datronicsoft

Training Course

Windows WDF
Device Driver Development

by

datronicsoft Inc.
One Corporate Centre,
Unit 3012
Julia Vargas Avenue corner Meralco Avenue
Ortigas Center, Pasig City, 1605
Metro Manila, Philippines

Version 2.4 dated April 7th 2013

TABLE OF CONTENT

PART I – INTRODUCTION

PART II – DRIVER DEPLOYMENT AND WDF KMDF BASICS

Driver Deployment

WDF Framework Object Model

PART III – DRIVER INTERFACES TO APPLICATIONS

Request Processing and IO-Queues

Framework Memory Objects and Request Memory

PART IV – DRIVER INTERFACES TO OTHER DRIVERS

PART V – DRIVER INTERFACES TO HARDWARE

PART VI – INSTRUMENTATION, TOOLS AND QUALITY

Chapter 1

WDF KMDF Driver Development Introduction

Common OS architecture concepts

- Overall Architecture
  - Monolithic
  - Layered
  - Microkernel (Client-Server)
- User Mode vs. Kernel Mode
- Certain CPU instructions not accessible in User Mode
  - e.g. Halting the processor
  - e.g. Accessing hardware registers
- Virtual memory with certain ranges not accessible in User Mode
  - e.g. kernel address space
  - e.g. other processes' address spaces (process isolation)

History and design goals

- Designed by David Cutler (former DEC / VAX / VMS)
- Portability
  - Available on different platforms (x86, DEC-Alpha, PowerPC, MIPS, IA64)
  - Hardware Abstraction Layer (HAL)
    New "plug in" extension capability for core system components (e.g. interrupt controller) in Windows 8
  - WDM Bus drivers (up to NT4 part of the HAL)
- Microkernel concepts
  - Not a Microkernel operating system
  - Similar modularity in kernel design
  - Large OS components in user mode (Subsystems)

Windows components in user mode and kernel mode

- User mode
  - Applications
  - Subsystems
  - Services
- Interface from user mode to kernel mode
  - Windows Native API
  - Not documented
  - Should not be used in shipping products
- Kernel mode
  - WDM Drivers
  - Executive

Windows Executive

- Kernel-Mode managers and libraries
  - Kernel
  - Io Manager
  - Plug’n’Play manager
  - Power manager
  - Memory manager
  - Hardware abstraction layer
- Executable files
  - Windows\system32\ntoskrnl.exe
  - Windows\system32\Hal.dll

Tasks of device drivers

- Device drivers do...
  - respond to I/O requests from applications, other drivers or Windows components
  - respond to Plug'n'Play events from the OS
  - respond to Power Management events from the OS
  - communicate with OS kernel via DDI
  - communicate with their hardware through bus drivers or hardware resources (registers, interrupts, etc.)
- Device drivers do not...
  - have a user interface
  - have a single sequence of execution (event driven instead)

Windows device driver types

- WDM Plug'n'Play drivers (kernel mode)
  - WDM function drivers (incl. WDF-KMDF)
  - WDM filter drivers (incl. WDF-KMDF)
  - WDM bus drivers (incl. WDF-KMDF)
  - Operating system supplied port drivers
- Miniport drivers (kernel mode)
  - Network, storage, audio video streaming, etc.
- WDF UMDF drivers (user mode)
- Others (kernel mode)
  - Legacy non Plug'n'Play drivers
  - Filesystem drivers

WDM driver model

- Direct interaction with executive components
- Using a subset of the function calls exported by ntoskrnl.exe
- Component indicated by prefix letters
  - IoXxx IO manager
  - MmXxx memory manager
  - PoXxx power manager
  - ObXxx object manager
  - KeXxx kernel
  - RtlXxx runtime library
- Rich set of features

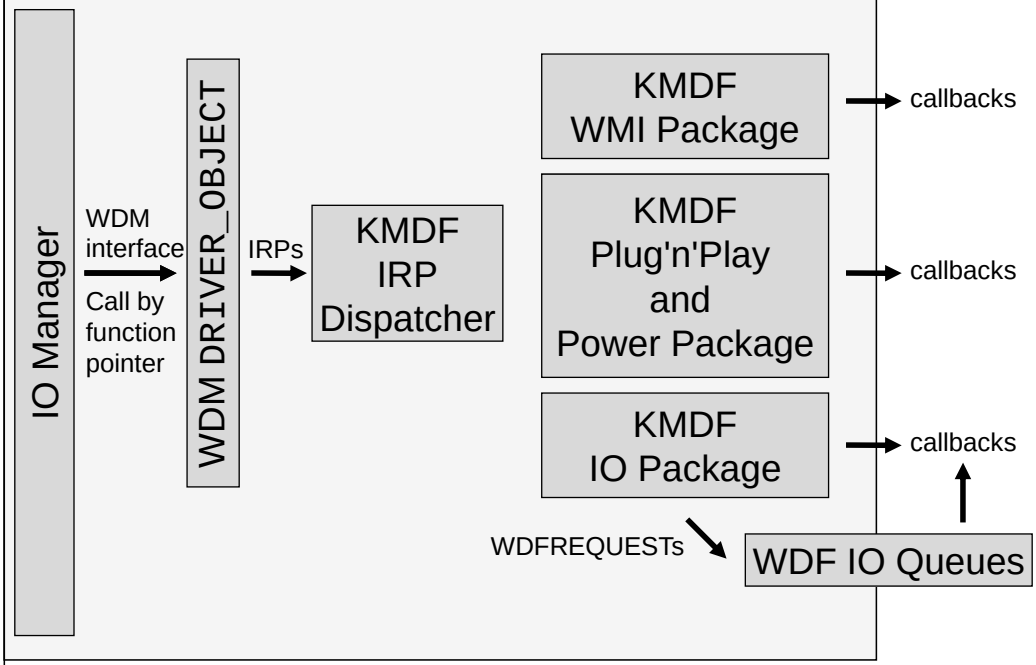
WDM driver architecture fundamentals

- File based
  - Drivers are accessed as if they were files
- Packet based
  - Each I/O operation is described by an IO-Request-Packets (IRP)
  - PnP, Power and other events are also described by IRPs
- Layered
  - Usually more than one driver layered above one hardware
  - IRPs can be handed through these layers in a device stack of layered drivers
- Asynchronous

Windows Driver Foundation (WDF)

- WDF is a library for simplifying WDM driver development
- WDF supports user mode drivers and kernel mode drivers
- WDF provides a consistent object model for user mode and kernel mode drivers
- WDF object model supports
  - Handling, queuing and cancellation of IO requests
  - Full Plug’n’Play state machine with default implementation
  - Full Power Management state machine with default implementation
  - Many more infrastructure tasks

WDF KMDF Overview



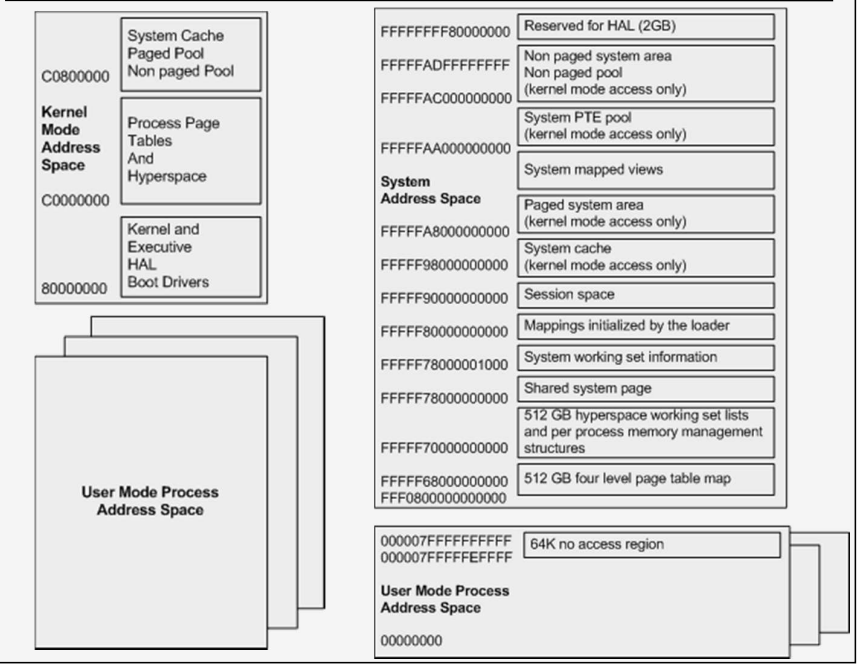
Process model

- Process isolation
- Memory protection
  - No access to operating system memory
  - No access to other processes' address spaces
  - Memory cleanup after process termination
- Process specific object handles
  - Handle cleanup after process termination

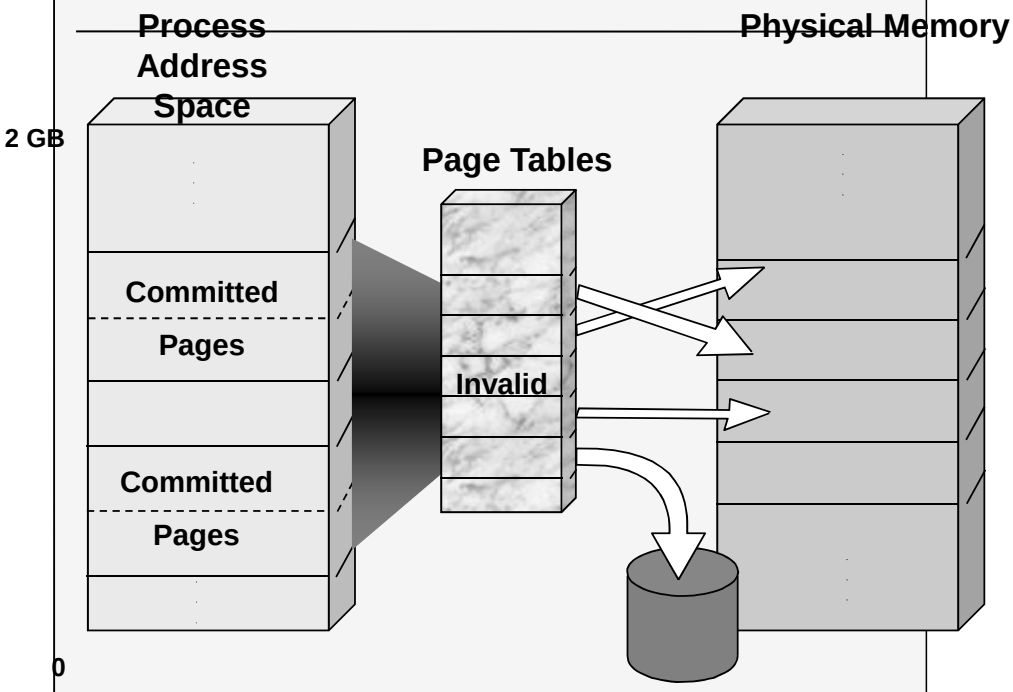
Virtual Memory

- Extension of physical RAM by paging file
- Protection
  - Protection of operating system memory
  - Process isolation
- One single privileged address space for the operating system
  - Applications cannot corrupt the OS
- Separate isolated address space for each user mode application
  - Applications cannot corrupt each other

X86 and x64 address space



Virtual Memory Fundamentals



Object based architecture

- Object based resource access and usage
  - Polymorphic CreateXxx(…) functions returning handles
  - Interprocess resource usage through names in Object Manager's namespace
- Tool for viewing Object Manager's namespace: winobj.exe (download from Sysinternals)
- Polymorphic synchronization through dispatcher objects

Windows trap handling

- Hardware independent trap handling model
- Distinguishing four different traps
  - Exceptions
  - System calls
  - Hardware interrupts
  - Virtual memory exceptions
- Exceptions raised by violating thread
  - Can be handled by the application itself using SEH (Structured Exception Handling)
  - Default handler shows dialog box and calls `ExitProcess()`

Chapter 2

Driver Installation

WDM driver package / setup disk

- Minimum driver installation disk (WDM only) contains
  - .inf file
  - .sys driver executable
- .cat file (digital signature)
- Optional Co-Installer.dll (mandatory for WDF)
- Optional executable which is written using
  - Setup API
  - DIFx API
  - SCM API
- Optional Microsoft installer setup (MSI)
  - Application centric setup packet in .msi file
  - Sometimes also used for installing drivers

Driver setup and signatures

- Unsigned drivers
  - 32 Bit x86 install => RED warning dialog box
  - 64 Bit x64 NO install
- Driver signed with own SPC
  - 32 Bit x86 install => grey dialog (do you trust?)
  - 64 Bit x64 install => grey dialog (do you trust?)
- Microsoft certified drivers (Hardware Certification Kit)
  - No user interaction necessary
  - Unattended install possible

Driver setup scenarios

- Hardware first
  - Manual installation (New Device DLL)
  - Installer application (DIFx Api, Setup Api, msi)
  - Application has to reinstall plugged and unplugged devices
- Software first
  - Installer application (DIFx Api, Setup Api, msi)
- Software only (virtual device - no physical hardware)
  - Device Manager option “Add Legacy Hardware”
  - Installer application (Setup Api, SCM Api, msi)
  - Service Control Manager (SCM) Api for filter drivers and legacy drivers

Driver Store

- `Windows\System32\DriverStore`
- Software first scenario
  - Driver Store contains all files of the driver package
  - `OEMxx.inf` file in `Windows\INF` directory pointing to driver package in driver store
  - Package is located when device is plugged
- Package is also added upon hardware first and software only scenarios (Vista and above)
- Pnputil tool
  - List all third party driver packages `pnputil –e`
  - Add package (software first scenario) `pnputil -a`

INF files

- Header
  - OS Selection, Version, etc.
- Manufacturer Section
  - Information about manufacturer
  - Points to models section (device list)
- Models section (device list)
  - Specify matching driver for hardware
  - Association between Hardware ID and DDInstall section
- DDInstall section
- DDInstall.Services section

WDK tool for INF file syntax check: chkinf.bat

Device detection and identification

- Detection by Plug'n'Play capable buses
  - Bus specific standard protocol
- Hardware ID
  - Detected during PnP bus enumeration by bus specific standard protocol
  - Bus specific unique identifier
  - Format: `<bus>\<busspecific-device-ID>`
- Instance ID (unique on one machine)
  - OS generated
  - Device generated if supported by device

Device categories and setup classes

- Low level device categorization by PnP bus protocol
- Windows OS categorization by Setup Classes
  - Defined by GUID
  - Many predefined system setup classes as e.g. CD-ROM Drives, Display Adapters, Human Interface Devices, Keyboard, Mouse, Network Adapter, Ports, Printer etc.
  - Class "Other Devices" is reserved for system. Must not to be used by device drivers
- If necessary vendors can define their own class
- Defined in the registry under HKLM\SYSTEM\CurrentControlSet\Control\Class

Co-Installer and Class-Installer Dlls

- Class-Installer
  - Usually operating system supplied (only if absolutely necessary should vendors define own classes and supply class installer)
  - Registry entry Installer32 under the Class key
- Class- and Device-Co-installers
  - Usually supplied by vendor to enhance installation and configuration of device
  - Registry entry under Class for Class-Co-Installers
  - Registry entry under Enum for Device-Co-Installers
- Optional user interface components
  - Class installer and co-installers can contain property sheets
  - Class installer can contain custom icon for device manager

Class- and Co-Installer interface

```
DWORD CALLBACK
XxxCoInstallerEntry(
                DI_FUNCTION InstallFunction,
                HDEVINFO DeviceInfoSet,
                PSP_DEVINFO_DATA DeviceInfoData,
                PCOINSTALLER_CONTEXT_DATA Context
                )
{
…
}
```

- DIF function codes
  - `DIF_INSTALLDEVICE`
  - `DIF_REMOVE`
  - `DIF_PROPERTYCHANGE`
  - `DIF_ADDPROPERTYPAGE_ADVANCED`
  - etc.

Driver registry keys

- The driver service key
  - HKLM\SYSTEM\CurrentControlSet\Services
  - Contains a key for every installed driver and service on the system
  - Only registry key for legacy drivers and filter drivers
- The driver hardware key
  - HKLM\SYSTEM\CurrentControlSet\Enum\Bus\PNPID
  - Is a representation of detected devices on all buses
  - Is present even before driver is installed
- The driver software key
  - HKLM\SYSTEM\CurrentControlSet\Control\Class
  - Contains an instance for each installed Plug'n'Play driver of a device setup class

Driver Install Framework DIFx

- Scope
  - Available only for signed function drivers and class filter drivers
  - Not available for filter drivers and legacy drivers
- Driver store for "Software First" scenario
  - Pnputil command line tool
- DPInst
  - Wizard generated and wizard based setup installer for simple driver packages
- DIFx Api
  - Programming interface for setup applications
- DIFxApp
  - For integration into msi based setup
  - Msi Custom Action dll

Setup API

- Typical operations
  - Enumeration of devices
  - Setting of device properties
  - Updating driver for Plug‘n‘Play devices
- Devcon.exe Tool
  - Command line device manager
  - Based on Setup API
  - Executable available in the WDK\tools directory
  - Source code available in the WDK\src\setup directory
  - Comprehensive source code example for Setup Api

WDF KMDF driver setup

- Installation disk contains
  - INF file
  - Driver executable (.sys)
  - WDF Co-installer
- WDF Co-installer
  - Located in the redist directory of the WDK
  - Device Co-installer for WDF Plug‘n‘Play drivers
  - Installs the correct version of WDF runtime on the target system if not present
  - WDF Co-installer needs manual invocation in case of legacy drivers and filter drivers

Special cases driver setup

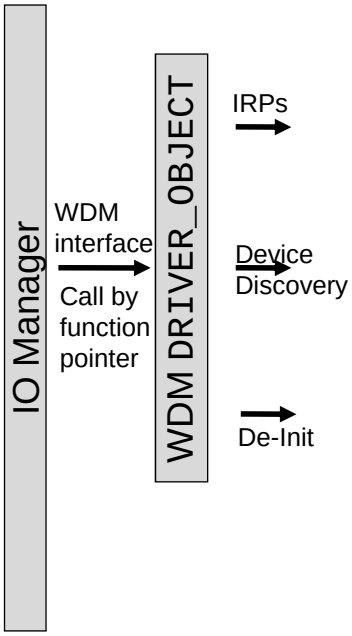
- Filter drivers
- Legacy drivers (old Windows NT 4 style)
  - Are still supported by current OS in limited way
  - Are installed by SCM (Service Control Manager) API and usually don't need INF file
  - Do not get any hardware resources assigned by Plug'n'Play OS
  - Usually cannot control hardware

Chapter 3

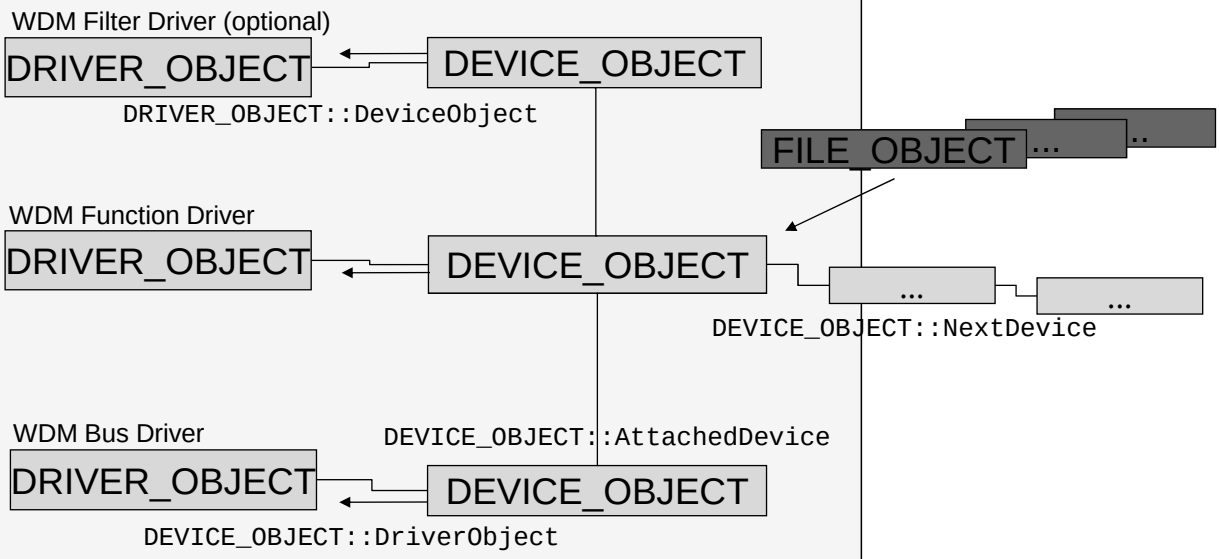
Object Models

WDM DRIVER_OBJECT

- List head for device instances which are controlled by this driver
- Entry points (function pointers) for function call interface to IO Manager
  - `PDRIVER_DISPATCH MajorFunction[]` IO interface for `IRP`s (IO Request Packets)
  - `PDRIVER_ADD_DEVICE AddDevice` function pointer in `PDRIVER_EXTENSION DriverExtension` member
  - `PDRIVER_UNLOAD DriverUnload`



WDM object model



WDF object based framework design

- All functionality grouped around Objects
  - Organized hierarchically (parent/child relationship)
  - Represent drivers, devices, queues, etc
  - Reference counted with clearly defined lifetime
  - Optional destructor callback when reference count reaches zero
  - Extension memory / object-context for storing variables
- Interaction with objects using
  - Methods – makes the object do something
  - Events – makes the driver do something
  - Properties – setting parameters

Framework object methods and properties

- Methods
  - Perform operations on the object
  - Have the conceptual this pointer as their first parameter
  - Optionally have a status return value of type `NTSTATUS`
  - Macro for checking status: `NT_SUCCESS(…)`
- Instead of direct access drivers are getting and setting property values through “can’t fail” calls
  - Set returns VOID
    `VOID WdfObjectSetValueName(WDFOBJECT Object)`
  - Get returns the value
    `ValueName WdfObjectGetValueName(WDFOBJECT Object)`

Framework object events

- Called by WDF when events occur
- Defined by driver as callback routines
- Implemented as C function pointers registered with the object
- Default defined by WDF if no handler is registered by the driver
- Driver callbacks should return the appropriate `NTSTATUS` error code upon failure

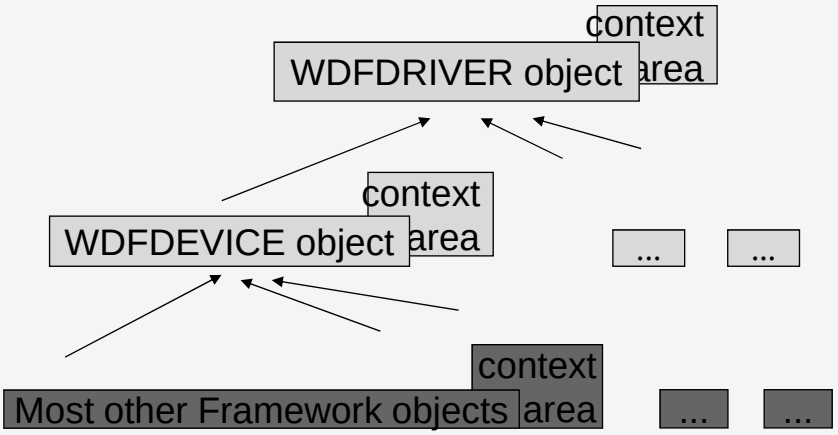
WDFDRIVER object

- `WdfDriverCreate`, initializes WDF and creates `WDFDRIVER` Object
- Provides two events
  - `EvtDriverDeviceAdd`
    Called when Plug'n'Play Manager detects a new device instance
  - `EvtDriverUnload` (optional)
    Called when the last device instance of this driver has been removed and the driver is unloading. Required only for additional driver clean-up or unload actions; typically WDF performs all necessary clean-up and object deletion

WDFDEVICE object

- `WdfDeviceCreate` creates `WDFDEVICE` Object
- Target for most operations
  - IO from user mode applications
  - Requests from other drivers
- Typically 1 to n device objects are associated with one driver object
- Parent of most other WDF objects

WDF Object Model



WDFQUEUE, WDFINTERRUPT, etc.

