Welcome

To Advance through Presentation
Use Page Up and Page Down Keys

Mac OS USB In-Depth

Jai Chulani
Technology Manager

We'll Cover

- USB Overview
- Where USB fits into the Mac OS
- How devices describe themselves
- How drivers describe themselves
- How the Mac OS determines what driver to use

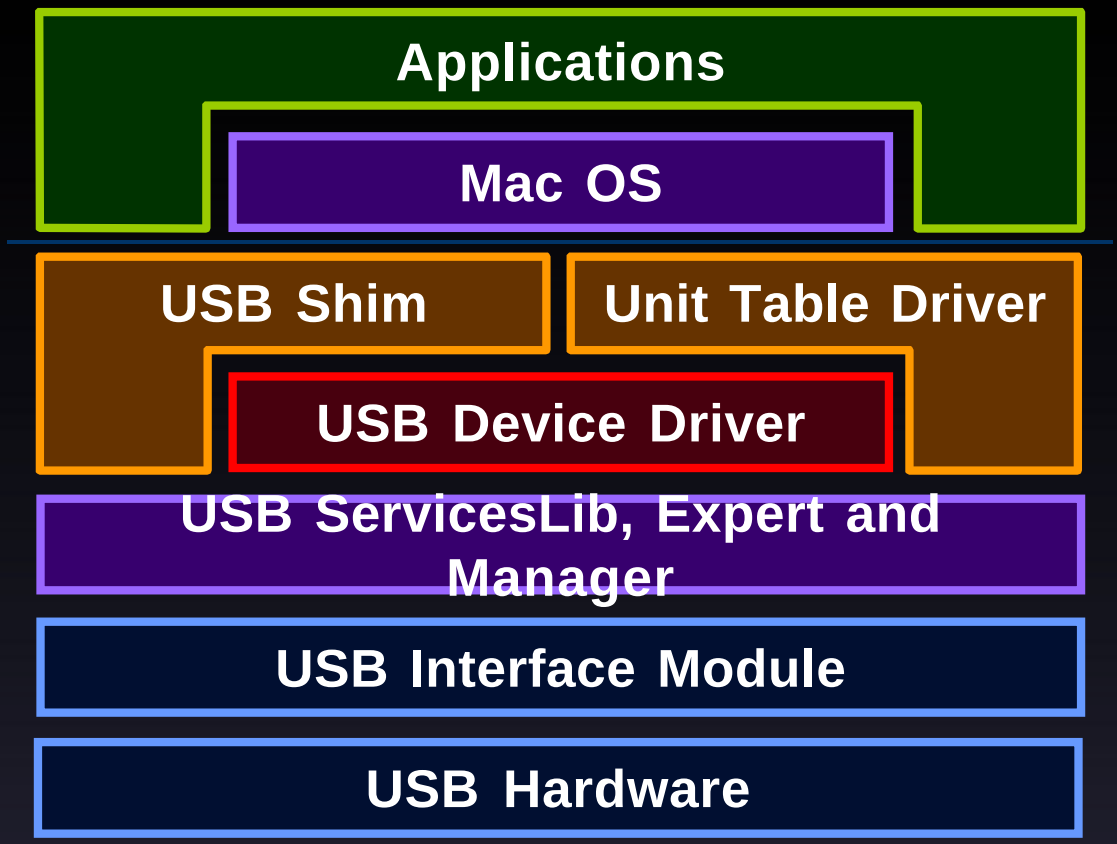
And Also. . .

- The internals of a device driver
- How drivers operate
- How drivers communicate with devices
- How drivers communicate with applications

Mac OS USB In-Depth

Craig Keithley
Technology Manager

USB in the Mac OS



USB Expert

- Handles requests by the hub driver to load class drivers
- Handles requests by the composite driver to load interface drivers
- Examines drivers and determines if they should be loaded

