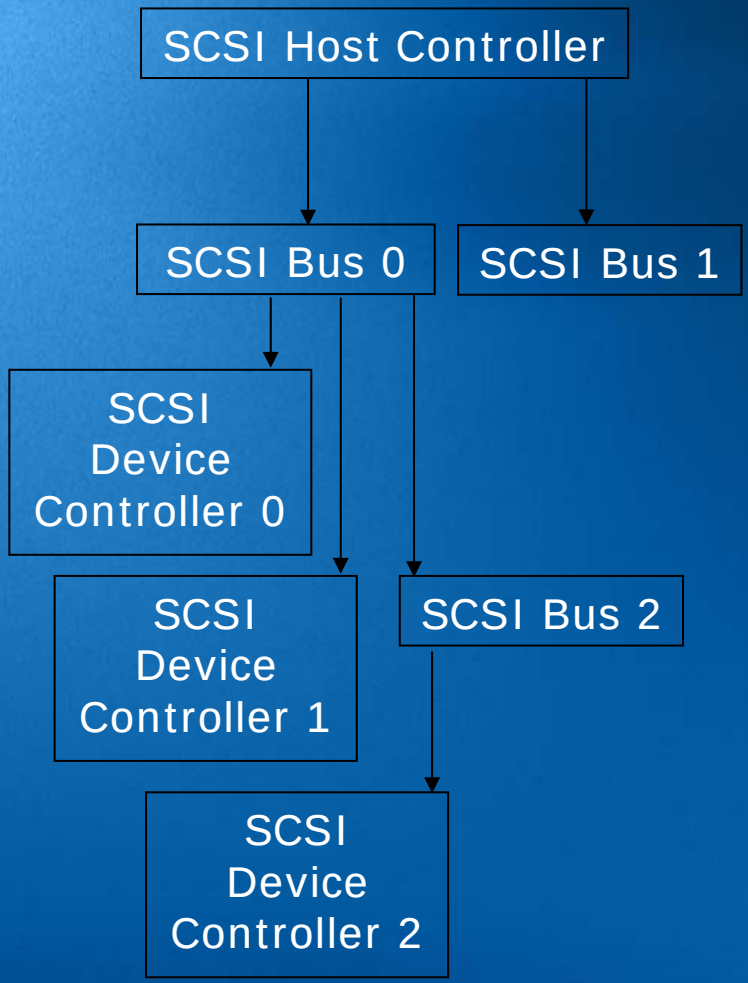
SCSI Bus Driver manages the bus (Firmware)

- Creates child handles for each device
- Passes commands down to devices
- Produces SCSI I/O on devices
- Produces Device Path on devices

SCSI Device Driver manages the device (Firmware)

- Responds to its 2 protocols

SCSI Driver Stack



OS Loader calls into Block I/O
SCSI Device Driver translates to SCSI IO
SCSI Bus Driver translates to Ext SCSI Pass Thru
SCSI HC Passes along SCSI Bus

How does the OS Loader know which drive?

DevicePath

SCSI Protocol Chart

Protocol Name	Producer (on which handle)	Consumer
Block I/O	System Firmware (Device Handle)	System Firmware
SCSI I/O and DevicePath	System Firmware (Device Child Handle)	System Firmware (SCSI Device Driver)
Ext SCSI Pass Thru and DevicePath	Host Controller Driver (Channel Child Handle)	System Firmware (SCSI Bus Driver)
Host Controller driver talks to its firmware		

- Each driver produces DriverBinding itself

SCSI Driver Stack

- PassThru() function (in Extended SCSI Pass Thru Protocol) maps to the Execute() function (in SCSI I/O Protocol) by the Bus driver.
- SCSI I/O is functionally identical to USB I/O or Disk I/O, etc...
- Example drivers in EDK