Framework object hierarchy

- WDFDRIVER object is the root, WDFDEVICE is parent of most other objects
- Some objects must always be children of WDFDEVICE or its descendants
- Some objects can have different parents
- Hierarchy controls life time
  - Parent holds reference count on each child
  - When the parent object is deleted, the child objects can be deleted automatically
- Hierarchy controls synchronization
  - Framework automatic synchronization of IO based on hierarchy
  - Synchronization scopes (device or queue)

Framework object lifetime

- WDF objects are reference counted by the framework which manages object lifetime.
- Pointer for access can be retrieved through the object handle
- Drivers can optionally extend object lifetime by using:
  - `VOID WdfObjectReference(IN WDFOBJECT Handle);`
  - `VOID WdfObjectDereference(IN WDFOBJECT Handle);`
- Object lifetime events support cleanup:
  - `EvtObjectDestroy`
  - `EvtObjectCleanup`

KMDF object context area

- KMDF objects can have one or more context areas
- Allocated from non paged pool
- Pointer for access can be retrieved through the object handle
- To use a context area:
  - Declaration of the context area's type `WDF_DECLARE_CONTEXT_TYPE_WITH_NAME`
  - Assignment of the context area upon object creation `WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE`
  - Later assignment of the context area to the object (support for multiple object contexts) `WdfObjectAllocateContext(…)`

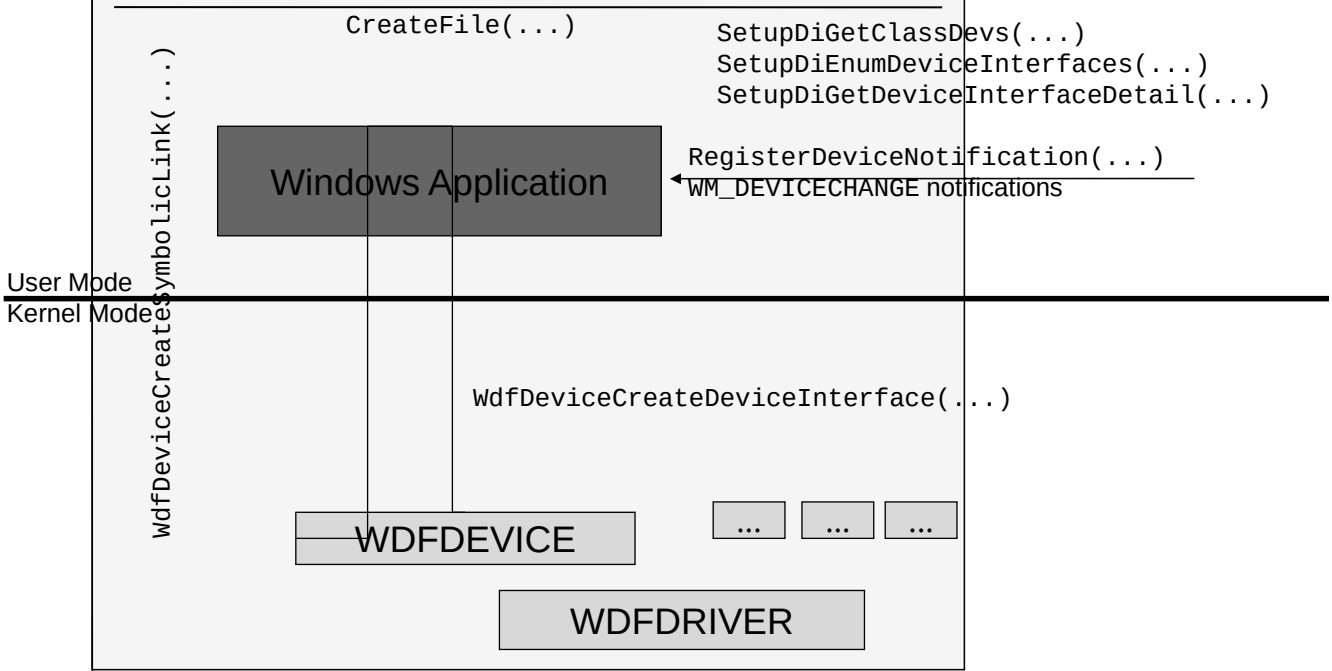
Implementation of default behaviour

- Typical standard behavior provided by WDF objects' default implementation
- Driver can override defaults to provide its own behavior
- Framework provides default handlers for most events

Chapter 4

Request Processing and IO-Queues

Device to application interfaces



Plug'n'Play Device Interface

- Identification by a globally unique ID (GUID)
- For applications to access the device
  - Driver registers device interface
  - Application enumerates device interface
  - Applications get symbolic link name from device interface
  - Applications use this auto generated symbolic link name to open the driver
- Plug'n'Play capability
  - Driver can enable and disable device interface
  - Applications can subscribe for notifications about device arrival and devices removal indication

File based driver model

- Same programming interface as file IO
- Win32 API file calls
  - `CreateFile(…)`
  - `ReadFile(…)`
  - `WriteFile(…)`
  - `CloseHandle(…)`
  - `DeviceIoControl(…)`
  - `CancelIo(…)`
- Asynchronous IO
  - `WaitForSingleObject(…)`
  - `GetOverlappedResult(…)`

Packet based driver model

- IO Manager creates an IO Request Packets (IRP) for each IO operation
- WDM IRPs are represented in WDF by Framework Request Objects
- Most important IN parameters of an IO operation
  - Buffer
  - Buffer length
- Most important OUT parameters of an IO operation
  - Status
  - Count bytes processed

WDF Request objects

- Packets of different type `WDF_REQUEST_TYPE`
- I/ O Requests presented by Queue object (handled by driver)
  - `WdfRequestTypeRead`
  - `WdfRequestTypeWrite`
  - `WdfRequestTypeDeviceControl`
- Requests presented by Device object (rarely used - usually handled by framework)
  - `WdfRequestTypeCreate`
  - `WdfRequestTypeCleanup`
  - `WdfRequestTypeClose`
- Requests NOT presented (handled by framework)
  - `WdfRequestTypePnp`
  - `WdfRequestTypePower`

Request processing

- Three fundamental ways of handling requests in a WDM or WDF driver
  - Completing immediately
  - Keep pending and queue for later completion
  - Forward to other (usually next lower) driver
- Returning request to IO Manager
  - `WdfRequestComplete(…)`
  - `WdfRequestCompleteWithInformation(…)`
  - Request MUST NOT be touched any more after completion

Request presentation by WDF Device object and File objects

- I/ O Requests presented by framework Device object
  - `EvtDeviceFileCreate`
  - If no callback supplied, CreateFile() always allowed by default implementation
  - Driver can define exclusive open policy without need for callback definition
  - Delivery of File object for tracking application handles
- Requests presented by framework File object
  - `EvtFileCleanup`
    (used for running in context of application)
  - `EvtFileClose`
- Rarely used. Typically these Requests are handled by framework default implementation

WDF IO-Queue objects for Request presentation

- Queue incoming requests
- Queue events act as request handlers presenting incoming requests to the driver
  - `EvtIoRead`
  - `EvtIoWrite`
  - `EvtIoDeviceControl`
  - `EvtIoInternalDeviceControl`
  - `EvtIoDefault`
- Drivers can actively extract requests from IO queues using `WdfIoQueueRetrieveNextRequest()`

IO-Queue Request type assignment

- Default queue of a device
  - All different request types for the device are forwarded to this queue
  - `WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE`
- Explicitly assigning a request type
  - Only a specific request type for the device is forwarded to this queue
  - `WDF_IO_QUEUE_CONFIG_INIT`
  - `WdfDeviceConfigureRequestDispatching()`

IO-Queue Request presentation

- Two different presentation models
  - Pull model (method calls)
  - Push model (event callbacks)
- Pull model
  - No delivery events (except when an empty queue gets a request)
  - Driver optionally selects specific Requests in the Io-Queue
  - Driver extracts Requests from Io-Queue
- Push model
  - Requests presented by
  - Io-Queue callbacks to signal new/next Request
  - Request delivery can be paused and resumed

IO-Queue dispatch types

- Three different request concurrency models indicated by `WDF_IO_QUEUE_DISPATCH_TYPE`
- Serial model `WdfIoQueueDispatchSequential`
  - Requests are presented by queue callbacks
  - Single sequentialized request at a time
  - Presentation of next request only after completion of previous request
- Manual model `WdfIoQueueDispatchManual`
  - Driver retrieves requests at its own pace
- Parallel model `WdfIoQueueDispatchParallel`
  - Requests are presented by queue callbacks
  - Requests are delivered to the driver as they arrive

IO-Queue concurrency and dispatch types

- Concurrency models
  - Many Requests at a time (concurrent)
  - One Request at a time (serial)
  - Combination of both
- Concurrent
  - `WdfIoQueueDispatchParallel`
- Serial
  - `WdfIoQueueDispatchSequential`

Request cancellation

- Whenever a request is kept for a longer time it must be cancellable because
  - Applications can call `CancelIo(...)` on asynchronous IO operations
  - Threads or applications with pending synchronous IO operations can be terminated
- Correct handling and synchronization of IO request cancellation is one of the most difficult things in WDM driver development
- WDF queues support request cancellation by default
  - Framework manages cancellation of request in the queue by default
  - Request removed from the queue are not cancellable any more by default
  - Default behavior can be overridden

Cancellation scenarios

- External termination of threads that are blocking in a synchronous IO operation (e.g. by Task Manager)
- Cancellable asynchronous IO operations using Win32 API call `CancelIo(…)`
- e.g. implementation of synchronous IO with timeout in user mode

Further information

- I/O Flow and Dispatching in WDF Drivers:
  - http://msdn.microsoft.com/en-us/library/windows/hardware/gg463271.aspx
- IRPs and IRP handling:
  - www.microsoft.com/whdc/driver/kernel/IRPs.mspx
- IRP cancellation:
  - www.microsoft.com/whdc/driver/kernel/cancel_logic.mspx
  - msdn.microsoft.com/en-us/library/aa480216.aspx

Chapter 5

Framework Memory Objects and Request Memory

Accessible Memory for Drivers

- Pool memory (two options allowed for drivers)
  - `PagedPool`
  - `NonPagedPool` (limited resource)
- Kernel stack
  - Limited to approximately 3 pages
  - Cannot dynamically grow
  - `IoGetRemainingStackSize(…)`
  - `IoGetStackLimits(…)`

Buffered IO and Direct IO

- Data Transfer between driver and application
- Buffered I/O (usually for small chunks of data)
  - I/O Manager allocates secondary buffer in NPP
  - I/O Manager is copying buffer content
  - Driver only uses secondary buffer
- Direct I/O (usually for large chunks of data)
  - I/O manager verifies the application buffer
  - I/O-Manager locks application buffer
  - I/O-Manager creates MDL (Memory Descriptor List)
  - Driver can map MDL to system virtual memory
  - Driver can use MDL for DMA

Method for read and write Requests

- For read operations (`IRP_MJ_READ`) and write operations (`IRP_MJ_WRITE`)
  - `WdfDeviceInitSetIoType(…)`
  - Transfer method set in Framework device object upon driver initialization

```
typedef enum _WDF_DEVICE_IO_TYPE
{
    WdfDeviceIoUndefined = 0,
    WdfDeviceIoNeither = 0,
    WdfDeviceIoBuffered,
    WdfDeviceIoDirect
} WDF_DEVICE_IO_TYPE,
PWDF_DEVICE_IO_TYPE;
```

Method for IOCTL Requests

- For I/O control operation `IRP_MJ_DEVICE_CONTROL`
  - Evaluated for each type of I/O control code
  - Control codes have to be generated by macro
  - `#include<devioctl.h>`
  - Flags for Specifying the transfer method: `METHOD_BUFFERED METHOD_IN_DIRECT METHOD_OUT_DIRECT METHOD_NEITHER`

```
CTL_CODE(
      IOCTL_XXX_BASE,
      0x008,
      METHOD_BUFFERED,
      FILE_READ_DATA|FILE_WRITE_DATA
      );
```


WDF Memory objects

- WDM memory
  - For local buffers
  - Allocated by `ExAllocatePoolWithTag(…)`
- WDF memory object
  - For IO buffers (transparent to Buffered IO and Direct IO)
  - For buffers that are passed to other components
- Pool Tag
  - Can be specified for each WDF memory object
  - Can be specified on a per driver base `DriverPoolTag` is a member of `WDF_DRIVER_CONFIG`
  - Pool Tag utility traces driver memory allocations

WDF Request memory functions

- WdfRequestRetrieveInputMemory()
- WdfRequestRetrieveOutputMemory()

- WdfRequestRetrieveInputBuffer()
- WdfRequestRetrieveOutputBuffer()

- WdfRequestRetrieveOutputWdmMdl()
- WdfRequestRetrieveInputWdmMdl()

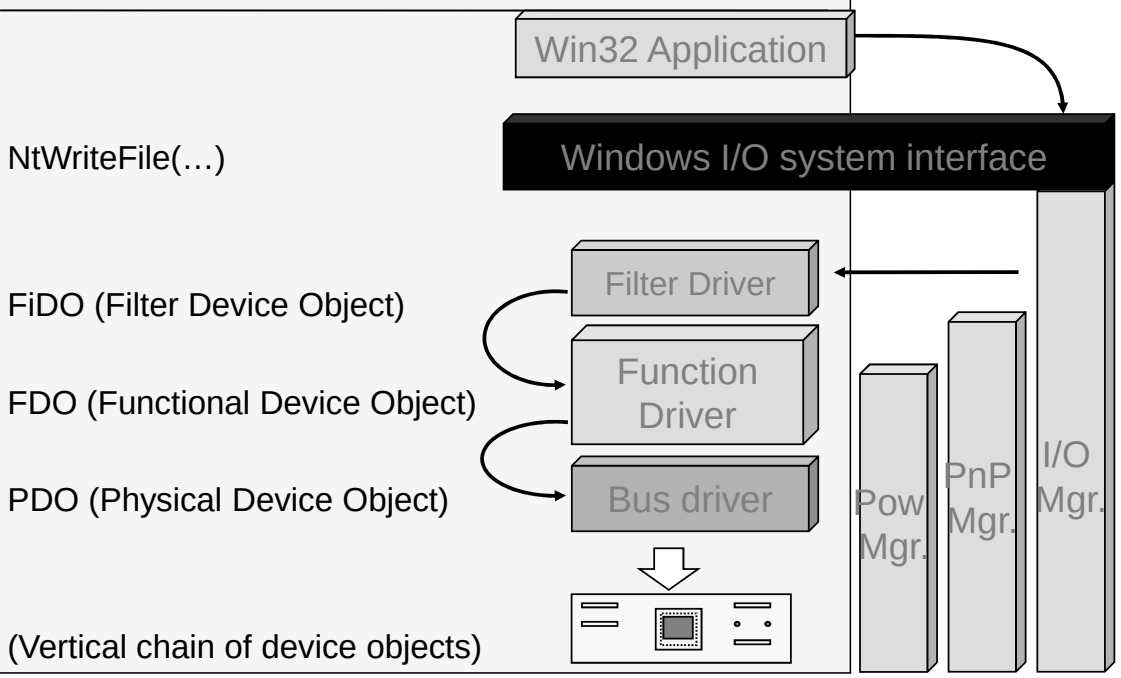
Chapter 6

Interaction between drivers

Driver roles

- Function driver (typically vendor supplied)
  - Access point for application
  - Implements the functionality
  - Core of the device stack
- Filter driver
  - Enhance, restrict or change functionality or hardware features
  - Optional within the WDM driver model
- Bus driver (typically OS supplied)
  - Supplied by Windows for all common buses e.g.: PCI, USB, FireWire (IEEE 1394), etc.
  - Root bus driver enumerates virtual devices without any hardware

Device stacks built by layered drivers



Reasons for a layered driver model

- Division of labor between function driver and bus driver
  - Standard Plug'n'Play functionality in Bus Driver (typically operating system supplied)
  - Device specific functionality in Function Driver (typically vendor specific)
- Routing of Events through the whole driver stack
  - Plug'n'Play events e.g. device start / device remove
  - Power Management events e.g. power up / power down
- Filter drivers supply standard hooking capabilities

Device stack attachment

- Attachment information
  - Function driver is associated with the PDO by hardware ID from the INF file
  - Filter driver is associated with the device stack by registry entries
- Attachment time
  - During buildup of the device stack
  - After creation of device object
- Implicite attachment in KMDF
- Explicit attachment in WDM
  - WDM drivers need to attach explicitly by `IoAttachDeviceToDeviceStack()`
  - Plug'n'Play manager supplies PDO to attach to in WDM `AddDevice()` routine

Device stacks and KMDF

I/O request types

- Framework delivers to the driver
- IO Queues presenting IO Requests
- Automatic forward only in filter drivers

- Plug'nPlay and Power Management Requests
  - Framework does not deliver to the driver
  - Framework automatically forwards to the next lower layer

WDM Function Driver Request Processing

- Power Requests `WdfRequestTypePower (IRP_MJ_POWER)`
  - Power device up and down
  - Event handling for power state changes
  - Typically forward all Power Requests
- Plug'nPlay Requests `WdfRequestTypePnp (IRP_MJ_PNP)`
  - Turn device on and off
  - Event handling for Plug'n'Play state changes
  - Typically forward all Plug'n'Play Requests
- IO Requests
  - Typically handled by WDM Function Driver
  - Typically completed by WDM Function Driver

WDM Bus Driver Request processing

- Power Requests `WdfRequestTypePower (IRP_MJ_POWER)`
  - Receive and handle all Power Requests
  - Power up / power down bus slot
- Plug'nPlay Requests `WdfRequestTypePnp (IRP_MJ_PNP)`
  - Receive and handle all Plug'n'Play Requsts
  - Some Plug'n'Play Requests only for bus drivers
  - Activate/deactivate bus slot
- IO Requests e.g. `WdfRequestTypeRead (IRP_MJ_READ)`
  - Typically do not receive nor handle IO Requests Exception: Raw PDO (e.g. HID game controllers)

WDM Filter Driver Request processing

- Filter drivers typically simply pass most requests to the next lower driver they are filtering
- Request processing on the way down
  - inspection and possibly modification of requests
  - Completion of request without forwarding
- Request processing on the way up
  - Registering a completion routine on the way down
  - Inspection and possibly modification of requests after lower driver has processed and completed them
  - Possibly regain ownership of the request again for later completion
- Sideband communication through control device object

Filter driver categories and setup

- `IoAttachDeviceToDeviceStack(..)`
- WDM Filter Drivers (early attachment)
  - Registry value `UpperFilters/LowerFilters`
  - For a whole device class - underneath Class key
  - For an individual device - underneath Enum key
  - Attachment during target device startup
- Non-WDM Filter Drivers (late attachment)
  - After buildup of the target WDM device stack
  - WDM and Non-WDM targets possible
  - Active, explicit attachment during run time
- Setup installation similar to Non-WDM drivers by
  - SCM API
  - `Services` section in INF file of other driver

KMDF support for WDM filter drivers

- KMDF implicit attachment after `EvtDriverDeviceAdd`
- WDF default forwarding of I/O requests that don't have an associated I/O Queue `WdfFdoInitSetFilter(…)`
- Requests received through the I/O Queue have to be completed or passed down to the next lower layer.

KMDF IO-Targets for inter driver communication

- IO Targets are device objects to send requests to. Can be used for a variety of different device objects
  - Default IO-Target for next lower device
  - General IO-Targets and Specialized IO-Targets (e.g. USB)
  - Remote IO-Target for device not related to own device stack
- IO Targets are associated with a queue to store requests that currently can't be delivered
- Monitor PnP state of targeted device object, queuing sent requests when appropriate
- Track pended and sent requests

Asynchronous Request forwarding

- Passing a request to the next lower driver
  - Preparing the request using `WdfRequestFormatRequestUsingCurrentType()`
  - Asynchronous send using `WdfRequestSend()`
- No request completion necessary after using `WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET`
- Request completion after lower driver request completion
- Information about lower driver request completion `WdfRequestSetCompletionRoutine ()`

Synchronous Request forwarding

- Forwarding and waiting for completion
  - `WdfIoTargetSendReadSynchronously()`
  - `WdfIoTargetSendWriteSynchronously()`
  - `WdfIoTargetSendIoctlSynchronously()`
  - `WdfIoTargetSendInternalIoctlSynchronously()`

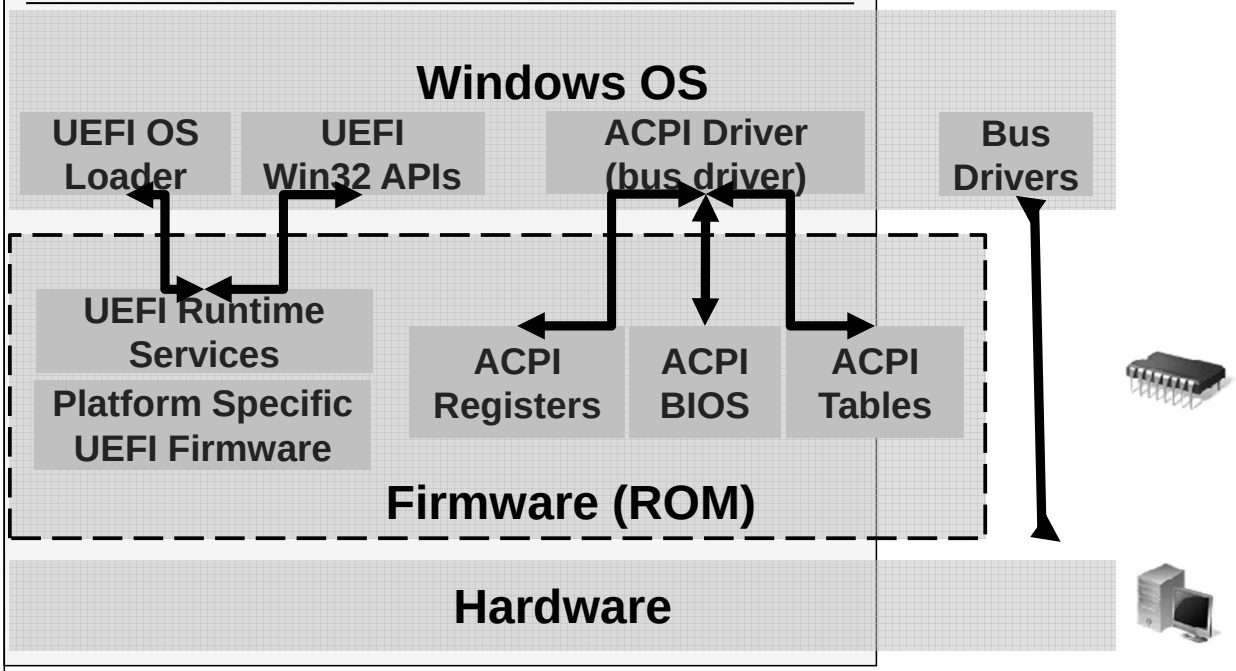
Communication between unrelated drivers

- Plug'n'Play device interface notifications in kernel mode
  - Subscription for device arrival notifications
  - Obtaining symbolic link name and reference to target driver's device object
  - Device discovery
- Plug'n'Play device handle notifications in kernel mode
  - Subscription for device query remove and removal notifications
  - Monitoring Plug'n'Play state of referenced device object
- Sending requests as long as device state permits

Chapter 7

Plug'n'Play

Hardware and firmware interfaces



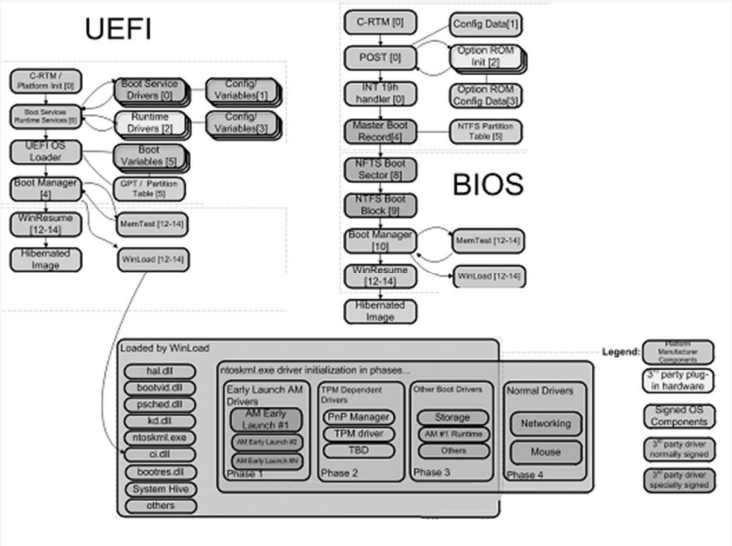
UEFI firmware protocols

- For disk boot
  - Block I/O protocol
  - Device Path protocol
- For input/output
  - Simple input protocol
  - Graphics output protocol
  - Simple text output protocol
- Graphics mode entered by Windows OS loader
  - OS loader preferably uses GOP

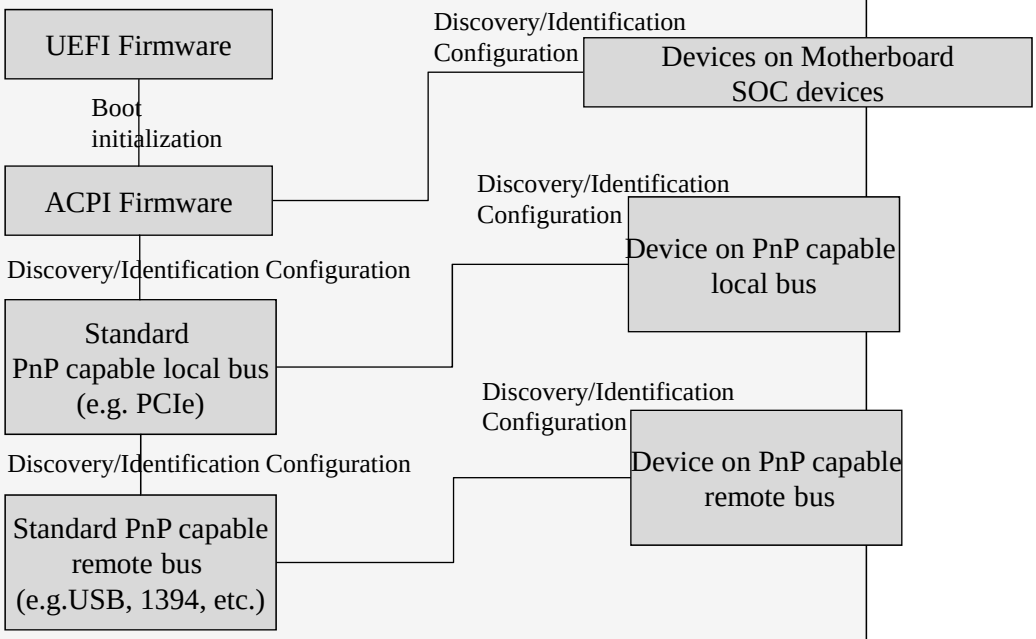
ACPI

- ACPI complements UEFI
  - UEFI mainly used during bootstrap
  - ACPI used at runtime
  - Limited runtime usage of UEFI