USB Manager

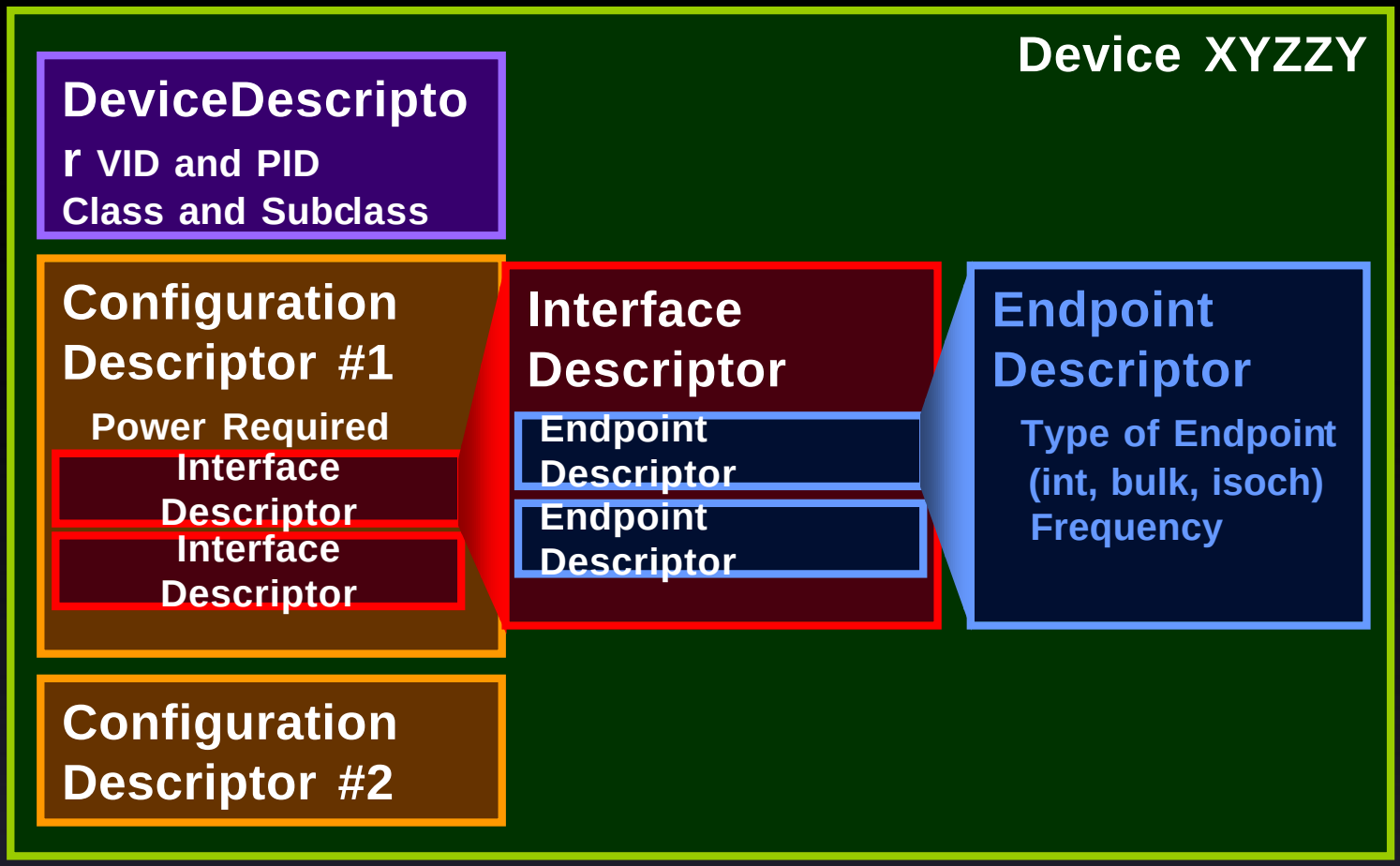
- Provides services to Shims and other clients
    - Sends notifications when devices are connected or disconnected
    - Allows the client to iterate through the USB device list and locate a particular device

Devices and Drivers

How Devices Describe Themselves

- All devices contain a “Device Descriptor”
- Devices have one or more “Configuration Descriptors”
- Configuration Descriptors usually contain one or more “Interface Descriptors”
- Each Interface Descriptor contains one or more “Endpoints”