Windows Boot Sequence (I)



Windows Boot Sequence (II)



Plug'n'Play capable standard buses

- Device detection, identification and configuration
- Bus specific standard for device detection
  - Windows supplies bus drivers for all common buses e.g.: PCI, USB, FireWire (IEEE 1394), etc.
  - Root bus driver enumerates virtual devices without any hardware
- Bus specific Vendor IDs and Device IDs
  - E.g. `PCI\VEN_XXXX&DEV_YYYY`
- Device configuration
- PCI bus example: WinDbg command `!PCI`

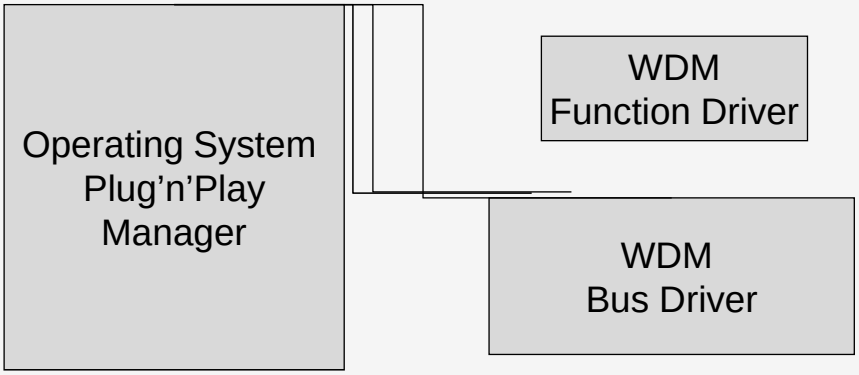
Windows Plug’n’Play software components

Runtime Plug’n’Play device state change events

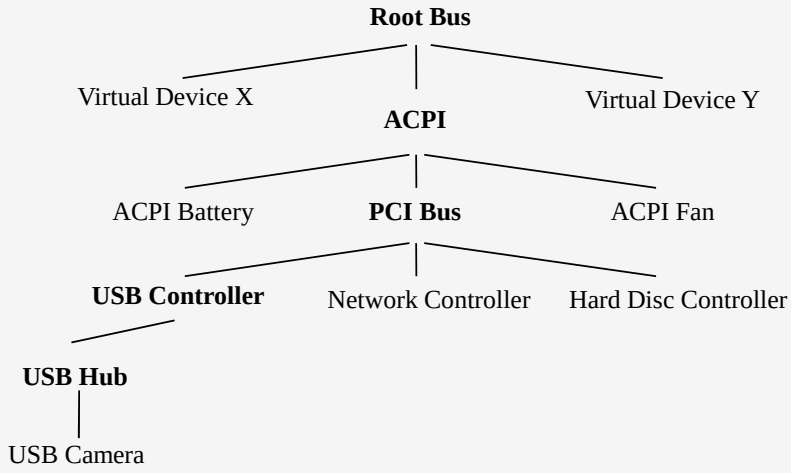
Initialization Plug'n'Play device capabilities
Plug'nPlay hardware configuration

Initialization Plug'n'Play device discovery
Plug'nPlay device identification

IRP_MJ_PNP



Plug'n'Play Manager's device tree



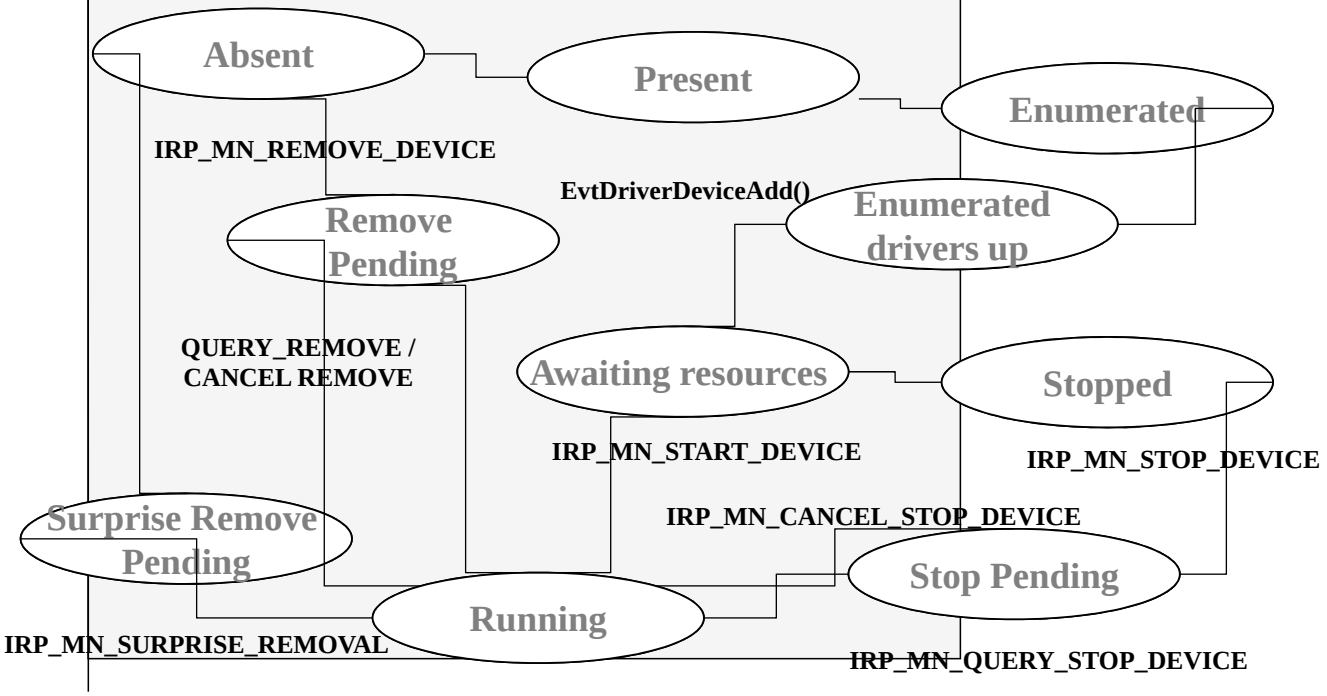
Scenario "New device detected"

- Sequence
  - 1. New hardware connected
  - 2. Bus driver recognizes new device
  - 3. Bus driver creates PDO
  - 4. Bus driver informs kernel mode PnP Manager
  - 5. PnP Manager asks bus driver for child list
  - 6. PnP Manager asks new PDO about details
  - 7. PnP Manager extends device tree
  - 8. User Mode PnP Mgr. installs function driver
  - 9. Invocation of Class Installer and Coinstaller(s)
  - 10. Kernel Mode PnP Mgr. loads function driver
  - 11. Kernel Mode PnP manager sends PnP events

WDM Plug'n'Play Events

- Plug'n'Play Manager transmits events by `IRP_MJ_PNP`
- Events handled by bus drivers only
  - Function drivers ignore and forward
  - e.g. `IRP_MN_QUERY_ID`
- Events handled by function drivers only
  - e.g. `IRP_MN_FILTER_RESOURCE_REQUIREMENTS`
- Events handled by Function drivers and bus drivers
  - Requests from Power Manager
  - `IRP_MN_START_DEVICE`
  - `IRP_MN_REMOVE_DEVICE`

WDM Plug’n’Play states and events



WDF Plug'n'Play related events

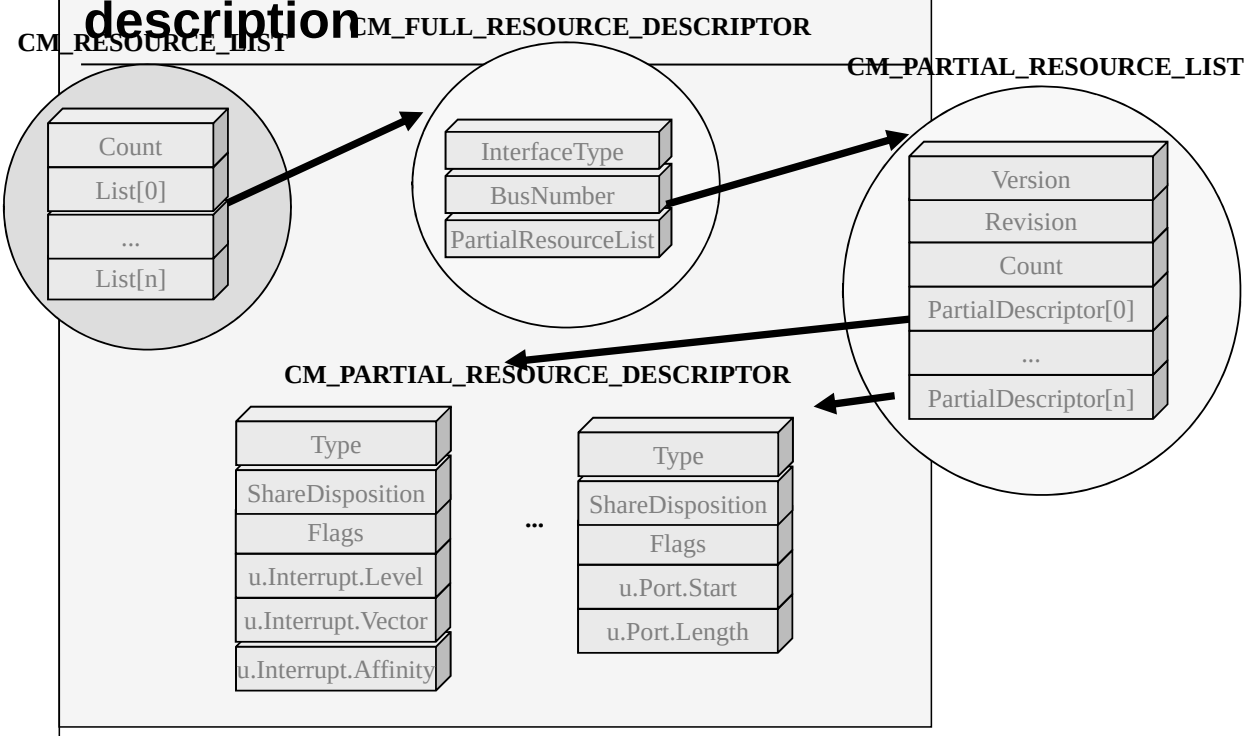
- Framework does NOT deliver Plug'n'Play Requests of type `WdfRequestTypePnp`
- For all devices which have memory-mapped registers
  - `EvtDevicePrepareHardware`
  - `EvtDeviceReleaseHardware`
- For all USB devices
  - `EvtDevicePrepareHardware`
- For all devices which use interrupts
  - `EvtInterruptEnable`
  - `EvtInterruptDisable`

WDF Plug'n'Play states

```
typedef enum _WDF_DEVICE_PNP_STATE {
    WdfDevStatePnpInvalid = 0x00,
    WdfDevStatePnpObjectCreated = 0x100,
    WdfDevStatePnpCheckForDevicePresence = 0x101,
…
…
    WdfDevStatePnpRestart = 0x131,
    WdfDevStatePnpRestartReleaseHardware = 0x132,
    WdfDevStatePnpRestartHardwareAvailable = 0x133,
    WdfDevStatePnpPdoRestart = 0x134,
    WdfDevStatePnpFinal = 0x135,
    WdfDevStatePnpNull = 0x136,
} WDF_DEVICE_PNP_STATE, *PWDF_DEVICE_PNP_STATE;
```

- Retrieval
  - `WdfDeviceGetDevicePnpState(...)`
- Subscription for state change events
  - `WdfDeviceInitRegisterPnpStateChangeCallback(...)`

Plug'n'Play hardware resource description



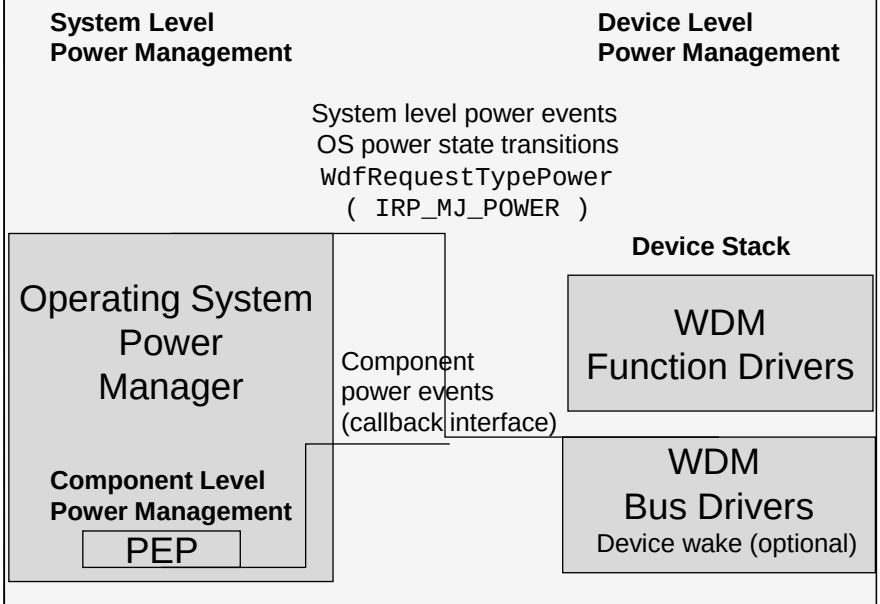
Plug'n'Play hardware resource assignment

- Framework Resource-List Object
  - Holds one or more resource descriptors `WdfCmResourceListGetCount()`
  - Descriptors are retrieved by `WdfCmResourceListGetDescriptor()`
  - Each descriptor holds information about one register or one interrupt
- Plug'n'Play drivers don't actively look for their hardware resources.
- Plug'n'Play drivers receive their hardware resources upon Plug'n'Play startup event
  - WDF: `EvtDevicePrepareHardware(…)`
  - WDM: `IRP_MN_START_DEVICE`

Chapter 8

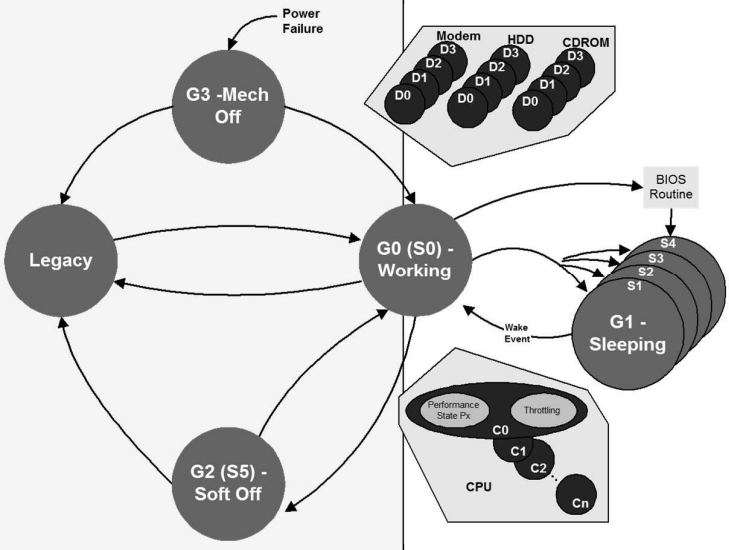
Power Management

Power management OS and software components



ACPI Power state classes and power states

- Global Power States
  - Prefix ‘G’ / G0 – G3
- System Power States
  - Prefix ‘S’ / S0 – S5
- CPU Power States
  - Prefix ‘C’ / C0 – C3
- Bus Power States
  - Prefix ‘B’ / B0 – B3
- Device Power States
  - Prefix ‘D’ / D0 – D3



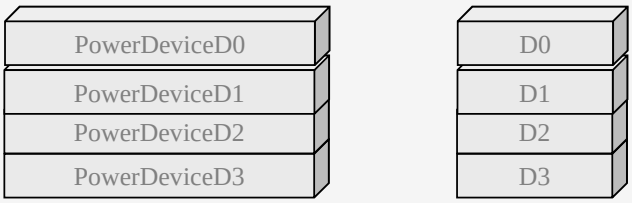
WDM system power states

- State transitions always have to go through S0

Enum name	ACPI State
PowerSystemWorking	S0
PowerSystemSleeping1	S1
PowerSystemSleeping2	S2
PowerSystemSleeping3	S3
PowerSystemHibernate	S4
PowerSystemShutdown	S5

WDM device power states

- No limits for state transitions
- Implementation of D0 and D3 (other states optional)



- D0 = Fully Powered on
- D1 = Sleeping (almost on)
- D2 = Sleeping (almost off)
- D3 = Off

Device power policy

- Power management for device stack
  - One driver is device “Power Policy Owner”
  - Typically function driver is PPO
- Required PPO functionality
  - Determine appropriate D-Power state mapping for each S-Power state
- Optional PPO functionality
  - Device idle power down
  - System wake up via `IRP_MN_WAIT_WAKE`

WDM power management events

- WDM Power-Events
  - `IRP_MJ_POWER`
    `IRP_MN_SET_POWER /IRP_MN_QUERY_POWER`
- WDM System- and Device-Power-Events

```
typedef enum _POWER_STATE_TYPE {
    SystemPowerState = 0,
    DevicePowerState
} POWER_STATE_TYPE, *PPOWER_STATE_TYPE;
```

  - System Power Events sent by Power Manager
  - Device Power Events sent by Power Policy Owner or Power Manager (idle management)

KMDF power management for D-States

- Framework does NOT deliver Power Management Requests of type `WdfRequestTypePower`
- WDF treats Plug'n'Play and Power Management with a unified model (as opposed to WDM)
- Power policy ownership
  - By default framework establishes function driver as power policy owner
  - Explicit setting during initialization `WdfDeviceInitSetPowerPolicyOwnership()`
- Power Management events of framework device object
  - `EvtDeviceD0Entry`
  - `EvtDeviceD0Exit`

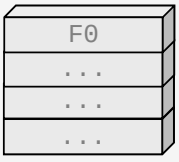
Power Managed IO Queues

- Request dispatching only in D0 state
- IO Queues are by default
  - Power managed in Power Policy Owner
  - Not power managed if not PPO (e.g. filter driver)
    Caution: Potential deadlock above PPO
- Benefits of power managed queues
  - Dispatch requests only in D0 state
  - Pause request dispatching in D1-D3 states but queuing doesn't stop
  - Queue events indicating stop and resume of request dispatching
    `EvtIoStop`
    `EvtIoResume`
  - Support for idle detection

Component Power Management

Optional (only available on Windows 8)

- Component Power States
  - ACPI (5.0) definition only for F0



  - F0 = Fully powered on
  - Power Policy Owner defines further states (F1,F2,..)
- Component Power Events (Windows 8 only)
  - Sent by Power Engine Plugin (PEP)
  - Extending previous OS capabilities

Power Management Framework (PoFx)

- Windows 8 Device Driver Interface (DDI)
- Device power management
- Device run time power management
  - Using D-states
  - Device idle detection
- Component level power management
  - Using F-States
  - Side by side with device level power management
  - Only while device is in D0 state
  - Multiple components per device

PoFx Functions

- Registration with PoFx
  - `PoFxRegisterDevice()`
  - `PoFxStartDevicePowerManagement()`
- `PoFxReportDevicePoweredOn()`
  - Must be invoked by the driver
  - Either after a system power transition to S0
  - Or after `DevicePowerRequiredCallback` callback
- `PoFxSetDeviceIdleTimeout()`
- Activation/Idle reference counting
  - `PoFxActivateComponent()`
  - `PoFxIdleComponent()`

PoFx Callbacks

- Component power callbacks
  - `ComponentActiveConditionCallback`
  - `ComponentIdleConditionCallback`
  - `ComponentIdleStateCallback`
- Device power callbacks
  - `DevicePowerNotRequiredCallback`
  - `DevicePowerRequiredCallback`

KMDF support for PoFx

- KMDF supplies only a thin wrapper around PoFx
- Device run-time power management (using D-states)
- Component power management (using F-states)
  - Limited WDF support (single component only)
- KMDF source samples: SingleComp and MultiComp

Chapter 9

Hardware on USB Bus

Device stack interaction with hardware

- Data transfer to and from device
  - Bus driver only
  - Function driver does not interact with hardware directly
  - Function driver only communicates with bus driver
- Device configuration
  - Bus driver
- Device detection
  - Bus driver

USB fundamentals

- Descriptors
  - Device descriptor (used by bus driver)
  - Configuration descriptor (used by function driver)
  - Interface descriptor (used by function driver)
  - Endpoint descriptor (used by function driver)
- Four different pipe (endpoint) types
  - Control transfer
  - Bulk transfer
  - Interrupt transfer
  - Isochronous transfer
- Default control pipe (endpoint 0) is always present
- Additional pipes (endpoints) are configured and selected by the device driver

USB bus driver interface

- USB bus driver interface through internal IOCTL
  - `IRP_MJ_INTERNAL_DEVICE_CONTROL`
  - `IOCTL_INTERNAL_USB_SUBMIT_URB`
  - USB Request Blocks (URBs)
- USB device configuration
- USB data transfers to and from device

USB spezialized IO-Targets

- Creating and selecting configuration and interface
  - `WdfUsbTargetDeviceCreate(…)`
  - `WdfUsbTargetDeviceSelectConfig(…)`
- Get access to an endpoint (pipe)
  - `WdfUsbInterfaceGetConfiguredPipe(…)`

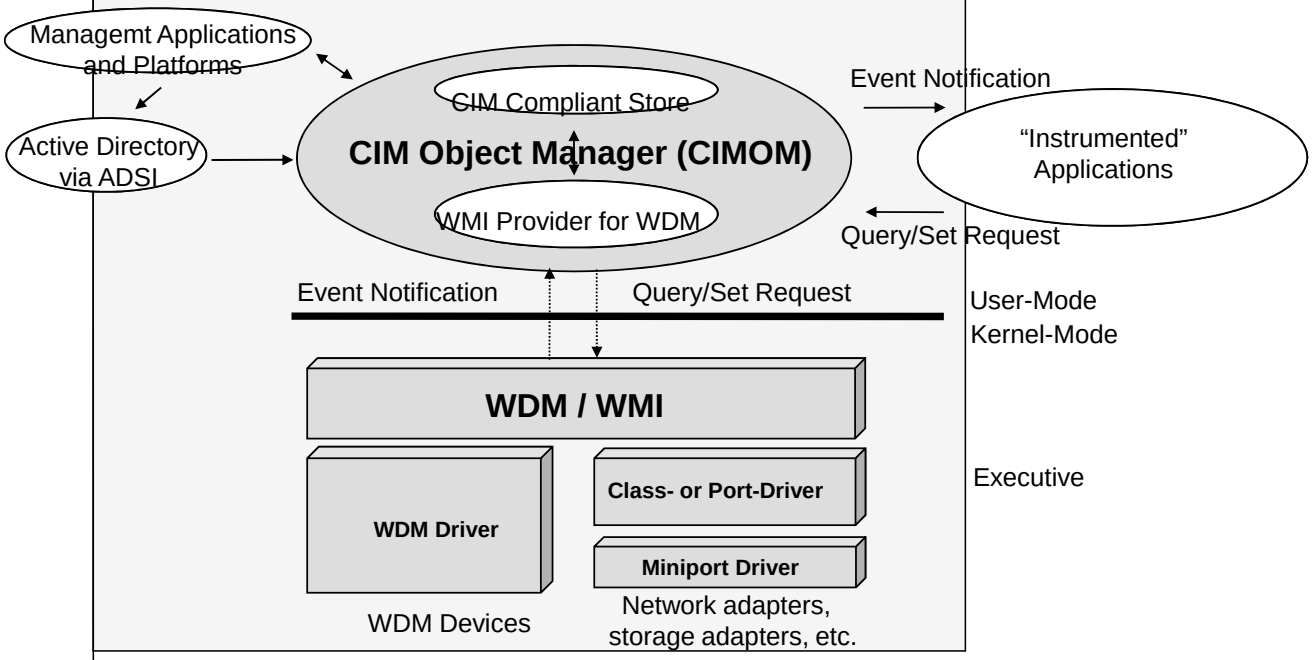
Chapter 10

WMI

WMI Introduction

- WMI is Microsoft's implementation of the Web-Based Enterprise Management (WBEM) and Common Information Model (CIM) standards from the Distributed Management Task Force (DMTF).
- Layered approach (three layers) for collecting and distributing management data
- Transmission of management data from drivers and applications to a management application (mostly statistic-, diagnostic- and performance information)

WMI Overview



Application access to WMI data

- COM interfaces
  - `IwbemServices`
  - `IwbemLocator`
  - `IWbemClassObject`
- Asynchronous event notification
  - `IwbemObjectSink IWbemServices::ExecNotificationQuery Async()`
  - WMI Query Language (WQL) similarities with SQL
- Very commonly used from scripting and .NET
  - Visual Basic Script (VBS)
  - Visual Basic .NET
  - etc.