Inside a USB Device



Demo

USB Modem

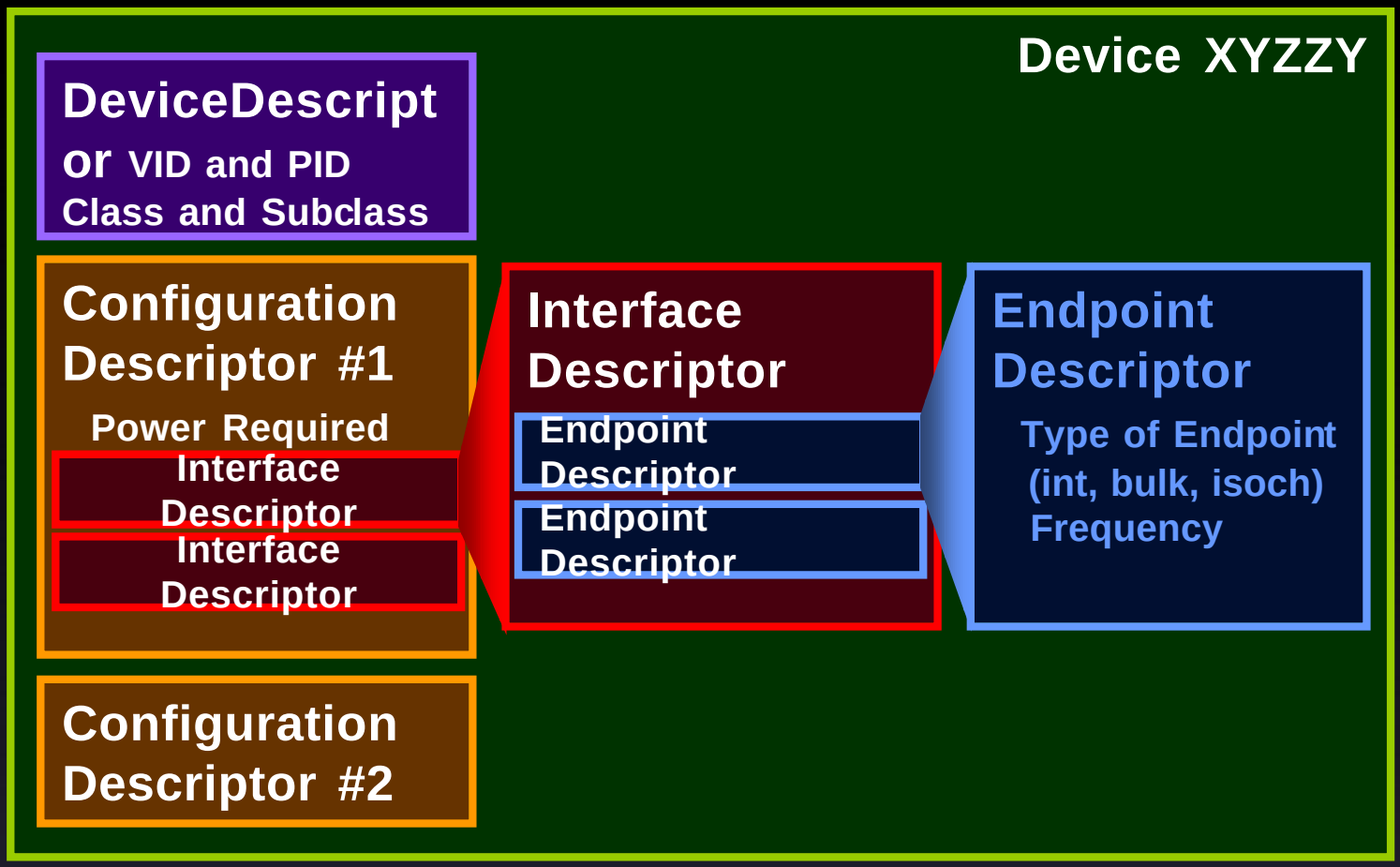
Device Class Codes

- Composite Class . . . 0
- Audio Class 1
- HID Class 3
- Display Class . . . 4
- Printing Class . . . 7
- Storage Class 8
- Hub Class 9
- Comm Class 2
- Data Class 10
- Vendor Specific . . . 255

Interface Class Codes

- Same as Device Class Codes, except…
- "Composite Class" code (0x00) is a reserved value in the interface class code field

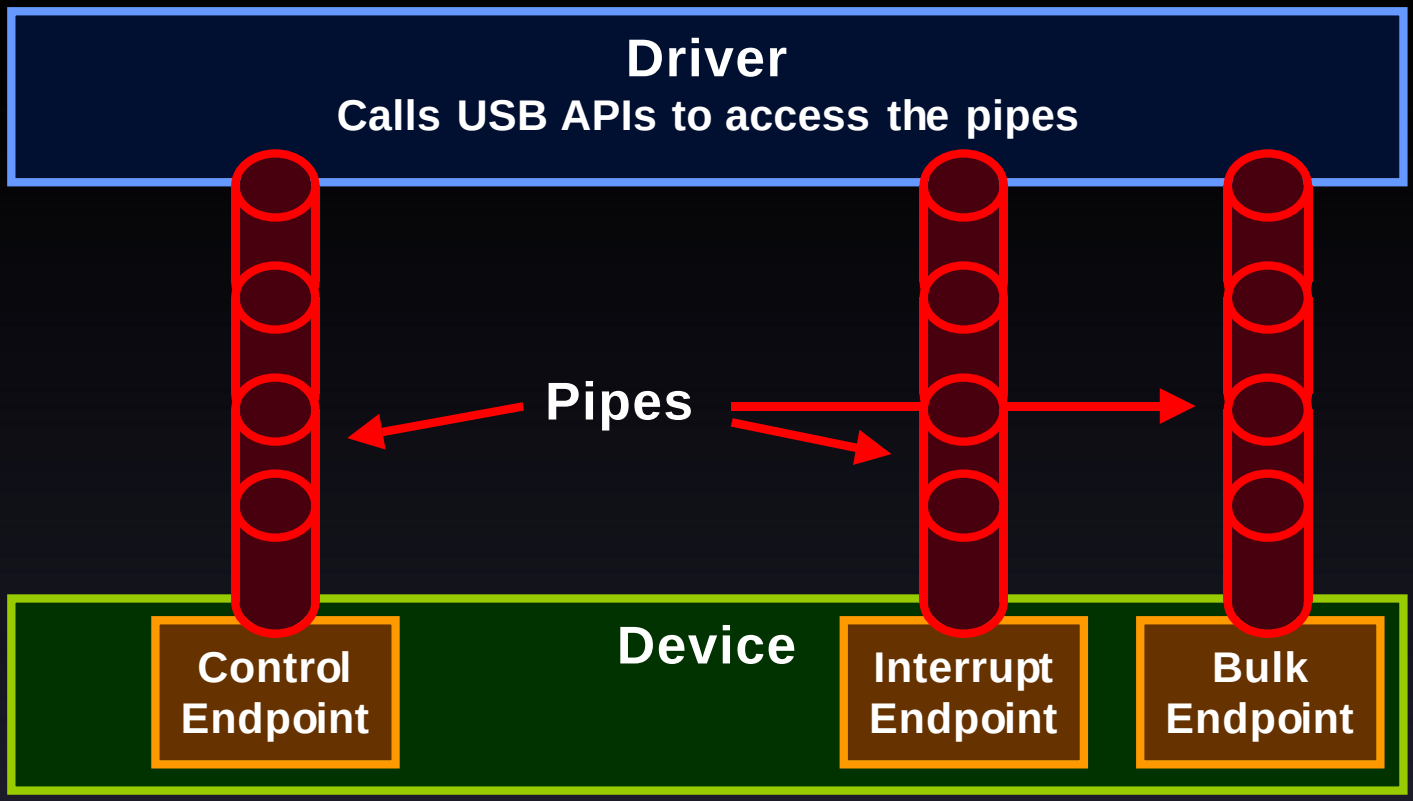
Device Review . . .



Communicating with Devices

- The connection between a driver and an Endpoint is a “Pipe”
- Four types of Endpoints
  - Control
  - Interrupt
  - Bulk
  - Isochronous

Pipes and Endpoints



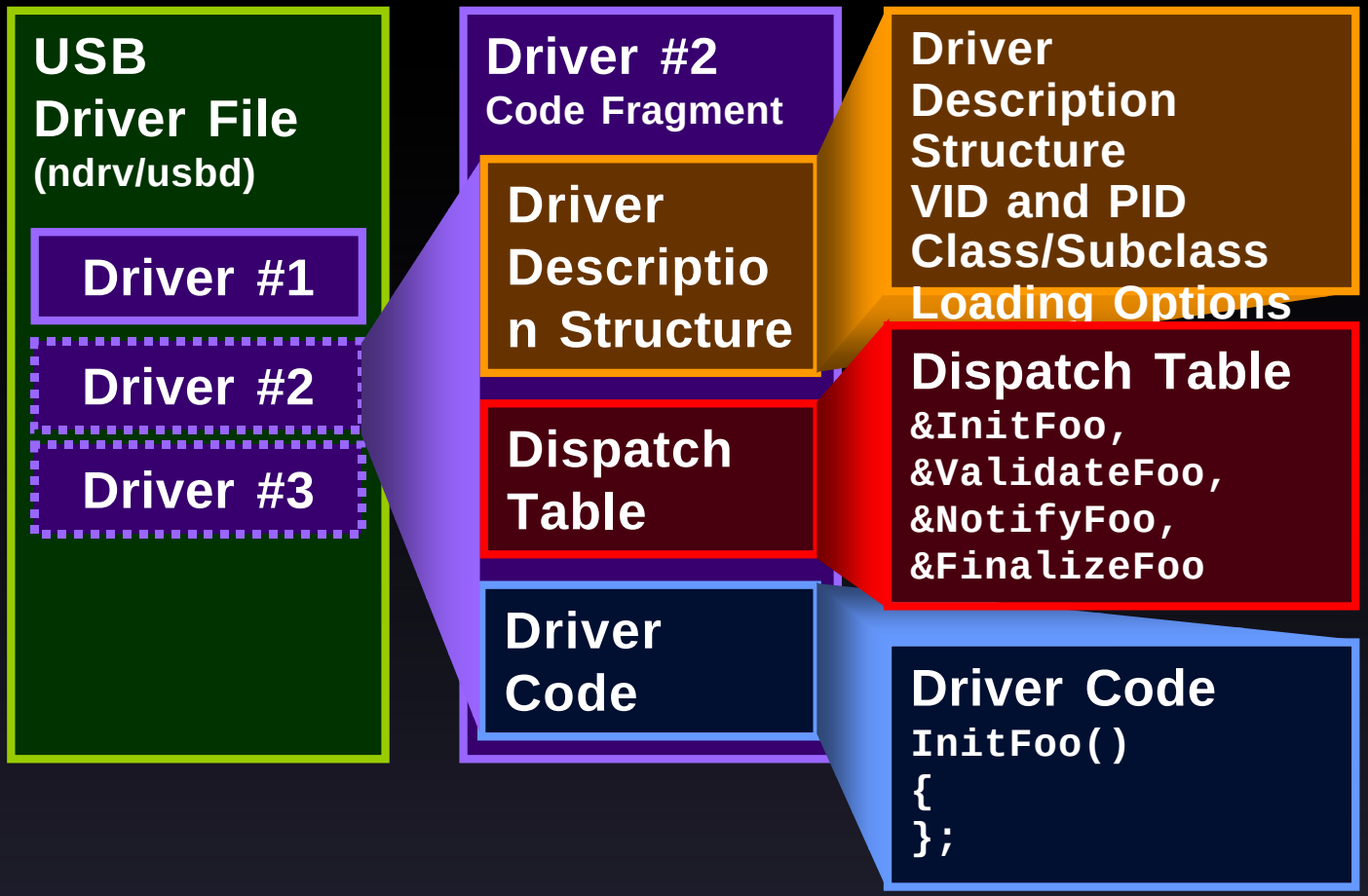
Types of Endpoints

- Control: Changes modes, gets descriptors, sets features, etc.
- Interrupt: Auto-polled periodic read or write of small packets to the device
- Bulk: Guaranteed quality, not guaranteed to be on time
- Isoch: Guaranteed to be on time, not guaranteed to be error free

Inside a USB Driver File

- One or more Driver Code Fragments
- Each Driver Code fragment contains
  - A Driver Description Structure
  - A Driver Dispatch Table
  - The Driver's code

Inside a USB Driver File



Driver Description Structure

- Vendor and Product ID of driver
- Device Class, Subclass, and Protocol
- Interface Class, Subclass, and Protocol
- Driver loading options indicate. . .
  - If the driver is Vendor Specific
  - If the driver should only load as an interface driver

Types of Drivers

- "Vendor Specific"
  - The driver loads only for your device
- "Standard Apple Drivers"
  - Generic drivers Apple supplies with the Mac OS

- Note: The recommended way to make a driver "vendor specific" is to put a Vendor ID in the driver description structure and to set the driver loading options correctly