WMI Data Providers and Data Blocks

- WMI data providers supply WMI data blocks, which can represent one or more of the following:
  - Data items which a driver sends to, or receives from, a WMI client
  - Methods (functions) which a driver executes on behalf of a WMI client.
  - Events which a driver sends to WMI clients
- WMI data blocks are instances of classes having attributes and methods
- Data block content depends on application (not defined by WMI)
- Managed Object Format (MOF) for description of classes
- Standard WMI data blocks for specific device classes defined in Wmicore.mof

WMI Class description sample

```
[WMI, Dynamic, Provider("WmiProv"),
Description("Sample class description"),
Locale("MS\\0x409"),
guid("{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx")]
class Vendor_SampleClass1
{
      [key, read]string InstanceName;
    [read] boolean Active;
    [WmiDataId(1)read, write,
     Description("unsigned long data")
    ] uint32 Xuint32;
};
```

WMI Methods

- KMDF drivers support WMI methods by implementing the event callback `EvtWmiInstanceExecuteMethod(...)`

WMI Events

- WMI Events
  - WMI classes, which are derived from `WMIEvent`
  - Typically used for indicating exceptional conditions (not to be used as an alternative for logging errors)
- WMI EventTracing
  - `WdfWmiProviderTracing flag` in `WDF_WMI_PROVIDER_CONFIG::Flags`
  - WPP software tracing supplements and enhances WMI event tracing by adding ways to simplify tracing the operation of the trace provider.

WMI Tools

- WMI Administrative Tools
  - CIM Studio
  - WMI Object Browser
  - WMI Event Registration
  - WMI Event Viewer

- `WMIC` command line tool
- `WBEMTEST`

- `MofComp`
- `WmiMofCk`

Chapter 11

Driver Signing

Driver Signing Overview

- Microsoft signature
  - Quality (defined by WHQL test specification)
  - Identification
  - Integrity
- Third party signature with code signing certificate
  - Identification
  - Integrity
- By default, Microsoft signatures take installation priority over third-party signatures (see “ranking” later this chapter)

Certificates

- Own Software Publisher Certificate (SPC) issued by Certification Authority (CA)
  - Can be obtained from many different authorities (see Microsoft web page for supported CAs)
  - Verisign is the most commonly used one
- Cross-Certificate for the selected CA that issues SPC
  - Needs to be downloaded from the MS website
  - Operating system kernel only has a single trusted Microsoft root authority.
  - Extend the chain of trust to multiple commercial CAs
- Test-Certificate

Driver Installation and Signing

- Unattended driver installation (no user interface)
  - Microsoft signature
  - SPC from trusted publisher
- Driver installation with gray “Security Dialog Box”
  - Third party signature
  - SPC from publisher with unknown trust
- Driver installation of unsigned drivers with red “Warning Dialog Box“ only on 32 Bit operating systems
- No installation
  - Unsigned drivers on 64 bit operating system
  - SPC from untrusted publisher

Signatures

- Signature of driver package
  - .cat file referenced by .inf file of Plug'n'Play driver
  - .cat file contains hash over whole driver package
  - .cat file needs to be signed as a means for signing the whole driver package
- Embedded signature of binaries
  - Needed for boot drivers SERVICE_BOOT_START
  - Verification by .sys file context menu "Properties"

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific
Chapter 11: Driver Signing

Files

- Digital ID from CA (Verisign example below)
  - `.spc (Public Key)`
  - `.pvk (Private Key)`
- For signing the driver
  - .pfx file
  - .cer (cross certificate)
  - Generated by WDK tool pvk2pfx
- Driver Package
  - `.inf file`
  - `.cat file`
  - `.sys`
- Test certificates: .cer

Tools

- Program Files (x86)\Windows Kits\8.0\bin\x64 \x86
- Production
  - Pvk2pfx
  - Signtool
  - inf2cat
- Testing
  - makecert
  - cert2spc

Further information

- MSDN Signing a Driver
  - http://msdn.microsoft.com/en-us/library/windows/hardware/ff554809(v=vs.85).aspx
- MSDN Tools for Signing Drivers
  - http://msdn.microsoft.com/en-us/library/windows/hardware/ff552958(v=vs.85).aspx

Chapter 12

Tools and Utilities

Tools

Gflags.exe (Debugging Tools For Windows)
Pool Monitor (poolmon.exe)
Windows Checked Build
Driver Verifier
Static Driver Verifier
Driver Test Manager (DTM)
Driver Test Framework
Signtool (for digital driver signing)

Framework Verifier

Subkey Parameters\Wdf under a WDF driver‘s Services key

Values

VerifierOn (REG_DWORD)

VerifyOn (REG_DWORD)

DbgBreakOnError (REG_DWORD)

VerifierAllocateFailCount (REG_DWORD)

TrackHandles (REG_MULTI_SZ)

VerboseOn (REG_DWORD)

TraceDelayTime (REG_DWORD)

HKLM\SYSTEM\CCS\Control\Wdf\Kmdf\Diagnostics

DbgPrintOn (REG_DWORD)

User-Mode Driver Framework

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

Introduction

- For both 32- and 64-bit devices
- For protocol-based or serial bus-based devices (portable music players, PDAs, cell phones, webcams)
- Used largely in Windows SideShow-compatible devices (auxiliary displays)
- Used in Windows Portable Devices (WPD) Framework, which supports USB removable storage (USB bulk transfer devices)
- Also for software-only drivers
- Is based on the same conceptual driver programming modes as the WDF-KMDF

Advantages

- Simpler driver environment
- Improved system stability
- Reduced security risks
- Access to the Win32 API
- Debugging with a user-mode debugger
- Object-oriented programming in C++
- comparable performance to kernel mode

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific
WDF-UMDF

Limitations

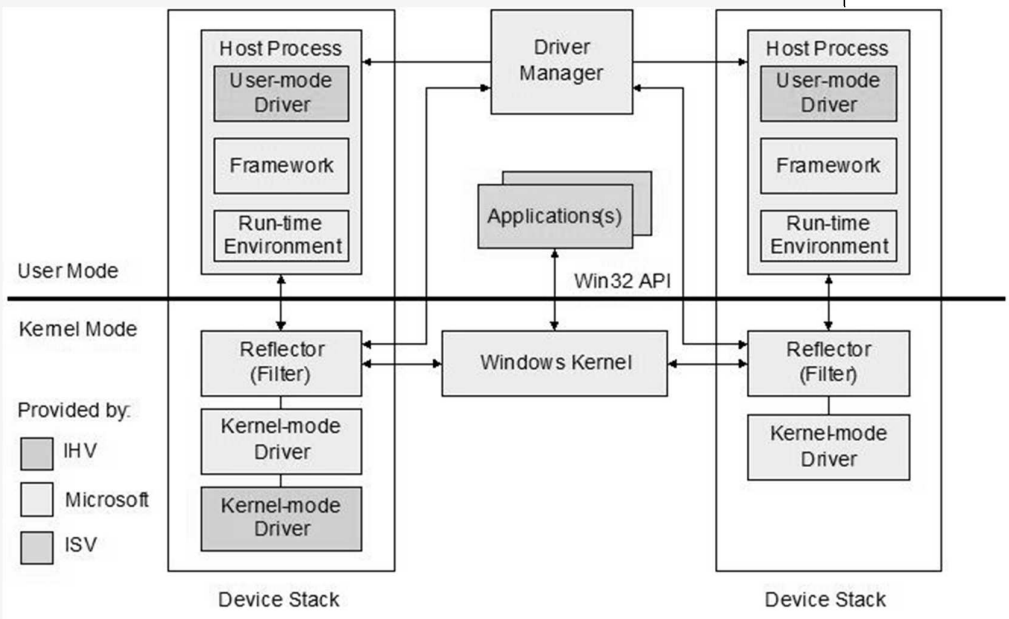
- No internal IOCTL support; only Read, Write, IOCTL
- Only for function and filter drivers, not for bus drivers
- Not for isochronous USB devices

Version: WDK 8 UMDF 1.11

- Integrated into Visual Studio 2012
- For target OS Windows Vista and later
- Supports hardware access and interrupt handling
- Supports *device pooling*, i.e. single instance of Wudfhost to host multiple device stacks (conserves memory consumption)
- Supports HID minidrivers
- Enhanced support for PnP and power management (device's idle capabilities)

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific
WDF-UMDF

Architecture



Key Components

- UMDF drivers
- Driver host process (Wudfhost)
- Reflector (a kernel-mode driver)
- Driver manager (a Windows service)

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

WDF-UMDF

Object Model

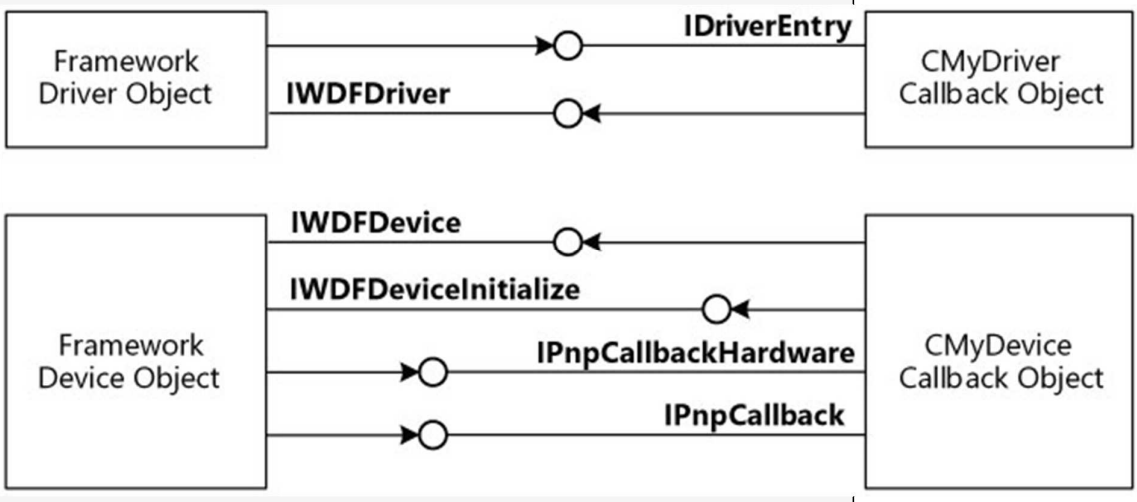
- Framework objects
  - Objects belonging to the framework
  - e.g. driver, device, I/O queues, etc.
- Callback objects
  - Objects implemented in driver and registered to the framework
  - e.g. device-specific responses to events raised by a framework object

Framework Objects

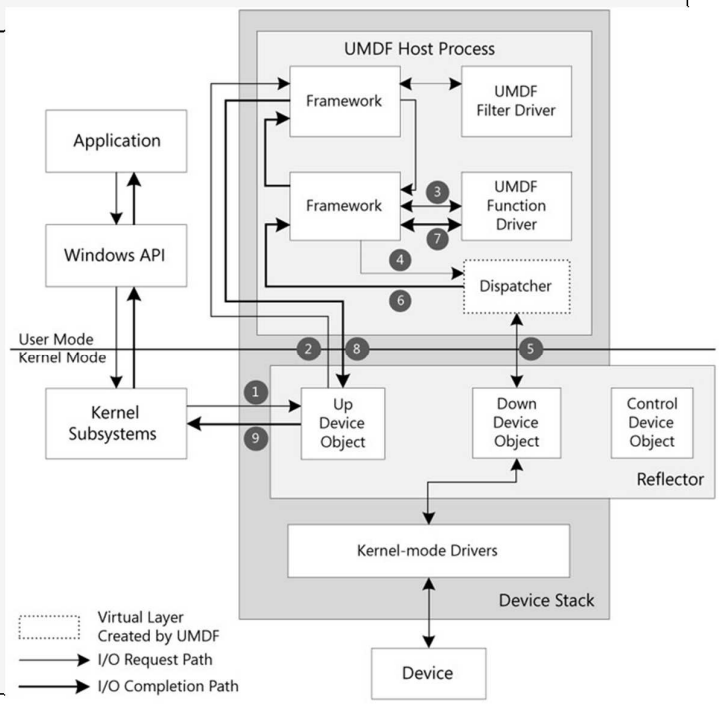
- Driver
- Device
- I/O queue
- I/O request
- I/O target
- Memory
- File
- Property store
- USB device
- USB interface
- USB pipe

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific
WDF-UMDF

Sample: Objects and Callback Interfaces



I/O Request Flow



datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific
WDF-UMDF

Programming: Why COM?

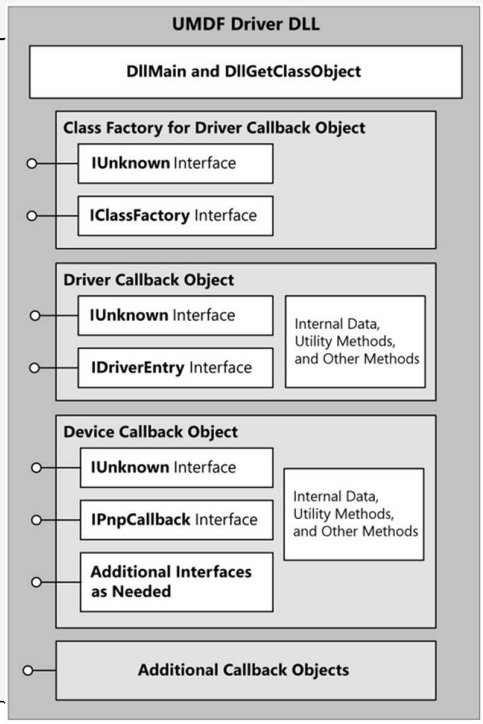
- Component Object Model (COM) is familiar to many applications programmers
- C++ as the preferred language for writing COM applications
- COM interfaces enable logical groupings of functions, so that the DDI is easy to understand and navigate
- Using COM enables the DDI to evolve without requiring existing driver dynamic-link libraries (DLLs) to be recompiled
- Numerous tools, including Microsoft Visual Studio and ATL, support COM-based applications and objects

Key COM Programming Skills

- How to use the COM objects that the UMDF run-time provides
- How to implement the required basic infrastructure to load the UMDF driver
- How to implement a UMDF callback object

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific
WDF-UMDF

UMDF Driver COM DLL



Further information

- Developing Drivers with the Windows Driver Foundation, Orwick and Smith, Microsoft Press, 2007
- MSDN Signing a Driver
  - http://msdn.microsoft.com/en-us/library/windows/hardware/ff560456(v=vs.85).aspx
- MSDN UMDF whitepapers
- WDK 8 UMDF driver samples

Windows Hardware Quality Assurance

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

Windows Certification Program

- Certification as a public statement of confidence from Microsoft that the tested hardware/software works well with the Windows Operating System
- Formerly known as Windows Logo Program (WLP)
- Uses Windows Hardware Certification Kit (HCK) to assess quality of hardware/driver (Kit formerly known as WLK, DTM, HCT)
- Requires certification submissions be done using Windows Hardware Dashboard portal, formerly known as Windows Quality Online Services (Winqual)

Benefits

- Signed device drivers
- Publication of product as **certified** in various catalogs and the compatibility center
- Free distribution channel for driver updates
- Performance telemetry on the product in the field
- Eligibility for various marketing incentives and a logo license

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

Windows Hardware Quality Assurance

Steps: Windows Hardware Certification Program

1. **Set up** a VeriSign and Hardware Dashboard accounts
2. **Design** your product that meets the certification requirements and policies
3. **Test** your product using Windows HCK
4. **Submit** your test package to the Hardware Dashboard Service
5. **Monitor and maintain** your product using the Hardware Dashboard Service, including review error reports and update driver distribution
6. **Stay informed** about changes to the certification program

Important Concepts

- **Feature -** a Windows capability exposed or enhanced by a device or a kernel-mode driver
- **Requirement -** the official written specification that defines what a feature must do to qualify for hardware certification
- **Tests -** validate that features are implemented on a device in accordance with requirements
- **Product Type -** contains a predefined list of testable features

NOTE: To receive a Windows certification, a product must implement all of the features of at least one product type

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

Windows Hardware Quality Assurance

Hardware Certification Requirements (3 Sets)

- System Level
  - Certification requirements for client or server systems (must be composed of Windows certified internal components)
- Device Level
  - Certification requirements for stand-alone devices (internal or external)
- Filter Driver Level
  - Certification requirements for filter drivers:
    - Windows Filtering Platform drivers
    - File Systems filter drivers
    - Antivirus filter drivers
    - Early Launch Anti-Malware (ELAM) drivers

Windows HCK

- For certifying systems or hardware devices for the following Windows Operating Systems
  - Windows 8
  - Windows Server 2012
  - Windows 7
  - Windows Server 2008 R2
- Is based on **feature** detection
  - HCK automatically searches for features on the device, and determines the appropriate tests

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

Windows Hardware Quality Assurance

Windows HCK

- Sample scenario: a multi-function printer device
  - Features:
    - a scanner
    - an Ethernet network port
    - a storage reader
    - a printer
  - HCK detects each feature, determines the associated requirements for it, and then schedules appropriate tests to verify that the requirements are met

Windows HCK: Setup and Testing

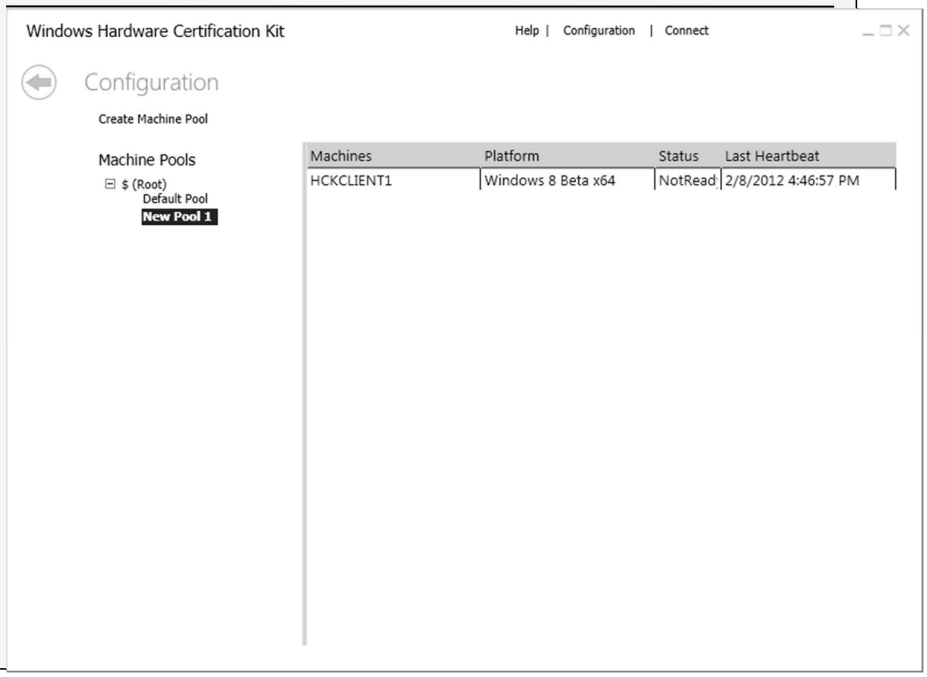
1. Review prerequisites (needs a test server computer, and one or more test client computers)
2. Install HCK Controller and HCK Studio software on the test server
3. Install HCK Client software on test computer(s)
4. On test server's HCK Studio:
   - Create a project
   - Create a machine pool defining one or more test computer(s)
   - Select target (system, device, driver) to certify
   - Select and run tests
   - Create a submission package
   - Submit the signed package for certification

datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

Windows HCK: Screenshots

Windows Hardware Quality Assurance

Windows HCK: Screenshots

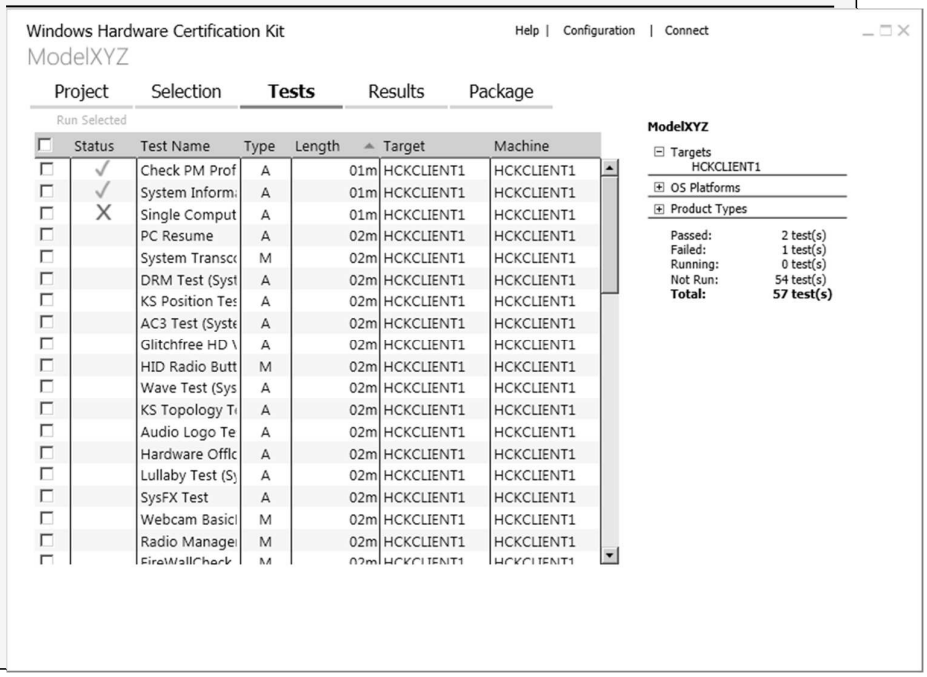


datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

Windows HCK: Screenshots

Windows Hardware Quality Assurance

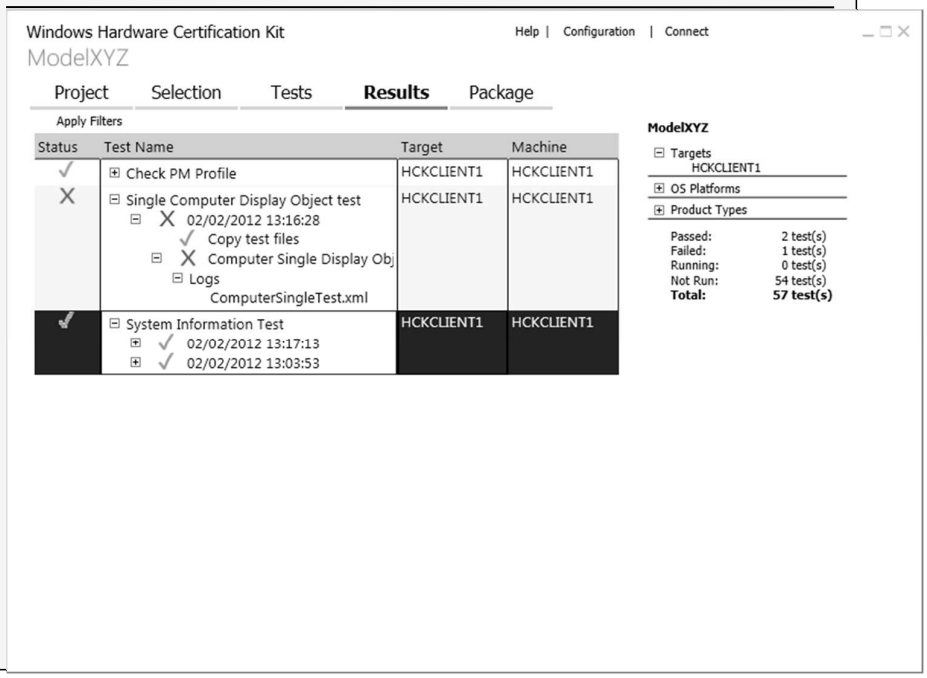
Windows HCK: Screenshots



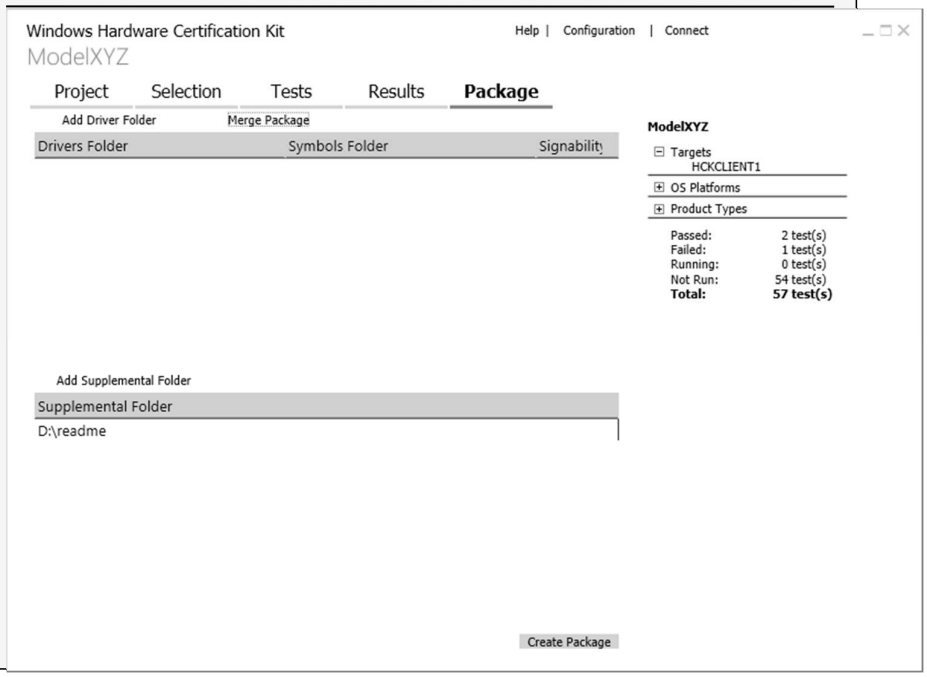
datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

Windows HCK: Screenshots

Windows Hardware Quality Assurance



Windows HCK: Screenshots



datronicsoft
Driver Development Knowledge Center
Europe & Asia / Pacific

Further information

Windows Hardware Quality Assurance

- MSDN Windows Hardware Certification
  - http://msdn.microsoft.com/en-us/library/windows/hardware/gg463010
- MSDN Windows Dev Center – Hardware
  - http://msdn.microsoft.com/en-us/windows/hardware
- MSDN Windows Dev Center - Hardware Dashboard Services
  - http://msdn.microsoft.com/library/windows/hardware/gg463091
- MSDN Forum Windows Hardware Testing and Certification
  - http://social.msdn.microsoft.com/Forums/en-US/whck/threads

datronicsoft

Windows Usb Driver Development with WDF KMDF

Hands On Lab Manual

Table of Contents

1 A Basic Driver Template

Overview

First the WDK development environment has to be set up. It consists of a Host PC (Debugger) running the development environment with debugger and a Target PC (Debuggee) for running and testing the driver. In this lab the Target PC is a virtual machine.