When a Device Is Attached

- The Hub driver…
  - Detects a device attach and "enumerates" the device
  - Asks the USB Expert to load a driver
- The USB Expert…
  - Loads the driver and calls its initialization routine

Matching Drivers to Devices

- The Expert examines every driver to see if there's a potential match to the device
  - Vendor and Product IDs
  - Class/Subclass/Protocol
- Vendor Specific drivers take precedence over Standard Apple Drivers

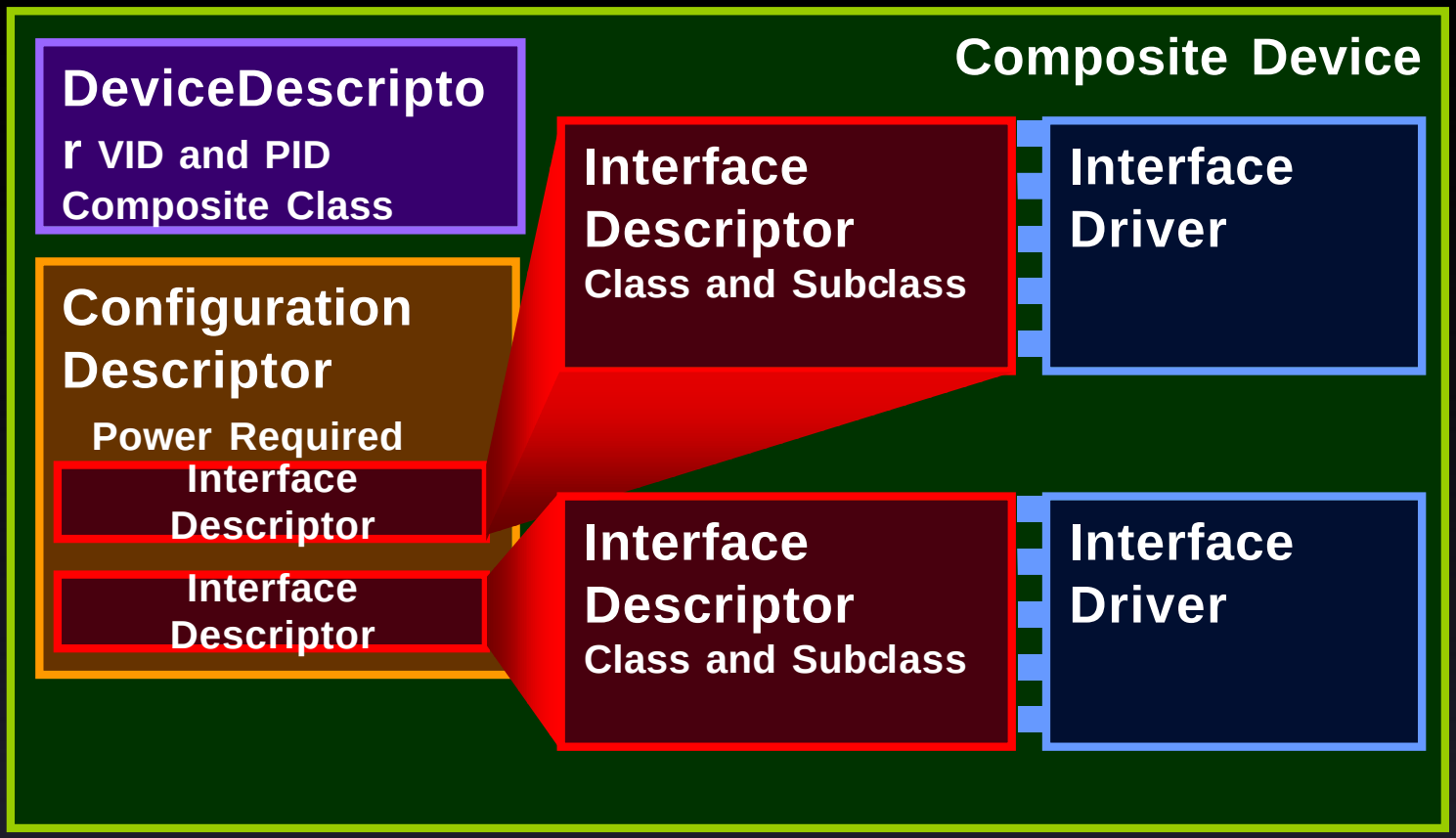
Matching Drivers (Cont.)

- Each driver's Validate Hardware routine is called
- The driver's rank is set to zero if the Validate Hardware routine returns an error
- The Expert calls the initialization routine of the highest ranking driver

Recommendations . . .

- Write Vendor Specific drivers
  - Put your Vendor ID in the Vendor ID field
  - Set the "kUSBDoNotMatchGenericDevice" flag in the Driver Loading Options field
- Put *your* Vendor ID in the device descriptor, not the chip maker's ID

A Composite Device



Device and Interface Drivers

- Device drivers must select a configuration, check power availability, and configure interfaces

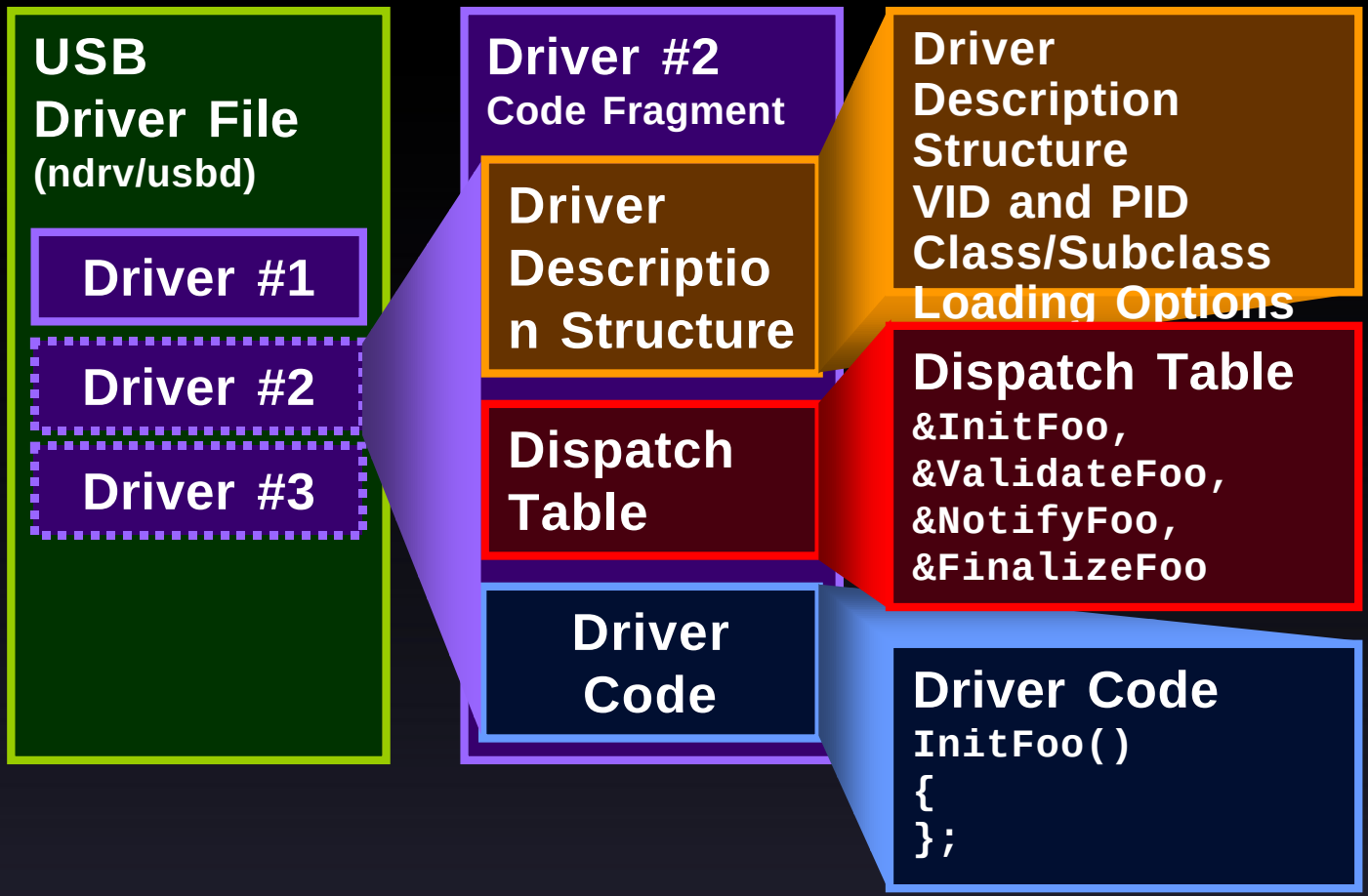
- Interface drivers don't need to select a configuration, worry about power consumption, or configure the interface

Writing a USB Driver

Writing a USB Driver

- "Vendor Specific" drivers are mandatory
- Driver writers can choose between writing a "device driver" or an "interface driver"
  - Device drivers are loaded for devices
  - Interface drivers are loaded for *interfaces* within composite devices

Inside a USB Driver File



Driver Dispatch Table

- Device Initialization routine
- Interface Initialization routine
- Validate Hardware routine
- Notification routine
- Finalize routine

Validate Hardware

- Allows the driver to report whether it should work for this USB Device
- Return kUSBNoErr if the driver is acceptable

Device Initialization

- Called when the driver is loaded for a device
- Initiates the first USB transaction (which will complete asynchronously)
- Drivers entered via this entry point must select a device configuration

Device Driver Startup

- Find the desired device configuration
- Set the configuration
- Open the interfaces in the configuration
- Find the “pipes” within the interfaces
- Prepare to read and/or write to the pipes

Selecting a Configuration

- Depends on how much power is available
- Depends on what interfaces the driver will use

- Example: Some modems have two configurations, one with vendor specific interfaces, the other with Comm and Data interfaces

Interface Initialization

- Called when the driver is loaded to control an Interface
- Initiates the first USB transaction (which will complete asynchronously)
- Drivers initialized via this entry point should not select a configuration

Interface Driver Startup

- Open the interface
- Find the “pipes” within the interface
- Prepare to read and/or write to the pipes

Writing Code

Driver Writing Basics

- Most APIs complete asynchronously
- Completion routines may be called at Secondary Interrupt time
- Don't "spin" on the param block status value
- Use only the USB Services Library and Driver Services Library