Then a minimum empty skeleton WDF KMDF USB driver is built. The code for this exercise is generated by Visual Studio wizard.

After successful compilation on the Host machine the device driver will be installed on the virtual Target machine.

The kernel debugger WinDbg will be used to remotely debug the target machine running the device driver. WinDbg can operate via serial cable, 1394, USB 2.0 (Vista and above), USB 3.0 Windows 8 or network (Windows 8). In this lab WinDbg operates via network.

Goal

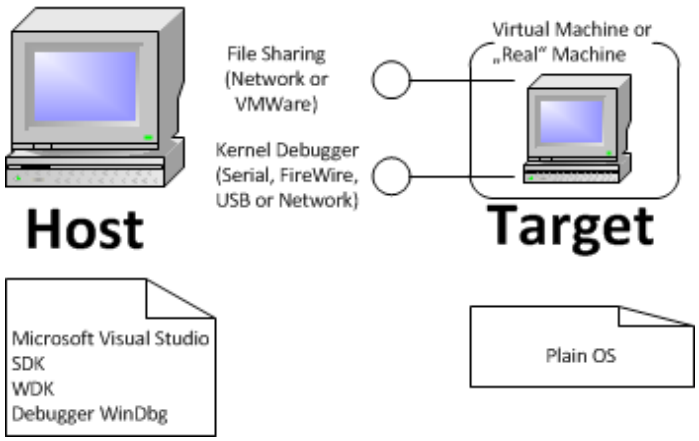
Upon completion of this lab the student will be able to compile, install and debug a minimum WDF KMDF driver template on Microsoft Windows platforms.

Setup

The template is located in the following directory:	-
Own new lab directory to create:	SmplDeviceUsbKmdfLab1
The following files are involved:	
The following files are new:	Driver.c Device.c Driver.h Device.h SmplDeviceUsb.inf

1.1 Setting up the build environment

This chapter makes the students familiar with the WDK development environment in Visual Studio 11.



➔ Step 1:
Installation of the development environment on the Host machine

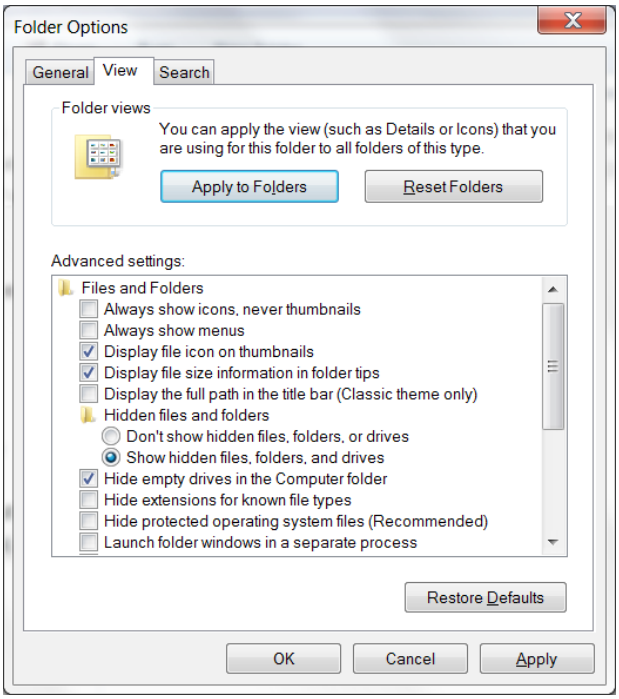
Install the software development tools below on the Host (Debugger) machine in exactly the same sequence:
- Visual Studio 11 (2012)
- Windows 8 SDK
- Windows 8 WDK

➔ Step 2:
Setting up the virtual Target machine environment

1. Install the virtual machine environment on the Host machine:
   - VMWare Player 5.02 or above
2. Install the guest operating system on the virtual Target Machine:
   - Microsoft Windows 8
3. Install “VMWare Tools”. This is necessary for copy-paste from the host to the virtual machine.
4. Add a second boot entry using bcdedit command from an elevated (run as administrator) Command Prompt console. Run bcdedit from a command line windows on the Target “as administrator”. In this command line Window please enter:
   `bcdedit /copy {current} /d “Win8 second boot entry”`

Hint: msconfig.exe can be used to configure the second boot entry as “Safe Boot”

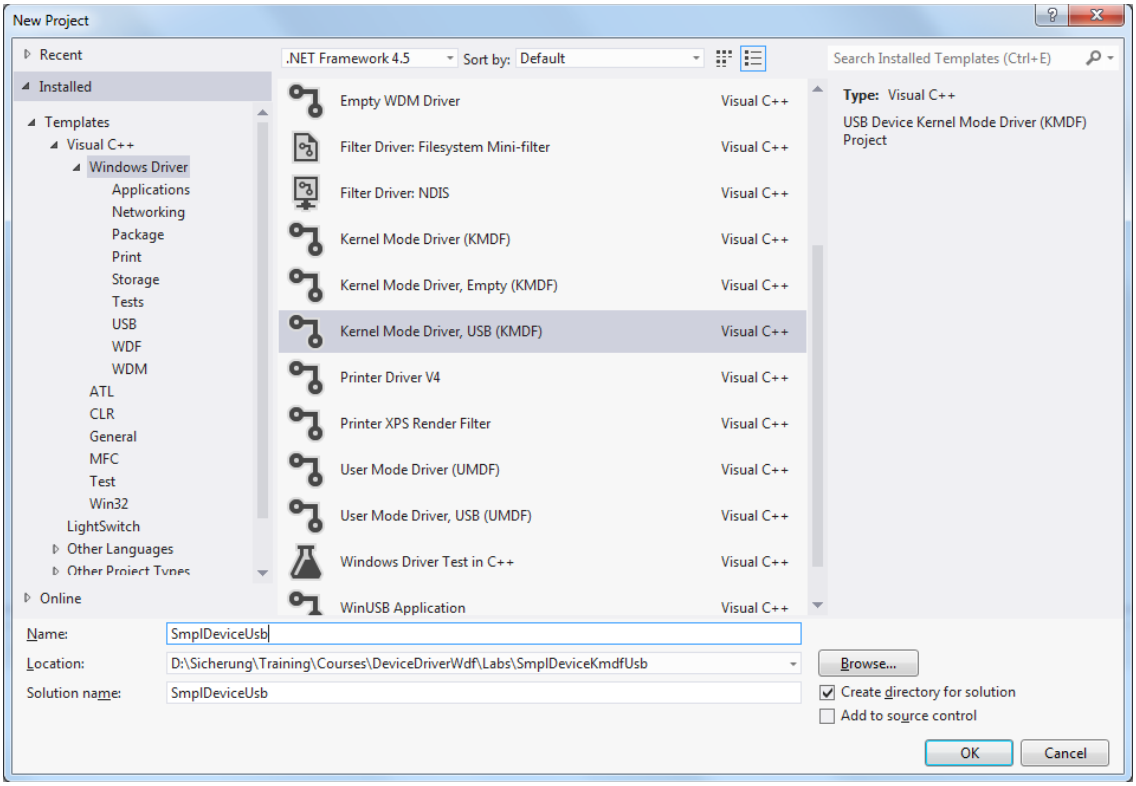
➔ Step 3:
Setting folder options



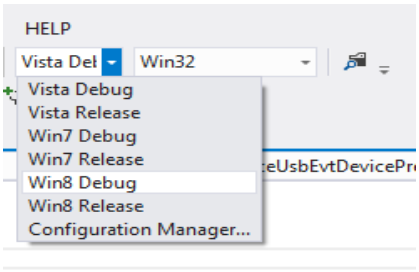
Three folder options should be changed:
- “Show hidden files, folders and drives” (enable)
- “Hide extensions for known file types” (disable)
- “Hide protected operating system files” (disable)

➔ Step 4:
Creating and compiling a driver template on the Host machine

Create a working directory named SmplDeviceKmdf\SmplDeviceKmdfLab1. In this directory, create a new Visual Studio project of type “Windows Driver” selecting “Kernel Mode Driver, USB (KMDF”). The new project shall be named SmplDeviceUsb.



Compile this wizard generated driver template for Win8 Debug:



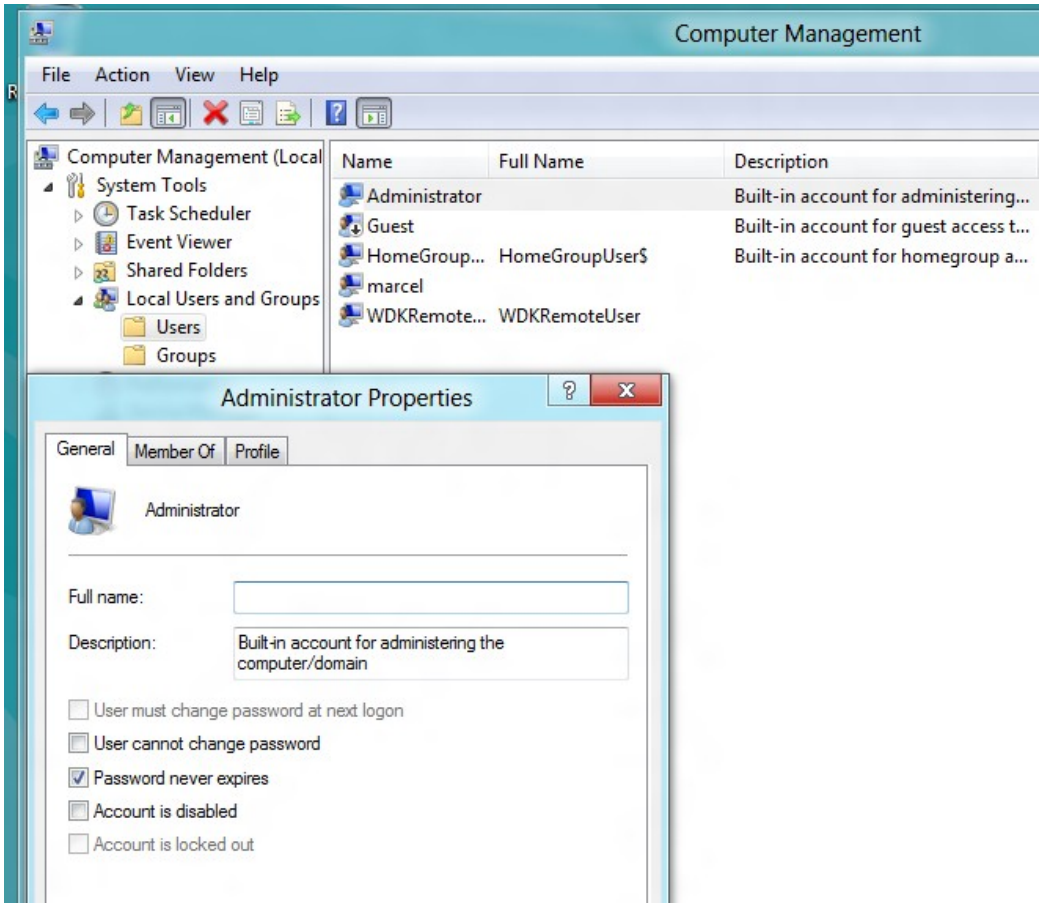
After compiling, the newly built driver executable (.sys file) is available in the appropriate subdirectory.

1.2 Setting up the test environment

This chapter makes the students familiar with the WDK test environment in Visual Studio 11 2012.

➔ **Step 1:**
Prepare the virtual target machine for deployment, debugging and testing

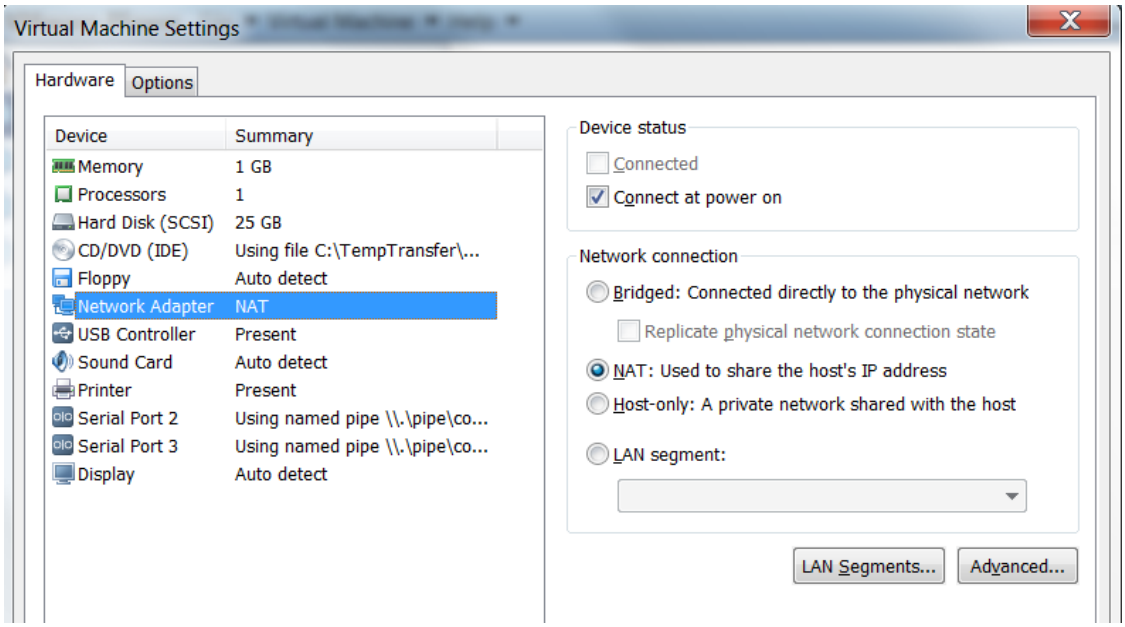
Enable the existing local Administrator account (it is disabled by default) on the virtual Target machine and assign a password to the local administrator account.



As an alternative, the following command can be used:

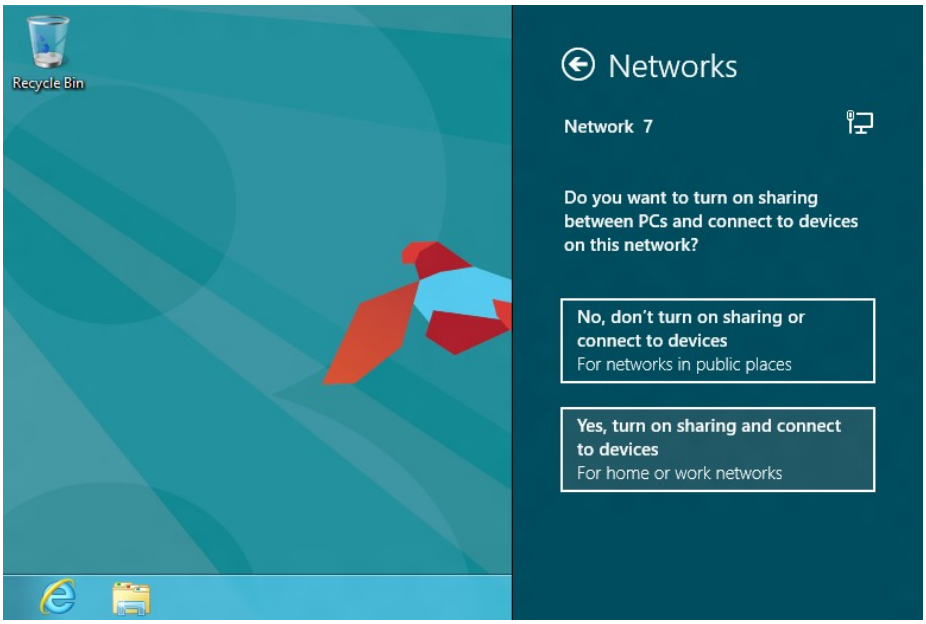
```
net user administrator /active:yes
```

In VMWare “Virtual Machine Settings” set the network adapter to NAT (Network Address Translation) and make sure that a network connection exists between Host and virtual Target machine (using ipconfig and ping commands).



Make sure that a network connection exists between Host and virtual Target machine (using ipconfig and ping commands).

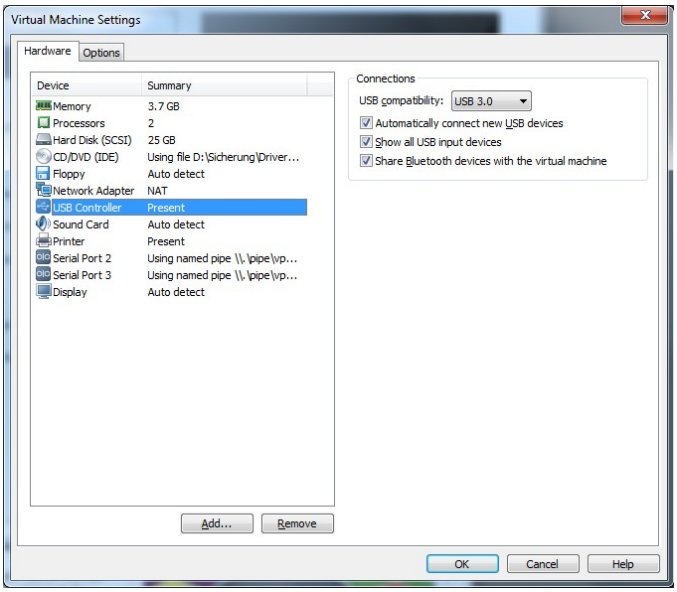
Enable file sharing:

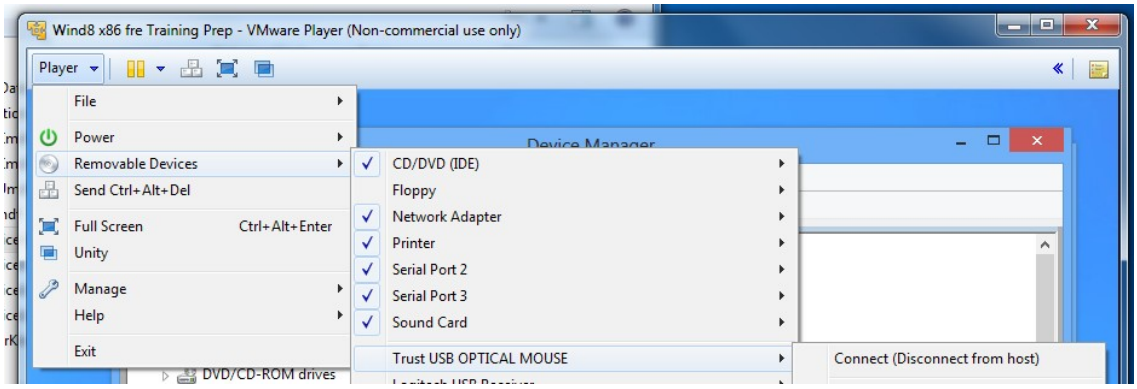


Make sure you can connect to the Target's share “users” from the host. In case the connection to the share is not successful, disconnect all other network connections on the host (e.g. company network, wireless, internet, etc.)

➔ Step 2:
Assigning a USB Controller and USB mouse to the virtual machine

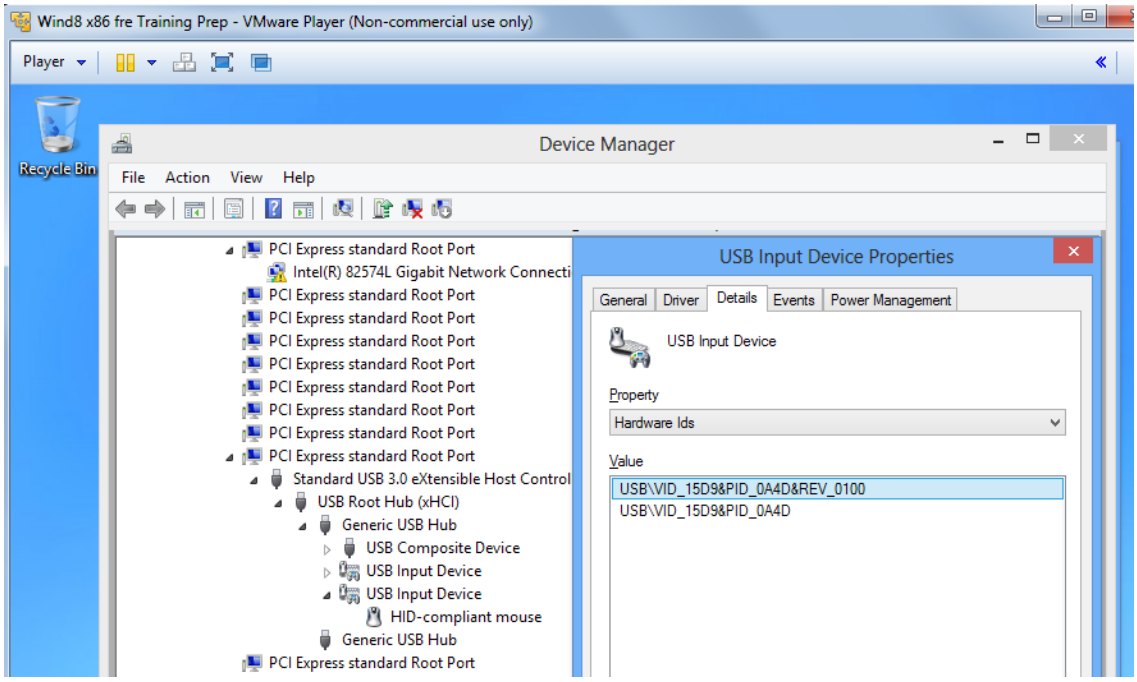
One USB controller with an additional secondary USB mouse has to be dedicated to the virtual machine.





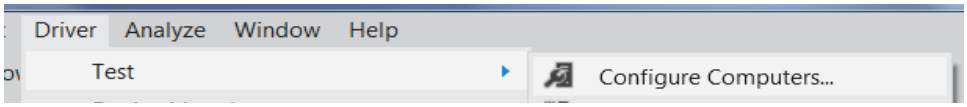
➔ Step 3:
Hardware ID of USB mouse in virtual machine

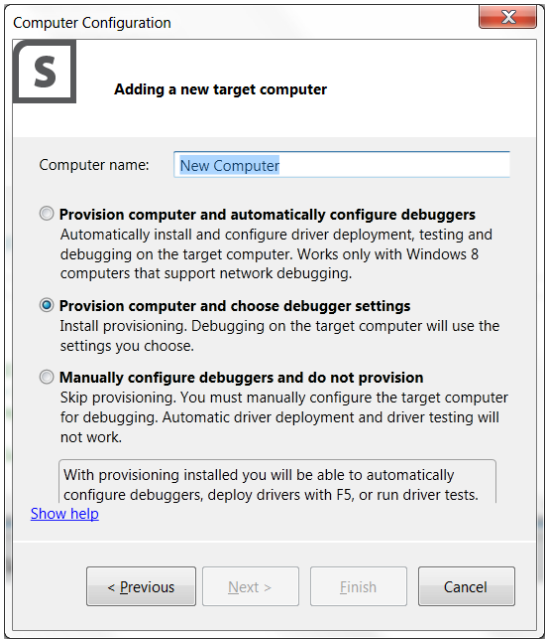
Find the USB mouse inb the virtual machine and determine its harddware ID in the form of `USB\Vid_Xxxx&Pid_Yyyy`.