Using Asynchronous APIs

- Almost all USL APIs require a completion procedure
- The Mac OS USB DDK examples use state machines to work with the asynchronous nature of the USB APIs
- In the DDK examples, one API's completion routine just calls another API

Opaque Datatypes

- Opaque datatypes
  - DeviceRef
  - InterfaceRef
  - PipeRef
- Avoid hard coding configuration, interface, or endpoint numbers

Device Driver Initialization

```
USBClassDriverPluginDispatchTable
        TheClassDriverPluginDispatchTable =
{
    kClassDriverPluginVersion,
    DriverValidateHW,
    DeviceInitialize,
    InterfaceInitialize,
    DriverFinalize,
    DriverNotifyProc,
};
```

APIs Used in a Device Driver

- USBFindNextInterface
- USBSetConfiguration
- USBNewInterfaceRef
- USBConfigureInterface
- USBFindNextPipe

Finding the Desired Configuration

USBFindNextInterface

- Parameters:
  - The power available to the device
  - The desired interface class, subclass, and protocol
- Returns:
  - Configuration and interface numbers

Setting the Device Configuration

USBSetConfiguration

- Parameters:
  - The configuration number returned by USBFindNextInterface
- The USB Services Library does a set configuration to the device

Getting an InterfaceRef

USBNewInterfaceRef

- Parameters:
  - The interface number returned by USBFindNextInterface
- Returns:
  - InterfaceRef to the interface

Configuring the Interface

USBConfigureInterface

- Parameters:
  - The interfaceRef returned by USBNewInterfaceRef
- The USB Services Library opens all the pipes in the interface

Finding the Pipes in the Interface

USBFindNextPipe

- Parameters:
  - The interfaceRef returned by USBNewInterfaceRef
  - pipe type and direction
- Returns:
  - pipeRef to the pipe

Reading From a Pipe

USBIntRead

- Parameters:
  - The pipeRef returned by USBFindNextPipe
  - A pointer to a buffer
  - The number of bytes to read
- Returns:
  - Data read from the pipe

Interface Driver Initialization

```
USBClassDriverPluginDispatchTable
        TheClassDriverPluginDispatchTable =
{
    kClassDriverPluginVersion,
    DriverValidateHW,
    DeviceInitialize,
    InterfaceInitialize,
    DriverFinalize,
    DriverNotifyProc,
};
```

Interface Drivers

- Loaded by the composite driver
- Examples:
    - Apple's Mouse Driver
    - Apple's Keyboard Driver
- Don't need to find interfaces or configure the device
- Supplied with the interfaceRef at startup

APIs Used in an Interface Driver

- ~~USBFindNextInterface~~
- ~~USBSetConfiguration~~
- ~~USBNewInterfaceRef~~
- USBConfigureInterface
- USBFindNextPipe

Driver Notification Proc

```
USBClassDriverPluginDispatchTable
        TheClassDriverPluginDispatchTable =
{
    kClassDriverPluginVersion,
    DriverValidateHW,
    DeviceInitialize,
    InterfaceInitialize,
    DriverFinalize,
    DriverNotifyProc,
};
```

Driver Notification

- The driver's notification routine can be called to inform the driver of the following:
  - Driver Removal
  - Sleep demand
  - Sleep request

```
kNotifySystemSleepRequest    = 1,
kNotifySystemSleepDemand     = 2,
kNotifyDriverBeingRemoved    = 11
```

Removal Notification

- Drivers are notified when they are about to be removed
- A Driver may report that it is still busy
- The notification routine will be called repeatedly until the driver reports it is not busy
- The Driver's finalize routine is then called, and the driver is removed

Sleep Notifications

- *Important for PowerBooks!*
- Sleep Request allows the driver to report that it is busy, and that the system should not sleep
    - Mounted Volumes, open serial connections, etc.
- Sleep Demand indicates that the system *is* going to sleep, and the driver *will* be removed

Sleep in the Future

- USB will allow the device to *Suspend* and the driver to remain loaded
- Suspended devices consume far less power than active devices
- Suspended devices must stop communication with the device
- Expect that desktop CPUs will ultimately support USB Suspend