➔ Step 4:
Install the Visual Studio test environment on the target machine

Add a new test machine in Visual Studio “Driver” menu:





➔ Step 5:
Update the INF file specifying the correct hardware ID

In Solution Explorer, expand Driver Files underneath the SmplDeviceUsb project, and open the INF file. Replace the template-generated dummy hardware ID with the hardware ID of the real

USB device. For example, from:
`%DeviceName%=MyDevice_Install, USB\VID_vvvv&PID_pppp`

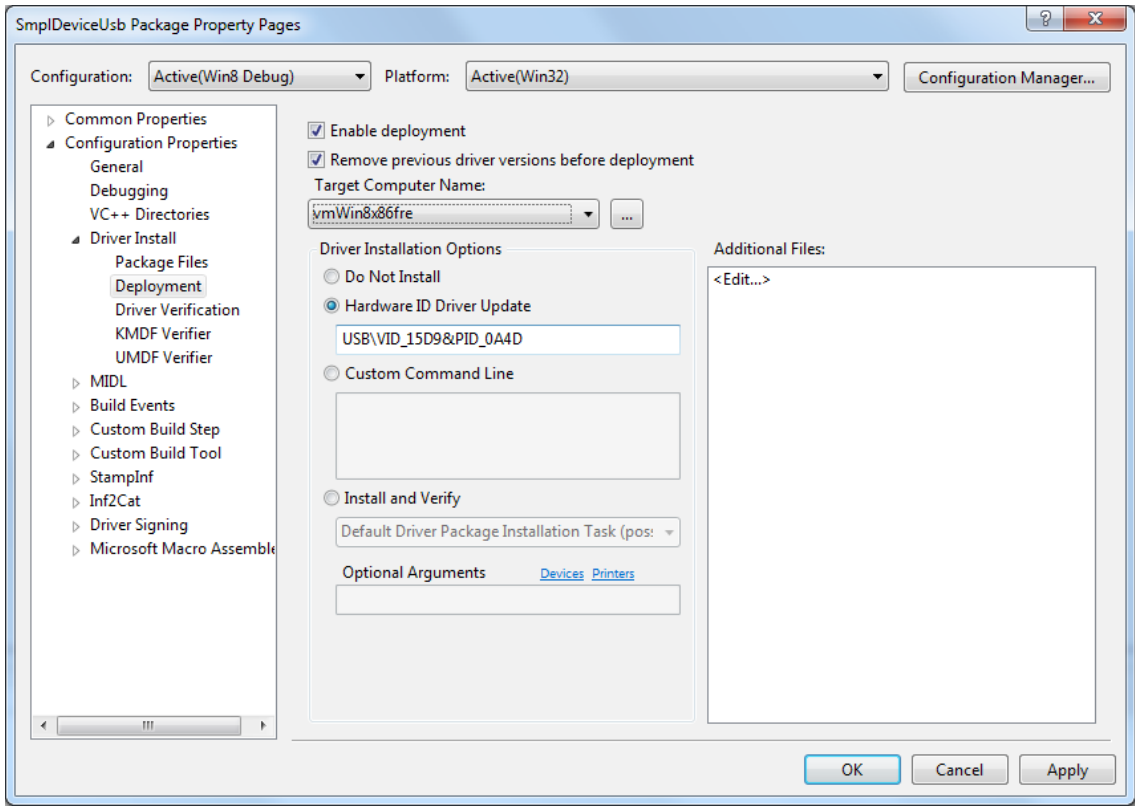
to:
`%DeviceName%=MyDevice_Install, USB\VID_093A&PID_2510`
(This is only an example! The real hardware ID must be substituted instead!)

Uncomment the line setting `FriendlyName` property in the registry:

```
[SmplDeviceUsb_AddReg]
...
HKR,,FriendlyName,,%SmplDeviceUsb.DeviceDesc%
```

➔ Step 6: Configure and run automatic driver deployment to the virtual Target machine

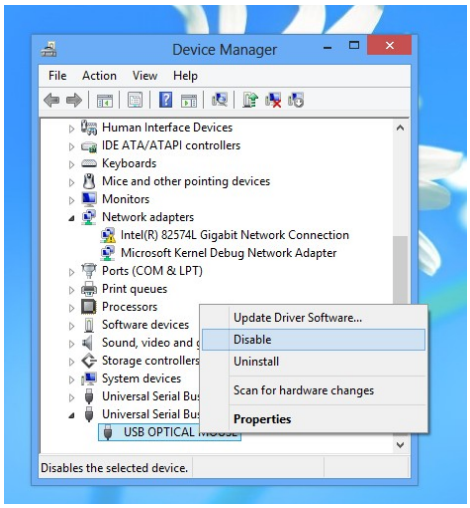
In the SmplDeviceUsb Package project properties, configure deployment to the virtual Target machine.



When rebuilding the SmplDeviceUsb Package project, the driver is automatically deployed to the vitual Target machine.

➔ **Step 7:**
Inspecting, enabling and disabling driver operation using Device Manager:

The driver is already fully Plug'n'Play capable. It can be inspected, activated and deactivated using Device Manager.



1.3 First inspection of source code and object model

This chapter makes the students familiar with the KMDF wizard generated code skeleton and the operating system driver and device object model.

➔ **Step 1:**
Inspection of the wizard generated function driver skeleton

The wizard generated minimal function driver template needs to be inspected. It has no functionality yet. It is neither interacting with an application nor with any hardware.

For support of static code analysis, the following function prototypes are defined in the header file:

```
DRIVER_INITIALIZE DriverEntry;
EVT_WDF_DRIVER_DEVICE_ADD SmplDeviceEvtDriverDeviceAdd;
```

The `DriverEntry` routine is the only function which is explicitly called by exported name. During `DriverEntry` the KMDF driver has to create the `WDFDRIVER` object

During the `WDFDRIVER` object's event handler `EvtDriverDeviceAdd` The the driver has to create a `WDFDEVICE` object.

Hint:	Always consult the WDK documentation (Help) and look up data structures and functions. These can then be copied and pasted directly from the WDK documentation to the driver source code.

➔ **Step 2:**
Inspecting the driver object and the device object

The tool WinObj.exe (download from www.sysinternals.com) can be used to look at the driver object and the device object on the virtual Target machine.

1.4 Setup of the WinDbg debugger environment

During this lab, the debugger will use a Local Area Network connection (only available in Windows 8). The environment of the debugger WinDbg will be set up on the Host machine. On the virtual Target machine kernel debugging has to be enabled using bcdedit to allow the Target to be controlled by the external debugger running on the Host machine.

➔ Step 1:
Inspection of the debugger settings on the target

Run bcdedit from a command line windows on the Target “as administrator”. In this command line Window please enter:

```
bcdedit /dbgsettings
```

If network debugging was already done by Visual Studio in the previous exercise, then nothing needs to be done any more.

➔ Step 2:
Enable kernel mode debugging on the virtual Target machine

ONLY if Visual Studio provisioning of the test machine was NOT successful, the following commands need to be run to enable kernel debugging over network on the virtual Target machine.

Run bcdedit from a command line windows on the Target “as administrator”. In this command line Window please enter:

```
bcdedit /dbgsettings NET hostip:xxx.xxx.xxx.xxx port:yyyyy
bcdedit /debug on
```

If this is successful, then a key is printed on the console. Please copy paste this key into a file. It will subsequently be needed by the debugger on the host machine to connect (see step below).

Target needs to be shut down and started again (caution: reboot might not be enough). After that, the kernel debugger will be active and WinDbg can connect.

➔ Step 3:
Setup of the debugger WinDbg on the Host machine

After starting WinDbg, the following settings have to be chosen:
- File menu → Kernel Debug → NET IP port number and key
- File menu → Symbol File Path
- File menu → Source File Path

The symbol file path to own driver symbols is required for debugging. It needs to point to the .pdb (Program Data Base) file generated when the driver was compiled.

The symbol file path to the Windows OS symbols is optional. The following line can be added (separated by semicolon) to WinDbg Symbol File Path:

```
SRV*YourLocalOsSymbolCacheFolder*http://msdl.microsoft.com/download/symbols
```

Hint:	For further info about WinDbg commands, consult the Appendix of this manual (e.g.: WinDbg useful commands).

➔ Step 4:
Adjustment of “Debug Print Filter” registry setting

Open or create the following registry key on the virtual Target machine:
`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter`.

Add a registry value of type `REG_DWORD` named IHVDRIVER. Set its value to 0xFFFFFFFF.

The new Debug Print Filter setting will become active after a reboot

➔ Step 5:
Instrumentation of source code using DbgPrintEx(…)

Use the statement `DbgPrintEx(DPFLTR_IHVDRIVER_ID, 1234,"…", …)` to trace the call of `DriverEntry ()` and `SmplDeviceEvtDriverDeviceAdd()`. The driver just has to be disabled and enabled in Device Manager while the debugger is connected. Then this debug output shows up on the debugger output screen.

Hint: WinDbg and the tool DbgView.exe (download from www.sysinternals.com) can both be used to visualize `DbgPrintEx(...)` traces on the virtual Target machine..

➔ Step 6: Instantiation of breakpoints

The debugger needs the correct driver symbols loaded (Windows operating system symbols are not needed at this stage).

The option „Symbol File Path has to point to the directory where the driver symbols are located. They reside in a separate .pdb file (program database). WinDbg loads symbols at driver startup. If connected later, symbols can be loaded using the command ".reload".

When the driver is started and the symbols are loaded, the source file Driver.c can be opened and a breakpoint can be set in the function `SmplDeviceEvtDriverContextCleanup(...)` using the F9 key . If successful, then the breakpoint is shown in red color.

Disable the device in Device Manager, then `SmplDeviceEvtDriverContextCleanup(...)` will be called and the breakpoint will be triggered. Single stepping in the debugger can be done using the F10 key.

2 Driver and Application Interactions

Overview

This lab teaches the basics about interaction between device drivers and user mode applications.

The driver exposes a Plug'n'Play style device interface to make the device visible and accessible from the user mode application. Applications use this device interface which is defined by a unique GUID for obtaining a symbolic link to the associated device object. This symbolic link is used by Win32 applications to get a handle for accessing the device. This handle can subsequently be used by the application to interact with the driver.

This lab later shows how this handle is being used for reading data from the driver via the Win32 API function `ReadFile(...)`.

Goal

Implementation of a Windows device driver, which is able to transmit data to a user mode application via the Win32 system call `ReadFile(...)`.

Setup Device Driver

The template is located in the following directory:	SmplDeviceUsbKmdfLab1
Own new lab directory to create:	SmplDeviceUsbKmdfLab2
The following files are involved:	Driver.c Device.c Driver.h Device.h SmplDeviceUsb.inf
The following files are new:	Queue.c Queue.h Public.h

Setup Application

Own new lab directory to create:	SmplApplicationConsole
The following files are involved:	
The following files are new:	SmplApplicationConsole.cpp

2.1 User mode application accessing the device driver

Applied Setup API functions in the user mode console application:

```
SetupDiGetClassDevs(…)
SetupDiEnumDeviceInterfaces(…)
SetupDiGetDeviceInterfaceDetail(…)
SetupDiDestroyDeviceInfoList(…)
```

Applied Win32 API functions in the user mode console application:

```
CreateFile(…)
ReadFile(…)
Close Handle(…)
GetLastError(…)
```

➔ Step 1: Creation of a user mode test application

Microsoft Visual Studio shall be used to create a new project type “Win32 Console Application”. Its name shall be it "SmplApplicationConsole". The following headers have to be included “setupapi.h", "objbase.h" and "initguid.h". The static library "setupapi.lib" has to be added to the linker input.

➔ Step 2: Enumerating the device interface via Setup API

Setup API must be used to write a function, which accesses the device interface via GUID to retrieve the corresponding symbolic link name. This function should be prototyped as follows:

```
bool SetupApiDeviceInterfaceInstanceEnumerate(GUID* pGuid, DWORD
instance, string& SymbolicLinkName);
```

At first this function has to retrieve a handle for the relevant class information. This is done via

`SetupDiGetClassDevs(...)`

After this it has to enumerate the interfaces belonging to this class via

`SetupDiEnumDeviceInterfaces(...)`

Then the function `SetupDiGetDeviceInterfaceDetail(...)` has to be called first to get the length of the Symbolic Links and then a second time to get the symbolic link itself. The Symbolic Link shall be printed on the console for information purposes.

Finally the allocated resources have to be released again using `SetupDiDestroyDeviceInfoList(…)`

Hint:	A complete error check should always be implemented on every Win32 API call. This includes a check on the return value as well as displaying error information in case of any error. In most cases this is done by the Win32 API function `GetLastError(...)`.

➔ Step 3: Reading data from the device using ReadFile Windows API

A function `DeviceRead(wstring SymbolicLinkName)` shall be created.

It shall take the previously enumerated symbolic link name as a parameter. The device shall be opened using `CreateFile(…)` and a loop shall read data from the device using `ReadFile(…)` 50 times. Before exiting the application the device has to be closed again using the Win32 API function `CloseHandle(…)`.

2.2 Plug'n'Play device interface exposed by driver

A Plug'n'Play style device interface exposes a Device Object to user mode applications.

Applied KMDF methods:

`WdfDeviceCreateDeviceInterface(…)`

➔ Step 1:
Definition of a new GUID for the Plug'n'Play style device interface

The template contains a separate header file named “Public.h”. The application include the header file with this GUID for accessing the driver.

Replace the wizard generated GUID by the one defined below:

```
// {D0C941C5-51B0-4d90-8278-FB17DF05FD65}
DEFINE_GUID(GUID_DEVINTERFACE_SMPLDEVICE,
0xd0c941c5, 0x51b0, 0x4d90, 0x82, 0x78, 0xfb, 0x17, 0xdf, 0x5, 0xfd,
0x65);
```

➔ Step 2:
Device interface creation (inspection of wizard generated code)

Inspect the wizard generated code. The function `WdfDeviceCreateDeviceInterface(...)` registers a device interface. The framework automatically enables all the device’s interfaces when the device is available and disables the device’s interfaces when the device is removed or disabled.

Hint:	After compiling the driver, it just needs to be copied to the target's \Windows\System32\Drivers directory. Subsequent disable and enable in Device Manager loads the new driver. It is not necessary to re-install the driver after each re-compile.

➔ Step 3:
Testing the functionality of the device interface

Run the Console application and test if it can enumerate the device interface

2.3 Simple driver handling of WDF Request for I/O

This lab introduces to the basics of handling WDF Request for I/O. The code template shall be changed to implement a specific IOQueue for I/O Requests of type `WdfRequestTypeRead` processing `ReadFile(...)` user mode application I/O Requests

Applied KMDF methods:

`WdfIoQueueCreate(…)`
`WdfRequestCompleteWithInformation`

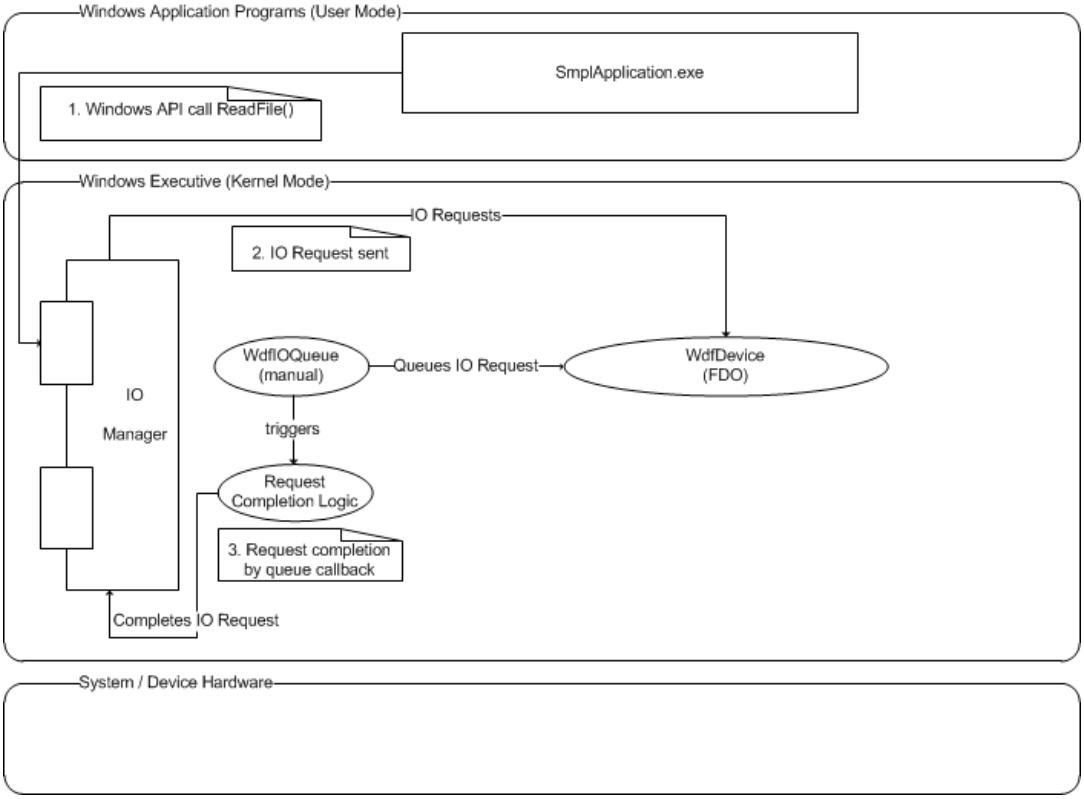
Applied KMDF event callbacks:

`EVT_WDF_IO_QUEUE_IO_READ`

Applied KMDF data structures:

`WDF_IO_QUEUE_CONFIG`

Overview diagram:



➔ Step 1: Implementation of an EvtIoRead event callback function

The `EvtIoRead` callback function is called when the framework receives a `ReadFile(…)` I/O Request from an application. The callback function below shall be declared in Queue.h and implemented in Queue.c. All it is supposed to do is completing the I/O Request and indicate zero bytes transferred.

```
EVT_WDF_IO_QUEUE_IO_READ SmplQueueEvtIoRead;

VOID SmplQueueEvtIoRead(
   __in WDFQUEUE Queue,
   __in WDFREQUEST Request,
   __in size_t Length
)
{
    DbgPrintEx( DPFLTR_IHVDRIVER_ID,
                1234,
                "SmplQueueEvtIoRead\n");
    WdfRequestCompleteWithInformation(Request, STATUS_SUCCESS, 0);

}
```

Hint:	The compiler is set to “warning level 4” and “treat warnings as an error” by default. Besides the useful results, this also generates a bit too much noise which can be disabled in the header file driver.h as follows: `#pragma warning (disable:4100) // unreferenced formal parameter`

➔ Step 2: Subscribing for read events of the default IoQueue

The wizard has generated a default queue for handling all possible kinds of I/O Requests. The wizard has only created a queue callback for `EvtIoDeviceControl`. Its implementation `SmplDeviceEvtIoDeviceControl` not needed and shall be removed.

To subscribe for queue events of type `WdfRequestTypeRead`, the previously implemented callback event handler function `SmplQueueEvtIoRead(...)` has to be assigned to the `EvtIoRead` member of the previously initialized `WDF_IO_QUEUE_CONFIG` data structure.

➔ Step 3: Testing the functionality

Please be sure to test the read dispatch method by calling the Win32 API function `ReadFile(…)`. At this stage the parameter indicating the byte count to read still should be zero. The driver still cannot transfer any data.

Use the debugger WinDbg (traces or breakpoint) to make sure that the appropriate EvtIoRead event callback function is invoked, whenever the corresponding Win32 API functions are called by the test application.

2.4 Transferring data in the queue's `EvtIoRead` callback

Applied KMDF methods:

`WdfRequestRetrieveOutputMemory(…)`
`WdfMemoryCopyFromBuffer(…)`

Applied KMDF data structures:

`WDFMEMORY`

➔ Step 1: Accessing the I/O Request's memory and perform read operation

To retrieve the output memory associated with the Request the method `WdfRequestRetrieveOutputMemory(...)` has to be used. It retrieves the output buffer for I/O requests in the same manner no matter if Buffered I/O or Direct I/O is used. If no method has been specified WDF uses Buffered I/O as default.

Hint: Always implement a complete error check on every single call which has a return status! This includes a check on the return value as well as displaying error information in case of any error. Example:

```
if(!NT_SUCCESS(Status))
{
    DbgPrintEx(DPFLTR_IHVDRIVER_ID,
               1234,
               "WdfRequestRetrieveOutputMemory failed! Status:%x\n",
               Status);
}
```

➔ Step 2: Copying data to the I/O Request's memory

The `WdfMemoryCopyFromBuffer(...)` safely copies the data from the buffer pointer to the destination WDFMEMORY handle. Each WDFMEMORY handle contains the size of the buffer. This function can verify that there will be no buffer overruns resulting from the copy operation.

In this lab the string "Hello World!" shall be copied to the buffer. Caller's buffer size in the `Length` parameter of the `EvtIoRead` callback shall be checked. If the application's buffer is not large enough the driver should return `STATUS_BUFFER_TOO_SMALL`.

➔ Step 3: Indicating the size of the data to the user mode application

`WdfRequestCompleteWithInformation(...)` method shall now indicate the number of bytes it is returning by the information parameter

➔ Step 4:
Modification of the test application and testing the functionality

In the user mode application a buffer has to be declared for taking the data delivered by the driver. Then the driver `EvtIoRead` callback function can be tested by calling the Win32 API function `ReadFile(…)`.

➔ Step 5:
Changeover to Direct I/O

During device initialization used `WdfDeviceInitSetIoType(…)` do change over to Direct I/O by selecting `WdfDeviceIoDirect`. Call this function prior to `WdfDeviceCreate(…)`.

3 Filtering Requests of the Keyboard Driver

Overview

This lab teaches the fundamentals of Windows driver layering.

A filter driver is implemented which is located above the operating system supported Keyboard Class driver (Keyboard Class driver is an “Upper Filter” of every Keyboard Port driver).

This sample has been chosen because for each keystroke is a directly related to an IRP being sent from the Windows Subsystem to the Keyboard Class driver.

Info: This keyboard filter above system supplied Keyboard Class Driver is for training and demonstration purposes only! Most real keyboard filter scenarios, would use a filter driver below the system supplied Keyboard Class Driver. Such a realistic filter driver would use `IOCTL_INTERNAL_KEYBOARD_CONNECT` to initialize a driver to driver callback interface. After that, keystrokes are transmitted via this callback interface (not via IRPs).

Goal

Writing a Keyboard Filter driver which can trace and/or modified each keystroke.

Setup

The template is located in the following directory:	-
Own new lab directory to create:	SmplFilterKmdf\
The following files are involved:	Driver.c Device.c Driver.h Device.h SmplDevice.inf
The following files are new:	Queue.c Queue.h

3.1 A KMDF filter driver skeleton

A minimum KMDF filter driver needs to be implemented. It has no functionality yet. It is very similar to the minimum function driver skeleton discussed earlier.

Applied KMDF methods:

`WdfFdoInitSetFilter(...)`

Applied KMDF data structures:

`WDFDEVICE_INIT`

➔ Step 1: Minimum wizard generated driver

Create a new KMDF driver template project again by the Visual Studio wizard. This time name it SmplFilter.

Remove public.h and all device interface related functionality. Temporarily disable `WDFIOQUEUE` object creation by commenting the call to `SmplFilterQueueInitialize(...)` in Device.c.

➔ Step 2:
Automatic Request forwarding to next lower device

`WdfFdoInitSetFilter(...)` enables automatic Request forwarding to the next lower device object for all Request types that are not assigned to a `WDFQUEUE`.
`WdfFdoInitSetFilter(...)` needs to be called BEFORE the `WDFDEVICE` object is created.

➔ Step 3:
INF file for setup as Class Upper Filter in the registry

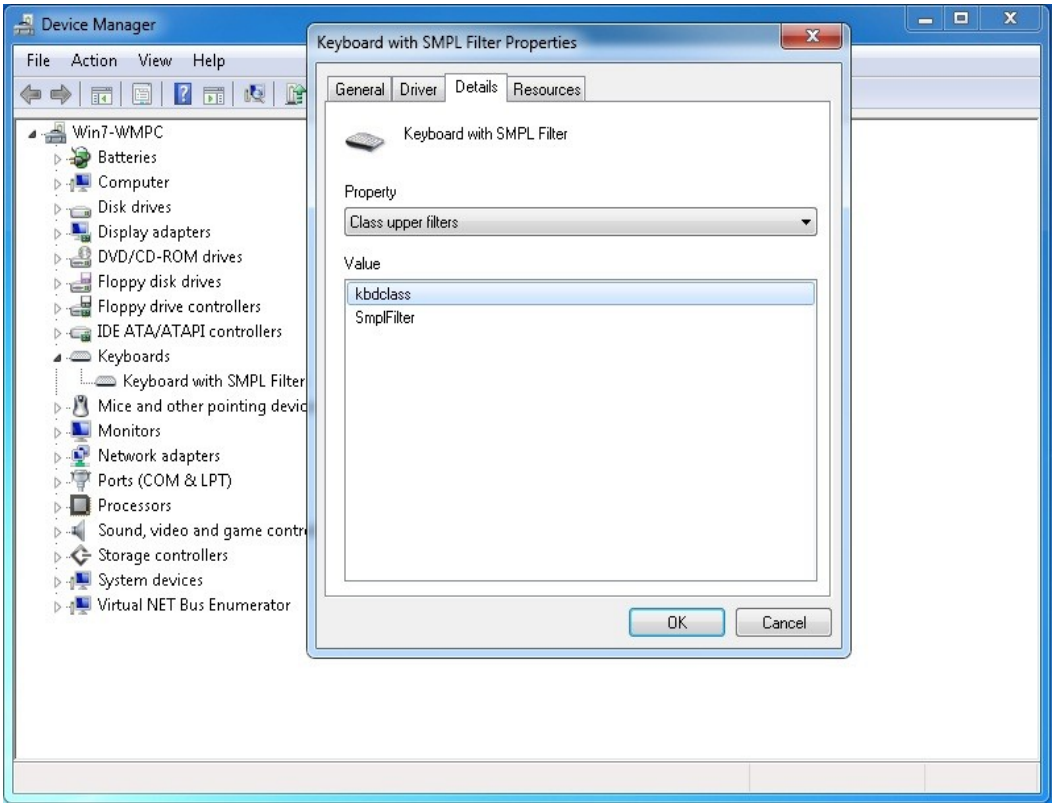
```
; SmplFilter.INF
; Copyright © 2013 datronicsoft/Marcel Rüdinger;
;   This file is part of the training session for
;   developing WDF device drivers. It is used to
;   install the WDF version of the SmplFilter device driver.

[Version]
Signature="$Windows NT$"
Provider=%DAT%
ClassGUID={4D36E96B-E325-11CE-BFC1-08002BE10318}
Class=Keyboard
DriverVer=01/28/2010,4.0.0.0

[DestinationDirs]
DefaultDestDir   = 12

;
; File sections
;
[SmplFilter.CopyFiles]
SmplFilter.sys

;
; Driver information
;

[Manufacturer]
%DAT%   = DAT.Mfg,ntamd64

; For XP 32 and above
[DAT.Mfg]
;%DAT.DeviceDesc11% = SmplFilter, ACPI\PNP0303
%DAT.DeviceDesc21% = SmplFilter, *PNP0303

; For XP 64 and above
[DAT.Mfg.ntamd64]
;%DAT.DeviceDesc11% = SmplFilter, ACPI\PNP0303
%DAT.DeviceDesc21% = SmplFilter, *PNP0303

;
; General installation section
;
[SmplFilter.NT]
; run the directives need by the port driver
Include=keyboard.inf
Needs=STANDARD_Inst

; Copy the driver over
CopyFiles=SmplFilter.CopyFiles

;
; Service Installation
;
[SmplFilter.NT.Services]
AddService = SmplFilter,,SmplFilter_Service_Inst
; Install the generic keyboard driver from keyboard.inf
Include=keyboard.inf
Needs=STANDARD_Inst.Services

[SmplFilter_Service_Inst]
DisplayName    = %SmplFilter.SvcDesc%
ServiceType    = 1                  ; SERVICE_KERNEL_DRIVER
StartType      = 3                  ; SERVICE_DEMAND_START
ErrorControl   = 0                  ; SERVICE_ERROR_IGNORE
;LoadOrderGroup = Pointer Port
ServiceBinary  = %12%\SmplFilter.sys
```

```
[SmplFilter.NT.HW]
; Add the device upper filter
AddReg = SmplFilter.HW.AddReg

; run the directives need by the port driver
Include=keyboard.inf
;Needs=HID_Inst.NT.Services

[SmplFilter.HW.AddReg]
HKLM,"SYSTEM\CurrentControlSet\Control\Class\{4D36E96B-E325-11CE-BFC1-
08002BE10318}","UpperFilters",0x00010008,"SmplFilter"

;
; Source file information
;

[SourceDisksNames]
1 = %DiskId1%,,,

[SourceDisksFiles]
SmplFilter.sys = 1,,

;
;--- SmplFilter Coinstaller installation ------
;

[DestinationDirs]
SmplFilter_CoInstaller_CopyFiles = 11

[SmplFilter.NT.CoInstallers]
AddReg=SmplFilter_CoInstaller_AddReg
CopyFiles=SmplFilter_CoInstaller_CopyFiles

[SmplFilter_CoInstaller_AddReg]
HKR,,CoInstallers32,0x00010000, "wdfcoinstaller01009.dll,WdfCoInstaller"

[SmplFilter_CoInstaller_CopyFiles]
wdfcoinstaller01009.dll

[SourceDisksFiles]
wdfcoinstaller01009.dll=1 ; make sure the number matches with SourceDisksNames

;
;--- SmplFilter WDF sections ------
;
[SmplFilter.NT.Wdf]
KmdfService = SmplFilter, SmplFilter_wdfsect

[SmplFilter_wdfsect]
KmdfLibraryVersion = 1.9

[Strings]
;
; Non-Localizable Strings
;
REG_SZ         = 0x00000000
REG_MULTI_SZ   = 0x00010000
REG_EXPAND_SZ  = 0x00020000
REG_BINARY     = 0x00000001
REG_DWORD      = 0x00010001
SERVICEROOT    = "System\CurrentControlSet\Services"

DAT = "datronicsoft"
DiskId1 = "datronicsoft Installation Disk #1 (Mouse)"
SmplFilter.SvcDesc="Training Course Keyboard Filter Driver"

;
; Localizable Strings
;
DAT.DeviceDesc11 = "Keyboard with SMPL Filter"
DAT.DeviceDesc21 = "Keyboard with SMPL Filter"
```

➔ Step 4:
Testing the functionality

Caution: This step might completely destroy all keyboard functionality. Backup the whole Virtual PC before proceeding!!!

In Device Manager, look for the PS/2 keyboard device and update driver software. To verify if the installation was successful, check the properties of the device and look for "Class Upper Filters“(See screenshot below):



Open Notepad and check if the keyboard is still working.

3.2 *WdfIoQueue for intercepting and forwarding Requests*

This lab section uses a WdfIoQueue queue for receiving all Requests that were initially targeted to the next lower driver. A default queue for all Request types shall be used.

Applied KMDF methods:

```
WdfIoQueueCreate(...)
WdfRequestSend(...)
```

Applied KMDF event callbacks:

```
EVT_WDF_IO_QUEUE_IO_DEFAULT
```

Applied KMDF data structures:

```
WDF_IO_QUEUE_CONFIG
WDF_REQUEST_SEND_OPTIONS
```

➔ Step 1:
Default queue implementation

Use the wizard generated default queue implementation (uncomment queue creation which was commented in the previous lab).

➔ Step 2:
Implementation of an EvtIoDefault event callback function

Remove the wizard generated handler `EvtIoDeviceControl` and its implementation `SmplDeviceEvtIoDeviceControl`. They are not needed any more.

A new handler for `EvtIoDefault` shall be implemented. This callback is invoked when the framework receives any Request from an application. The following callback function shall be declared in Queue.h and implemented in Queue.c:

```
EVT_WDF_IO_QUEUE_IO_DEFAULT SmplFilterEvtIoDefault;
```

All it is supposed to do is forwarding the Request to the next lower driver using

```
WDF_REQUEST_SEND_OPTIONS_INIT(&Options,
                              WDF_REQUEST_SEND_OPTION_SEND_AND_FORGET);

bResult = WdfRequestSend(Request,
                      WdfDeviceGetIoTarget(WdfIoQueueGetDevice(Queue)),
                      &Options);
```

To verify that the callback function gets invoked the following code has to be used:

```
DbgPrintEx(DPFLTR_IHVDRIVER_ID, 1234, "SmplFilterEvtIoDefault\n");
```

➔ Step 3:
Testing the functionality

Use the debugger to make sure that the appropriate `EvtIoDefault` event callback function is invoked, whenever a keyboard key is pressed and released. This can be achieved by using debug trace output via `DbgPrintEx(...)`.

3.3 Completion routine for WdfRequestTypeRead

The driver has to subscribe for a completion routine to receive `WDFREQUEST`s of type `WdfRequestTypeRead` after the lower driver has completed them.

Applied KMDF methods:

```
WdfRequestSetCompletionRoutine(...)
WdfRequestFormatRequestUsingCurrentType(...)
WdfRequestComplete(...)
WdfRequestGetParameters(...)
```

Applied KMDF event callbacks:

```
EVT_WDF_REQUEST_COMPLETION_ROUTINE
```

Applied framework data structures in the driver:

```
WDF_REQUEST_PARAMETERS
```

➔ Step 1:
Definition of a completion routine for WdfRequestTypeRead

A Request completion routine needs to be declared and implemented. It has the following prototype:

```
EVT_WDF_REQUEST_COMPLETION_ROUTINE SmplFilterCompletionRoutineRead;
```

This routine has gained ownership of the Request. It needs to complete the Request again using `WdfRequestComplete(...)`.

The driver has to subscribe for a completion routine upon each read Request to receive the Request again after the lower driver has completed it.

➔ Step 2:
Setting the completion routine for read Requests

The `EvtIoDefault` callback now has to identify Requests by calling

```
WDF_REQUEST_PARAMETERS RequestParameters;
WDF_REQUEST_PARAMETERS_INIT(&RequestParameters);

WdfRequestGetParameters(Request, &RequestParameters);
```

If the returned Request type is `WdfRequestTypeRead`, the driver needs to format the Request for forwarding to the next lower driver using `WdfRequestFormatRequestUsingCurrentType(...)` and set the completion routine by `WdfRequestSetCompletionRoutine(...)`.

After that the read Request can be forwarded to the next lower driver by `WdfRequestSend(...)`. Request send options are not used in this case.

3.4 Inspecting keyboard data

This lab inspects keyboard data by tracing it in the debugger.

Applied KMDF methods:

```
WdfRequestRetrieveOutputMemory(...)
WdfMemoryGetBuffer(...)
```

➔ Step 1:
Add keyboard header file

The following header file needs to be included:

```
#include <ntddkbd.h>
```

➔ Step 2:
Inspect data of read Request in completion routine

The buffer of the read Request can be retrieved by `WdfRequestRetrieveOutputMemory(...)` and `WdfMemoryGetBuffer(...)`. The resulting pointer can be casted to the following data structure:

```
PKEYBOARD_INPUT_DATA pKeyboardData;
```

The makecode member indicates the key.

4 Controlling a USB Device Hardware

Overview

Plug'n'Play manager informs the driver about its associated hardware resources upon device startup. This lab teaches how Windows device drivers can find, access and use their hardware resources.

WDF Drivers get the information about their hardware resources within the `EvtDevicePrepareHardware` event callback function. From within this callback routine, the USB descriptors (configuration, interface, etc) may be read and selected necessary to make the USB hardware ready to use.

This lab also teaches the basics of setting up of a USB Continuous Reader to fetch device data, and controlling it via power management callbacks.

Goal

Basic implementation of minimum necessary Plug'n'Play and Power Management for USB devices.

Setup (Driver)

The template is located in this directory:	SmplDeviceUsbKmdfLab2
New lab directory to create:	SmplDeviceUsbKmdfLab3
The following file(s) are new:	Usb.c
The following file(s) are involved:	Device.h/.c

Setup (Application)

New lab directory to create:	
The following file(s) are new:	
The following file(s) are involved:	SmplApplicationConsole.cpp

4.1 Plug'n'Play

Applied KMDF methods:

`WdfUsbTargetDeviceGetInterface (…)`
`WdfUsbTargetPipeGetIoTarget (…)`

➔ Step 1:
Inspect the wizard generated code *for USB IOTarget initialization*

See `EvtDevicePrepareHardware` callback.

➔ Step 2:
Move USB functionality to a separate new source file

Create new file Usb.c and define a new function `SmplUsbIoTargetInitialize(...)`.
Move all USB functionality from EvtDevicePrepareHardware into this new function.

```
NTSTATUS SmplUsbIoTargetInitialize(WDFDEVICE Device)
{
    WDF_USB_DEVICE_CREATE_CONFIG createParams;
    WDF_USB_DEVICE_SELECT_CONFIG_PARAMS configParams;
    PDEVICE_CONTEXT pDeviceContext = DeviceGetContext(Device);
    NTSTATUS status = STATUS_SUCCESS;

    if (pDeviceContext->UsbDevice == NULL)
    {
   …
}
```

➔ Step 3:
Trace USB pipes of the first interface

In Usb.c define a new function `SmplUsbIoTargetInterfaceTrace`. It should be prototyped as follows:

```
VOID SmplUsbIoTargetInterfaceTrace(WDFUSBINTERFACE TargetInterface);
```

Using `WdfUsbInterfaceGetNumConfiguredPipes`, loop through the USB pipes for that interface. For each pipe, get the pipe info using `WdfUsbInterfaceGetConfiguredPipe`.

Call `SmplUsbIoTargetInterfaceTrace` in `SmplUsbIoTargetInitialize`, passing in the first USB interface obtained using `WdfUsbTargetDeviceGetInterface`.

Rebuild and test the driver to see traces of the USB pipes.

➔ Step 4:
Extension of the driver defined WDFDEVICE object context data structure

The device object context structure is a driver-defined storage for device global data. Device hardware configuration parameters are an important example for device global variables. The device object context needs to be extended by the variables below:

```
typedef struct _DEVICE_CONTEXT
{
        ...
        WDFUSBPIPE UsbInterruptPipe;
        ULONG UsbInterruptMaximumPacketSize
        WDFIOTARGET UsbSpecializedIoTarget;


} DEVICE_CONTEXT, *PDEVICE_CONTEXT;
```

The accessor method `DeviceGetContext(...)` is defined globally. This method is for retrieving the associated data structure `DEVICE_CONTEXT` from the framework device object. It should be inspected now for being ready to use it later.

➔ Step 5:
Saving the USB Pipe and IoTarget in the Device Object Context Area

In Usb.c `SmplUsbIoTargetInterfaceTrace()` should be renamed and edited to return the pipe of type `WdfUsbPipeTypeInterrupt`, if any.

```
BOOLEAN SmplUsbIoTargetInterfaceInfoGet(_In_ WDFUSBINTERFACE TargetInterface,
                                        _Out_ PDEVICE_CONTEXT pSmplDeviceContext)
```

The accessor method `DeviceGetContext(...)` should be used to assign the USB Interrupt Pipe, Maximum packet size and IoTarget to the Device Object Context Area.

The IO Target associated with a pipe, can be retrieved by `WdfUsbTargetPipeGetIoTarget(...)`.

4.2 Device Level Power Management

This lab section subscribes for power management callbacks and uses them for device initialization and uninitialization.

Applied KMDF event callbacks:

```
EVT_WDF_DEVICE_D0_ENTRY
EVT_WDF_DEVICE_D0_EXIT
```

Applied framework data structures in the driver:

```
WDF_PNPPOWER_EVENT_CALLBACKS
```

➔ Step 1:
Defining device level Power Management callbacks for D0 entry and exit

The framework automatically sets device power policy ownership. So the driver can simply subscribe for power change callbacks which are triggered upon D0 entry and exit. The following two callback functions shall be declared in Device.h and implemented in Device.c:

```
EVT_WDF_DEVICE_D0_ENTRY SmplDeviceUsbEvtD0Entry;
EVT_WDF_DEVICE_D0_EXIT SmplDeviceUsbEvtD0Exit;
```

➔ Step 2:
Subscribing for device level power management events D0 entry and exit

The driver has to subscribe for the above callbacks in the `EvtDriverDeviceAdd(...)` before `WdfDeviceCreate(…)` is called.

➔ Step 3:
Testing `EvtDeviceD0Entry` and `EvtDeviceD0Exit`

The Plug'n'Play and Power Management callback functions can be tested by enabling and disabling the device in Device Manager. Debug traces must appear to verify that the functions are invoked.

4.3 USB Continuous Reader

This lab section implements a USB continuous reader.

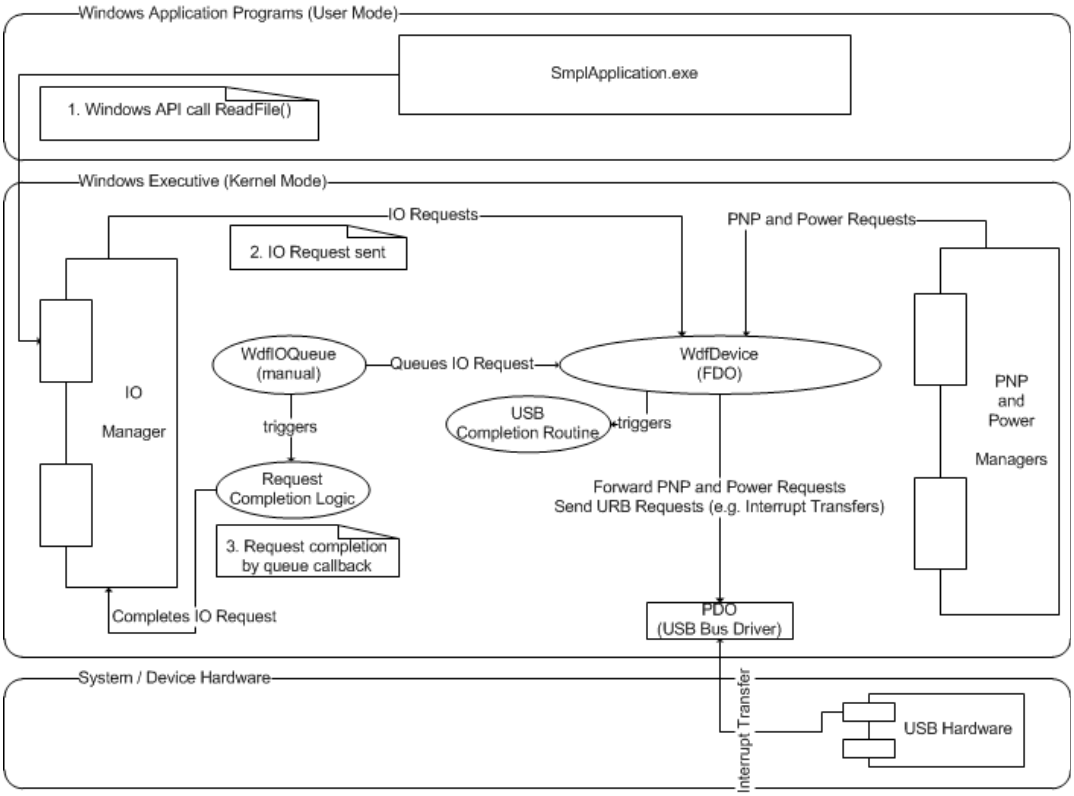
Applied KMDF methods:

```
WdfUsbTargetPipeConfigContinuousReader(...)
WdfIoTargetStart(…)
WdfIoTargetStop(…)
```

Applied KMDF event callbacks:

`EVT_WDF_USB_READER_COMPLETION_ROUTINE`

Overview diagram:



➔ Step 1:
Implementation of `EvtUsbTargetPipeReadComplete`

In Usb.c an event handler for `EvtUsbTargetPipeReadComplete` needs to be defined and implemented:

```
VOID SmplUsbEvtTargetPipeReadComplete(
   _In_ WDFUSBPIPE Pipe,
   _In_ WDFMEMORY Buffer,
   _In_ size_t NumBytesTransferred,
   _In_ WDFCONTEXT Context
)
```

All it needs to do for now is tracing the buffer

➔ Step 2:
Implementation of USB Continuous Reader

In Usb.c a new function `SmplUsbIoTargetReaderConfigure` should be defined:

The configuration structure for the Continuous Reader should be initialized using `WDF_USB_CONTINUOUS_READER_CONFIG_INIT`. The following information must be provided:

- callback function for read completion `EvtUsbTargetPipeReadComplete`
- pointer to the Device Object Contect Area maximum packet size
- transfer length retrived from the Device Object Contect Area

Next, `WdfUsbTargetPipeSetNoMaximumPacketSizeCheck` must be called to indicate driver intends to read data less than the pipe's `MaximumPacketSize`.

Then the Continuous Reader for the interrupt pipe should be configured using `WdfUsbTargetPipeConfigContinuousReader`, passing in the Continuous Reader's configuration structure.

`SmplUsbIoTargetReaderConfigure`, just like `SmplUsbIoTargetInitialize`, must be called within EvtDevicePrepareHardware.

➔ Step 3:
Starting and stopping IOTarget operation in EvtDeviceD0Entry and EvtDeviceD0Exit

The USB Continuous Reader needs to be started and stopped in `EvtDeviceD0Entry` and `EvtDeviceD0Exit`. This is done by calls to `WdfIoTargetStart(...)` and `WdfIoTargetStop(...)`.

➔ Step 4:
Testing the functionality of the USB Continuous Reader

When move the USB mouse, debug traces of `SmplUsbIoTargetEvtInterruptPipeReadComplete()` callback function must be seen.

5 Simple Application IO with Hardware

Overview

This lab demonstrates basic processing of application software IO operations as a result of device hardware actions.

The WdfIoQueue needs to be configured to manual dispatch type. WdfRequests from the application will remain pending in the queue until explicitly removed.

Data from the USB device is received from within the USB Continuous Reader's interrupt pipe completion callback routine. In this routine the driver will try to extract one WdfRequest from the WdfIoQueue and complete it.

Goal

Implementation of a Windows device driver that handles IO Operations as a result of hardware action and can handle synchronous and asynchronous IO operation from one or more user mode threads or processes and cancellation of asynchronous operations.

Setup

The template is located in the following directory	SmplDeviceUsbKmdfLab3
Own new lab directory to create:	SmplDeviceUsbKmdfLab4
The following files are involved:	Driver.c Driver.h Device.c Queue.c Usb.c
The following files are new:	Collection.c

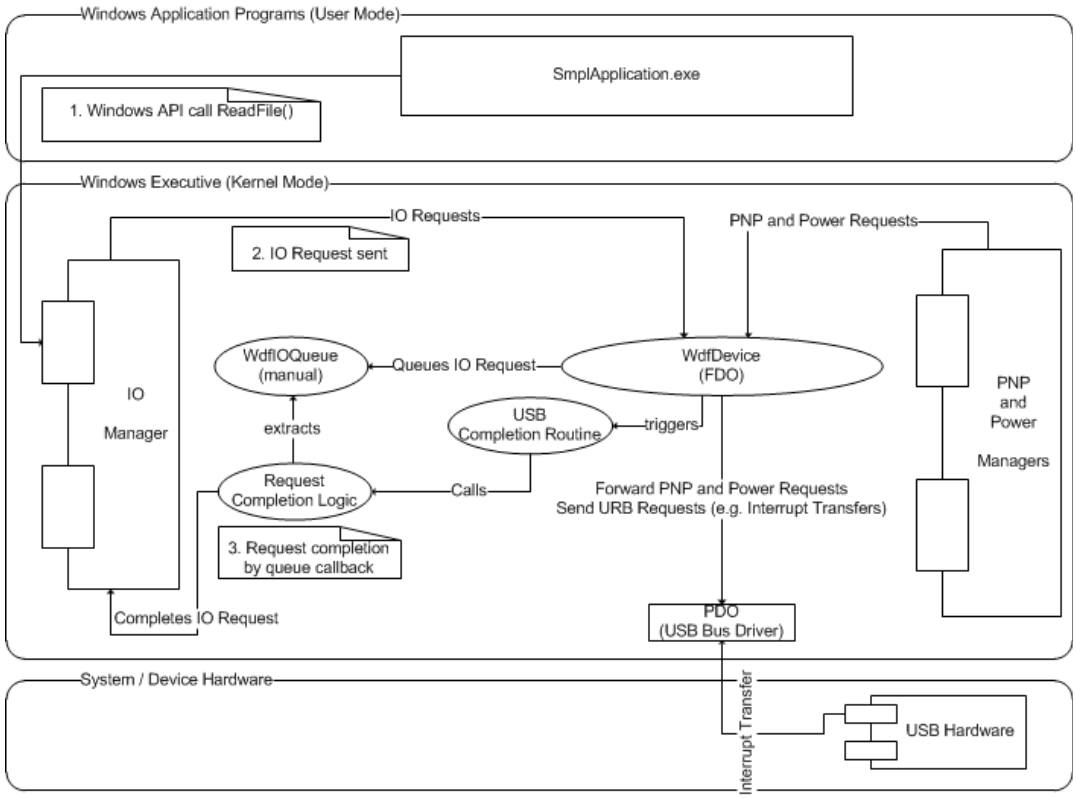
5.1 Simple Request completion upon hardware action

In this lab section, IO completion will be implemented depending on data arriving at the serial port.

Applied KMDF methods:

`WdfIoQueueRetrieveNextRequest(…)`

Overview diagram:



➔ Step 1:
Setting the queue's dispatch method to manual

In the previous exercises, the queue worked in a "push model" delivering `WDFREQUESTS` to the driver. For this lab, the queue operation needs to be changed to a "pull model" where the driver actively retrieves Requests from the queue.

The initialization of the queue in the `EvtDriverDeviceAdd(…) needs to be changed.` The second parameter of the `WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE` must be set from `WdfIoQueueDispatchParallel` to `WdfIoQueueDispatchManual`.

➔ Step 2:
Removing previous Read queue event handler function

References and implementations of all previous `WDFQUEUE` event handler functions (e.g. `EvtIoRead`) must be removed.

➔ Step 3:
Save WDFQUEUE in the device object context area

In Queue.c in `SmplDeviceUsbQueueInitialize(...)` get a pointer to the device object context area and save the WDFQUEUE handle returned by `WdfIoQueueCreate(...)` in the device object context area.

➔ Step 4:
Retrieving Requests from the queue in the callback function

After receiving the data from USB interrupt pipe, a Request has to be dequeued manually using the `WdfIoQueueRetrieveNextRequest(…)` function.

If a Request can be retrieved successfully from the queue, it has to be completed. This is done using `WdfRequestCompleteWithInformation(…)`. For now, no data shall be transferred to the application yet.

➔ Step 5:
Running the user mode application for testing

The test application created in the previous exercise can be used again to test this driver.

5.2 Application IO with USB data (partial / incomplete)

➔ Step 1:
Creating KMDF Collection object for buffering USB data

In Device.c in SmplDeviceUsbCreateDevice(...) create a WDF collection object with the device object as parent. Extend the device object context area and add a new member `WDFCOLLECTION Collection;`

Save a handle to the collection object in the device object context.

```
WDF_OBJECT_ATTRIBUTES_INIT(&Attributes);
Attributes.ParentObject = device;
status = WdfCollectionCreate(&Attributes,
                             &(deviceContext->Collection));
```

➔ Step 2:
Buffering received data in the WDF Collection

Create a new file Collection.c. In this file implement a new function

```
NTSTATUS
SmplCollectionInsertMemoryCopy(
    _In_ WDFCOLLECTION Collection,
    _In_ WDFMEMORY Memory
)
```

In Usb.c in `SmplUsbIoTargetEvtInterruptPipeReadComplete(...)` all data received from USB needs to be inserted into the Collection immediately.

➔ Step 3:
Request completion using data from Collection

In SmplUsbIoTarget.c in `SmplUsbIoTargetEvtInterruptPipeReadComplete(...)` if there is a Request retrieved from the Queue, a WDFMEMORY buffer needs to be retrieved from the Collection.

Then the request needs to be completed with data from the Collection.

➔ Step 4:
Test with sample application

Test with sample application.