Communicating with Drivers

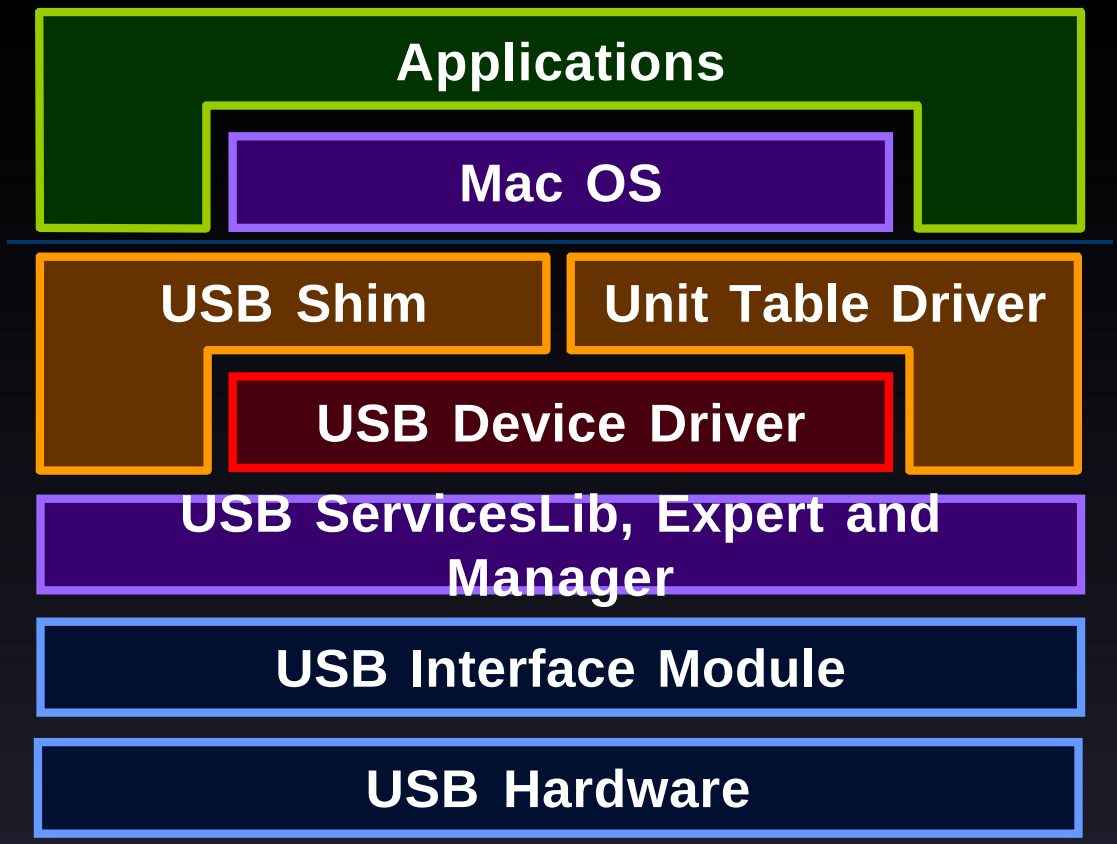
Communicating With Drivers

- Locate the driver's code fragment
- Locate a symbol within the code fragment (dispatch table or function)
- Call into the driver using the symbol
    - Read or Write data
    - Install a callback function

Who Would Call a Driver?

- USB Shims
- Unit Table Drivers
- Applications

USB in the Mac OS



USB Shims

- Always loaded and resident
- Loaded *before* the INIT parade
- Can install Unit Table Drivers

Unit Table Drivers

- Pros:
  - Provide Classic Device Manager APIs
  - Useful when emulating legacy drivers
- Cons:
  - Not needed or appropriate for every class of USB device
  - Special handling required if the unit table needs to grow

Calling USB Drivers from Applications

- *This is not recommended because…*
  - Applications should concern themselves with the *service* and not the device
  - Handling hot unplugs within an application can be problematic
  - Most USB APIs complete at Secondary Interrupt time

Reasons for Shims or Unit Table Drivers

- Shims/UTDs can handle device connect and disconnect and “buffer” the application from a disappearing driver
- Allows Applications to be unconcerned about which bus (FireWire or USB) is used

Locating a USB Driver

- Ask the USB Manager what devices are attached
- Request notification when devices are connected or disconnected

Ask the USB Manager

- Use the USBGetNextDeviceByClass API
- Parameters:
  - Device Class, Subclass, and protocol
- Returns:
  - deviceRef and code fragment connection ID

Requesting Notification

- Use the USBInstallDeviceNotification API
- Parameters:
  - Device Class, Subclass, and protocol
  - Event type, Product and Vendor Ids
  - Notification procptr
- USB Manager calls the notification proc when a matching device is connected, disconnected, etc.

Handling Notifications

- Check the notification type

```
kNotifyAddDevice        = 0,
kNotifyRemoveDevice     = 1,
kNotifyAddInterface     = 2,
kNotifyRemoveInterface          = 3
```

- Determine the deviceRef (provided in the notification param block)
- Locate the driver's code fragment

```
USBGetDriverConnectionID(&theDevRef,  &connID);
```

USBDelay

- Let's the driver wait for a specific number of USB frames
- Also used to get to "task time"
  - Some Driver Services Library functions can only be called at task time
  - i.e.: Name Registry access can only occur at task time

Memory Allocation

- USBAllocMem
- USBDeallocMem

- *Remember*: A USB Driver may be operating at secondary interrupt time, which is not safe for NewPtr and other OS APIs

Other Useful USL APIs

- Pipes
  - USBClearPipeStallByReference
  - USBAbortPipeByReference
  - USBGetPipeStatusByReference
- Device and Control Requests
  - USBDeviceRequest
  - USBMakeBMRequestType

Debugging Drivers

- Use MacsBug when possible
- Use USBExpertStatusLevel to write to the USB Expert Log
- Use the Power Macintosh Two Machine Debugger if you have legacy Macs with serial ports

Demo

USB Prober Expert Log

Command Level

Final Thoughts

Final Thoughts

- Follow the USB Specifications
- Join the USB-IF
    - http://www.usb.org
- Provide product strings in your device
- Be aware of how PowerBook sleep may affect your devices
- Work on “no-restart” installs
- Vendor specific drivers are mandatory

Related Sessions

USB Feedback Forum: Tell us what you think about USB	Hall J2 **Thurs., 9:00am**
FireWire Feedback Forum: Tell us what you think about FireWire	Hall J2 **Thurs., 2:30pm**

Think different.™

Welcome

To Advance through Presentation
Use Page Up and Page Down Keys