5.3 Application IO with USB data (final)

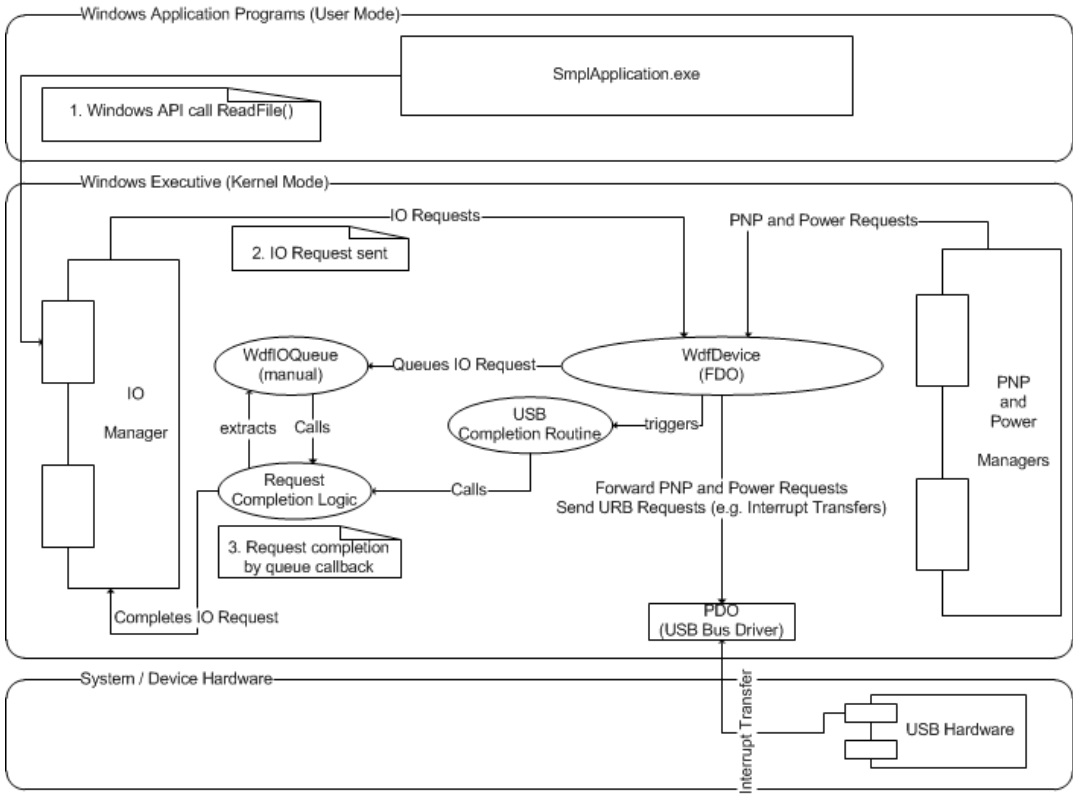
Applied KMDF methods:

`WdfDeviceInitRegisterPnpStateChangeCallback(...)`

Applied KMDF event callbacks:

`EVT_WDF_DEVICE_PNP_STATE_CHANGE_NOTIFICATION`

Overview diagram:



➔ Step 1:
Implementing a separate function for Request completion

In SmplUsbIoTarget.c in `SmplUsbIoTargetEvtInterruptPipeReadComplete(...)` the code checking Queue and completing Requests with data from Collection shall be moved to a separate function. This function should be called `SmplUsbIoTargetRequestTryComplete(...)`. This function should check first if there is data in the collection. Only if there is data in the Collection it should attempt to retrieve a Request. Only if it retrieved a Request it should remove the data from the collection and complete the Request. The above should be done until there is either no more data in the Collectin OR there is no more Request available.

➔ Step 2:
Setting the synchronization scope to `WdfSynchronizationScopeQueue`

The parent `WDFDEVICE` synchronization scope has to be set to `WdfSynchronizationScopeQueue`. The `WDFQUEUE` synchronization scope has to be set to `WdfSynchronizationScopeInheritFromParent` as shown below:

```
    WDF_OBJECT_ATTRIBUTES Attributes;
…
    WDF_OBJECT_ATTRIBUTES_INIT(&Attributes);
    Attributes.SynchronizationScope =
                            WdfSynchronizationScopeInheritFromParent;
```

```
…
    status = WdfIoQueueCreate(  …
                                &Attributes,
                                …);
```

➔ Step 3: EVT_WDF_IO_QUEUE_STATE implementation

A new callback function shall be created in Queue.c to be invoked whenever the Read Queue changes state from empty to non-empty. It can be registered by calling `WdfIoQueueReadyNotify(…) after WdfIoQueueCreate`(...). Name the callback function `SmplDeviceUsbEvtIoQueueReady(…)`. From this callback the above implemented function needs to be called too.

➔ Step 4: Synchronization by externally acquiring the Queue's object presentation lock

In SmplUsbIoTarget.c in `SmplUsbIoTargetEvtInterruptPipeReadComplete(...)` the code inserting into the Queue AND the code completing Requests needs to be synchronized `WdfObjectLockAcquire(...) and WdfObjectLockRelease(...).`

In `SmplQueueEvtState(…)`. No lock needs to be Acquired, because the callback is already holding the Queue's object presentation lock.

5.4 Application Plug'n'Play capability

Proper response to Plug'n'Play notifications enables device removal while there are still open handles by applications. Plug'n'Play notifications require a Windows GUI application. They are delivered to the application by the Windows message `WM_DEVICECHANGE`.

Applied Setup API functions in the user mode application:

`RegisterDeviceNotification(…)`

Applied Win32 API functions in the user mode application:

```
CreateFile(…)
Close Handle(…)
```

➔ Step 1: DBT_DEVTYP_HANDLE Notifications device removal

From within the handler for DBT_DEVICEQUERYREMOVE the application's open handle to the device needs to be closed.

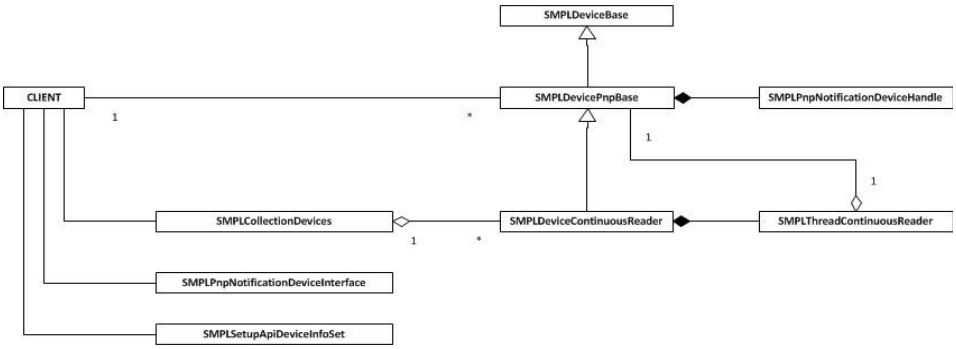
➔ Step 2: Testing Device Handle notification for DBT_DEVICEQUERYREMOVE

If the Plug'n'Play handle notification above is implemented properly, the device can be disabled in device manager even while one or multiple application(s) is (are) still running. Otherwise The system asks for a reboot.

➔ Step 3: DBT_DEVTYP_DEVICEINTERFACE notifications for device arrival

From within the handler for DBT_DEVICEARRIVAL the application needs to open a handle to the device. Device Interface symbolic link name is a parameter of the notification.

➔ Step 4:
A complete Plug'n'Play sample application



➔ Step 5:
Test using SmplApplicationPnp with private cursor pointer

Test the driver functionality running SmplApplicationPnp „as administrator“.

6 Windows Management Instrumentation

Overview

Windows Management Instrumentation is a universal programming interface for all kinds of operating system, network and application resources. WMI is a meta object model, which is flexible enough for being used with any kind of resources.

WDM drivers can use WMI mechanisms to publish information, permit configuration and supply notifications and logging of events.

Goal

Implementation of a WDM driver which can transmit runtime information data to an individual application or a management application.

Setup

The template is located in the following directory:	SmplDeviceUsbKmdfLab4
Own new lab directory to create:	SmplDeviceUsbKmdfLab5
The following files are involved:	-
The following files are new:	Wmi.c

6.1 Driver WMI Support

This lab will add driver capability to transmit statistical information to applications using WMI. This information includes the count of USB interrupts transfers, etc.

Applied KMDF methods:

```
WdfDeviceAssignMofResourceName(…)
WdfWmiInstanceCreate(…)
WdfWmiInstanceGetDevice()
```

Applied KMDF event callbacks:

```
EVT_WDF_WMI_INSTANCE_QUERY_INSTANCE
```

➔ Step 1: Creation of a schema description

The following MOF file describes the "Managed Object". It should be saved as SmplDeviceUsb.mof

```
[Dynamic, Provider("WMIProv"),
 WMI,
Description("Schema for Smpl Device USB Driver"),
 guid("DAD19B37-3F3F-480C-BEE2-8A08306AD6C0"),
 locale("MS\\0x409")]

class SmplDeviceUsb
{
          [key, read] string InstanceName;
          [read] boolean Active;
          [WmiDataId(1), Description("Count of interrupts") ]
          uint32 CountInterrupts;
};
```

➔ Step 2: Compiling, linking and registering the MOF resource

The MOF file needs to be included into the Visual Studio driver project. Visual Studio's project options for the MOF compiler issue the following command for compiling the .mof file into a .bmf file:

```
Mofcomp -B:SmplDeviceUsb.bmf SmplDeviceUsb.mof
```

The binary resource must be linked to the driver. This is achieved by the following entry in the project resource file:

```
MOFRESOURCE              MOFDATA MOVEABLE PURE    "SmplDeviceUsb.bmf"
```

In `SmplDeviceUsbEvtDeviceAdd(...)` the MOF resource needs to be registered using `WdfDeviceAssignMofResourceName(...)`.

➔ Step 3: New data structure for recording device interrupt count

In device.h a new data structure needs to be defined:

```
typedef struct _SMPL_DEVICE_WMIDATA
{
 ULONG WmiCountInterrupts;

}SMPL_DEVICE_WMIDATA, *PSMPL_DEVICE_WMIDATA;
```

A variable of this type `SMPL_DEVICE_WMIDATA` needs to be added to the device object context.

```
typedef struct _DEVICE_CONTEXT
{
...
    SMPL_DEVICE_WMIDATA WmiData;

} DEVICE_CONTEXT, *PDEVICE_CONTEXT;
```

This variable shall be initialized to 0 in `EvtDriverDeviceAdd` and shall be incremented upon each USB interrupt transfer.

➔ Step 4: Implementation of EvtWmiInstanceQueryInstance callback

A new file wmi.c needs to be created and included in the project. The following WMI callback needs to be impemented:

```
NTSTATUS
SmplWmiEvtQueryInstance(
    _In_ WDFWMIINSTANCE WmiInstance,
    _In_ ULONG OutBufferSize,
    _Out_writes_bytes_to_(OutBufferSize, *BufferUsed) PVOID OutBuffer,
    _Out_ PULONG BufferUsed
    )
{
    PDEVICE_CONTEXT pDeviceContext =
            DeviceGetContext(WdfWmiInstanceGetDevice(WmiInstance));

    RtlZeroMemory(OutBuffer, OutBufferSize);
    RtlCopyMemory(OutBuffer, &(pDeviceContext->WmiData),
                  sizeof(pDeviceContext->WmiData));

    *BufferUsed = sizeof(pDeviceContext->WmiData);
    return STATUS_SUCCESS;

}
```

➔ Step 5: Creating and registering an instance of a WMI data provider

To register as a WMI data provider with the WMI repository, a new function needs to be implemented in wmi.c :

```
NTSTATUS
SmplWmiRegistration(
    WDFDEVICE Device
    )
```

This function first assigns the mof resource name:

```
DECLARE_CONST_UNICODE_STRING(mofRsrcName, SMPLMOFRESOURCENAME);

status = WdfDeviceAssignMofResourceName(Device, &mofRsrcName);
```

Then it registers the provider and create the instance in one call to `WdfWmiInstanceCreate()` as shown below:

```
WDF_WMI_PROVIDER_CONFIG_INIT(&providerConfig,
                                  &GUID_WMI_DATA_SMPLDEVICE);

providerConfig.MinInstanceBufferSize = sizeof(SMPL_DEVICE_WMIDATA);

WDF_WMI_INSTANCE_CONFIG_INIT_PROVIDER_CONFIG(&instanceConfig,
                                             &providerConfig);

instanceConfig.Register = TRUE;
instanceConfig.EvtWmiInstanceQueryInstance = SmplWmiEvtQueryInstance;
```

```
status = WdfWmiInstanceCreate(Device,
                              &instanceConfig,
                               WDF_NO_OBJECT_ATTRIBUTES,
                               WDF_NO_HANDLE);
```

In Device.c `SmplDeviceUsbEvtDeviceAdd(...)` the driver registers as WMI data provider invoking `SmplWmiRegistration(...)`. Then the driver will start to receive WMI callbacks and makes data blocks available to WMI clients.

6.2 A Visual Basic WMI Consumer

Applied Visual Basic functions:

`GetObject(…)`

➔ Step 1: Basic WMI access from Visual Basic Script

The current version number of WMI can be retrieved by the following small sample script:

```
Dim TestObject            'AS Object

Set TestObject = GetObject_
("winmgmts:root\default:__cimomidentification=@")

MsgBox "WMI Version: " & TestObject.versionusedtocreatedb
```

➔ Step 2: Access to WMI driver data from Visual Basic Script

To access driver statistical data from Visual Basic Script the following sample program is needed:

```
Dim objServ           'As WbemScripting.SwbemServices
Dim objWO             'As WbemScripting.SwbemObject
Dim objObjectset      'As WbemScripting.SWbemObjectSet

Set objServ = GetObject("winmgmts:root\WMI")
Set objObjectset = objServ.InstancesOf("SMPLDEVICEUSB")

For Each obj In objObjectset
         MsgBox obj.InstanceName
         MsgBox obj.CountInterrupts
Next                  'obj
```

6.3 WPP tracing via WinDbg on host machine (remote)

➔ Step 1: Extract .tmf and .tmc files from the .pdb file

The .tmc and .tmf files need to be extracted from the .pdb file like e.g:

```
"C:\Program Files (x86)\Windows Kits\8.0\Tools\x64\traceview.exe"
-parsepdb SmplDeviceUsb.pdb
```

➔ Step 2: Set WMI trace search path

First, the correct operating system symbols need to be loaded. This can be done by setting WinDbg symbol file path to

```
SRV*X:\YourLocalSymbolFolder*http://msdl.microsoft.com/download/symbol
s
```

The WMI trace info search path needs to point to the .tmf and .tmc files.

```
!wmitrace.searchpath
X:\...\SmplDeviceKmdfLab1\SmplDeviceUsb\Win8betaDebug
```

➔ Step 3:
Activate and deactivate WMI traces

A named trace session needs to be started with output redirected to the kernel debugger.
```
!wmitrace.start XYZ -kd
```

Traces can be enabled for the individual trace GUID {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx} which the wizard has generated in the header file trace.h
```
!wmitrace.enable XYZ XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX -level 0xff
-flag 0x7fffffff
```

The following commands disable and stop traces again:

```
!wmitrace.disable
!wmitrace.stop
```

6.4 WPP tracing via Traceview on target machine (local)

Adding trace messages in various execution paths of the driver helps uncover the driver's behavior. Visual Studio's driver template uses WPP to implement this tracing feature, and functional WPP trace statements are already inserted into the template-generated source codes.

Various WDK tools could be used to show the WPP traces generated by the driver. In this training, WDK TraceView is used as both the “controller” and “consumer” of the trace messages extracted from the UMDF driver--the trace “provider”. And one way to setup the driver as the provider is to use its symbol file.

➔ Step 1:
Get a copy of the driver's PDB symbol file

Save a copy of `SmplDevice.pdb` into **X:\MyLocalDriverSymbolFilePath**.

➔ Step 2:
Setup TraceView for showing WPP traces

Launch TraceView **as Administrator**. (TraceView.exe is located in **Tools\x86** sub-folder of WDK 8 installation directory).

From the **File** menu, choose **Create New Log Session**.

Click on **Add Provider** button.

Select **PDB (Debug Information) File** option as the method to obtain control GUID information, and provide the absolute file name of the driver's PDB file (e.g. `X:\MyLocalDriverSymbolFilePath\SmplDevice.pdb`).

Click **OK**.

Click on **Next >** button to open the **Log Session Options** dialog.

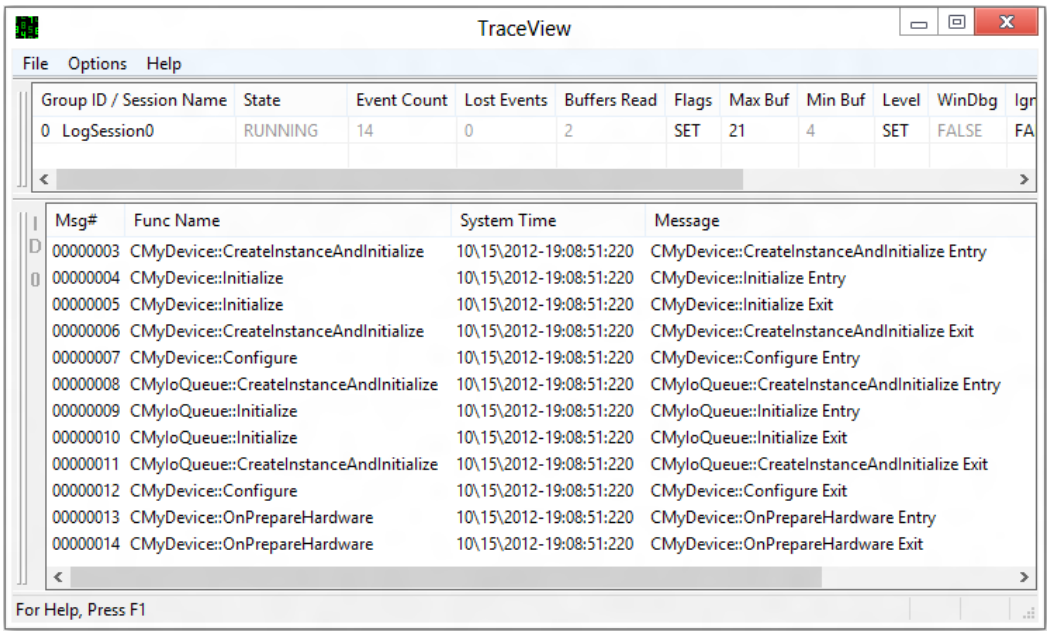
Click on **>>** button for the **Set Flags and Level** option.

In **Tracing Flags and Level Selection** dialog, right-click on the GUID, and in the pop-up menu choose **Level > Reserved9**, and click **OK**.

Click **Finish**.

➔ Step 3:
Inspect basic driver operations from the traces

In Device Manager, disable and re-enable the device associated with the driver, and view the generated real-time trace messages.



7 Driver Signing

Overview

This lab demonstrates the basics of code signing using a test Digital ID.

Microsoft Visual Studio 2012's automated driver test signing will be overridden. Specifically:

- a test certificate will be created,
- a PFX file will be generated from the test certificate's SPC and PVK files,
- the driver package will be manually signed using the PFX file (cross-certificate not needed), and
- the signed driver package will be installed on a test machine with TESTSIGNING mode enabled.

The skill gained, however, may be used to sign drivers using a production certificate issued by a trusted certification authority like **VeriSign**.

Goal

Creation of a test certificate, signing of driver, and installation of the signed driver package.

Setup

The template is located in the following directory	SmplDeviceUsbKmdfLab5
Own new lab directory to create:	C:\TestSigning\DriverPackage
The following files are involved:	.inf file .cat file .sys file
The following files are new:	.cer file .spc file .pvk file

7.1 Preparation of the driver package

➔ **Step 1:**
Rebuilding the driver in Visual Studio

Rebuild the driver. And for convenience, move the contents of the generated driver package into a new working directory named C:\TestSigning\DriverPackage.

➔ **Step 2:**
Inspection of the generated driver package

Open the INF file, and notice the CatalogFile entry. Also notice that both SYS and CAT files have been signed already. They will be signed with a different certificate shortly.

Delete the CAT file.

7.2 Creation of a test Digital ID

Used WDK tools:

```
MakeCert.exe
Cert2Spc.exe
```

➔ **Step 1:**
Creation of a test CER file with accompanying PVK file

```
makecert -n "CN=TestCert Kmdf Training" -pe -r -sv
       C:\TestSigning\MyPVK.pvk C:\TestSigning\MyCER.cer
```

➔ **Step 2:**
Generation of an SPC file from the CER file

```
cert2spc C:\TestSigning\MyCER.cer C:\TestSigning\MySPC.spc
```

➔ **Step 3:**
Generation of a PFX file from the SPC and PVK files

```
pvk2pfx -spc C:\TestSigning\MySPC.spc -pvk C:\TestSigning\MyPVK.pvk
       -pi <your PVK password> -pfx C:\TestSigning\MyPFX.pfx -f
```

NOTE: The same PFX file can be reused for signing the same driver package whenever the driver is rebuilt afresh, and other drivers.

7.3 Signing and Testing

Used WDK tools:

```
Inf2Cat.exe
SignTool.exe
```

➔ **Step 1:**
Generation of a CAT file

```
inf2cat.exe /driver:C:\TestSigning\DriverPackage /os:8_X86 /verbose
```

➔ **Step 2:**
Signing the SYS and CAT files using the PFX file

```
signtool sign /v /f C:\TestSigning\MyPFX.pfx /p <your PFX password>
        /t http://timestamp.verisign.com/scripts/timstamp.dll
        C:\TestSigning\DriverPackage\<SYS file>
```

```
signtool sign /v /f C:\TestSigning\MyPFX.pfx /p <your PFX password>
       /t http://timestamp.verisign.com/scripts/timstamp.dll
       C:\TestSigning\DriverPackage\<CAT file>
```

➔ Step 3:
Enabling TESTSIGNING mode in the test machine

In the test machine, verify the `testsigning` mode status using `bcdedit`. If not yet set, do:

```
bcdedit -set testsigning on
```

Then restart the test machine.

➔ Step 4:
Installing the test certificate

Copy the signed driver package into the test machine. Right-click on the CAT file, and choose **Properties | Digital Signatures** tab, and follow the steps to view and install the certificate.

IMPORTANT: The certificate must be explicitly installed in **Local Machine**, **Trusted Root Certification Authorities** store.

➔ Step 5:
Installing the signed driver

In the test machine, update the device driver using Device Manager. During installation, you must see:

- the message `This driver has an Authenticode(tm) signature` in the **Update Driver Software – USB Input Device** box, and
- a confirmation message to trust the driver's publisher in the **Windows Security** box.

8 Appendix

8.1 WinDbg useful commands

➔ Symbol file path

Symbol file path (own symbols only):
```
X:\MyLocalDriverSymbolFilePath
```

Symbol file path (own symbols AND OS symbols from Microsoft internet symbol server):
```
SRV*X:\MyLocalOsSymbolCachePath*http://msdl.microsoft.com/download/symbols;X:\MyLocalDriverSymbolFilePath
```

Hint:	Using internet symbols can take very long loading times (while symbols are not yet locally cached). This can make WinDbg hang for a long time (minutes!) e.g. when setting breakpoints.

➔ Symbol loading and breakpoints

Symbol file path (pointing to Microsoft internet symbol server):
```
SRV*X:\MyLocalOsSymbolCachePath*http://msdl.microsoft.com/download/symbols;X:\MyLocalDriverSymbolFilePath
```

```
bl                      (list breakpoints)
bc *                    (clear all breakpoints)
.reload                 (reload symbols)
.reload -i              (reload symbols ignoring bad symbols)
.reload -u              (unload symbols)
```

➔ Crash dump analysis

```
!analyze -v             (crash dump analysis)
```

➔ Debugger extension wdfkd

```
.load wdfkd.dll         (load wdf debugger extension)
!wdfkd.help             (shows usage information)
```

8.2 WinDbg Target Driver File Replacement

➔ Step 1: Creation of a “driver replacement map file”

A "driver replacement map file" needs to be created and saved on the host machine. Example:

```
map
\Systemroot\system32\drivers\SmplDeviceUsb.sys
C:\TrainingWdfKmdf\SmplDeviceKmdfLabX\objchk_win7_x86\SmplDeviceUsb.sys
```

➔ Step 2: Activating the replacement in WinDbg using .kdfiles

Windbg needs to be opened and connected to the target computer. Then the following command needs to be entered in the command line:

```
> .kdfiles <map file path>
```

Now the driver replacement map takes effect and the target target PC can be rebooted.

➔ **Step 3:**
Deleting the driver replacement map

If the driver replacement map is not deleted, it persists until the debugger is exited. Whenever the target pc is about to load a driver, it queries the kernel debugger to determine whether this driver has been mapped.

For deleting, the following command needs to be entered in the command line:

```
> .kdfiles -c
```

8.3 Debugging via COM port

➔ **Step 1:**
Create COM ports in the virtual machine

While the virtual machine is still switched off, install two new serial ports using the “Virtual Machine Settings” menu. The COM ports should be shown as COM2 and COM3. The named pipe for COM2 should be “\\.\pipe\vpcserial“. The named pipe for COM3 should be “\\.\pipe\vpcdebug“.

➔ **Step 2:**
Enable kernel mode debugging on the virtual Target machine

To allow the Host to control the virtual Target machine, the COM port must be set to a named pipe. In “Virtual Machine Settings” menu, make sure that COM3 is present and assigned to named pipe “\\.\pipe\vpcdebug”.

Run bcdedit from a command line windows on the Target “as administrator”. In this command line Window please enter:

```
bcdedit /dbgsettings serial debugport:3 baudrate:115200
bcdedit /debug on
```

Target needs to be shut down and started again (caution: reboot might not be enough). After that, the kernel debugger will be active and WinDbg can connect.