WDM Device Driver Development with WDF KMDF

Hands On Lab Manual

Manila / Philippines

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 device 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

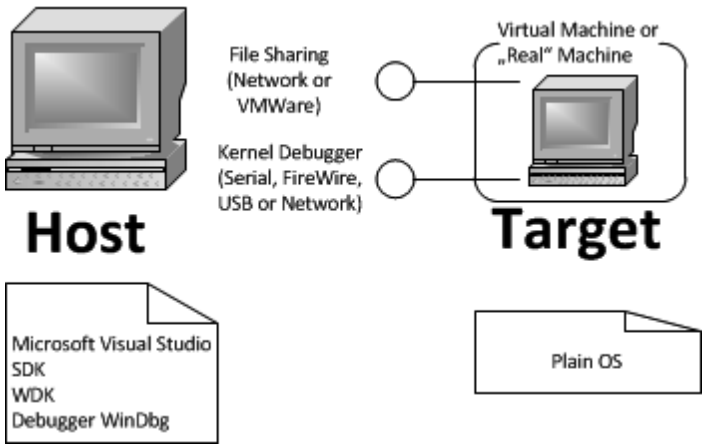
Upon completion of this lab the student will be able to compile, install and debug a minimum WDF KMDF driver template on Microsoft Windows platforms.

Setup

The template is located in the following directory:	-
Own new lab directory to create:	SmplDeviceKmdf\SmplDeviceKmdfLab1
The following files are involved:	
The following files are new:	Driver.c Device.c Driver.h Device.h SmplDevice.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

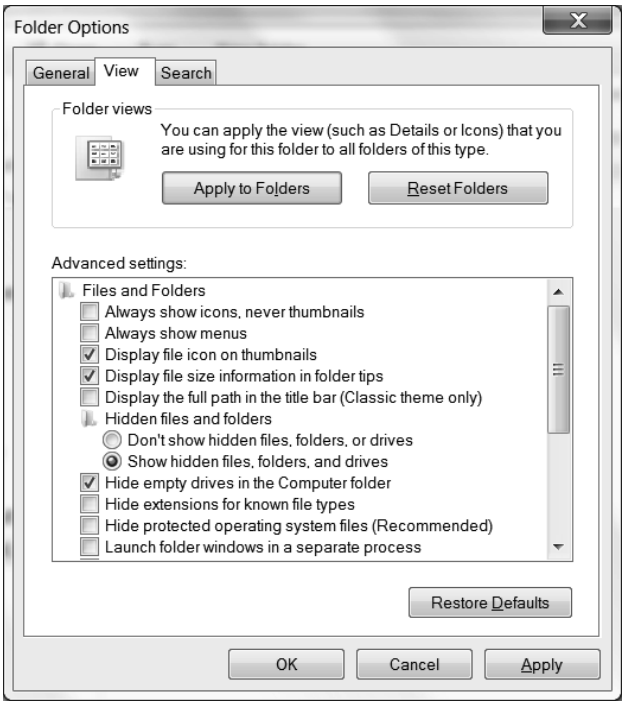
Install the software development tools below on the Host (Debugger) machine in exactly the same sequence:
- Visual Studio 11 (Release Candidate)
- Windows 8 SDK (Release Preview)
- Windows 8 WDK (Release Preview)

➔ Step 2:
Setting up the virtual Target machine environment

1. Install the virtual machine environment on the Host machine:
   - VMWare Player 4.04 or above
2. Install the guest operating system on the virtual Target Machine:
   - Microsoft Windows 8 Release Preview
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 boot entry for WinDbg (serial)”`

Hint: In case the target is a 64 bit operating system, the setting below might help installing unsigned drivers:
`bcdedit -set loadoptions DDISABLE_INTEGRITY_CHECKS`

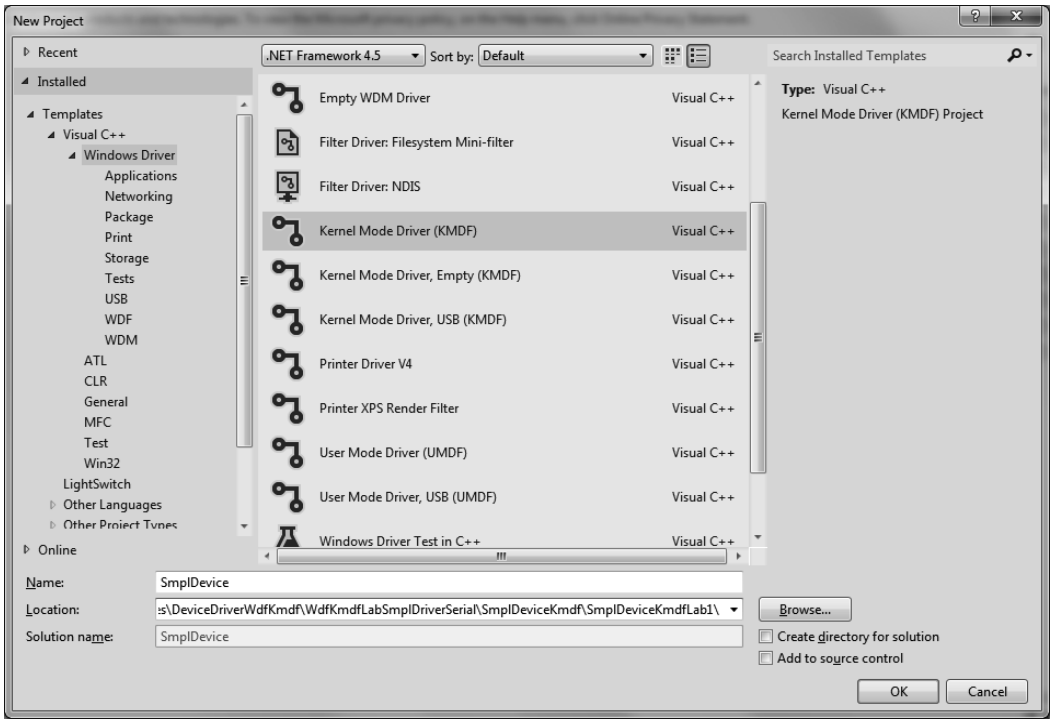
➔ Step 3:
Setting folder options



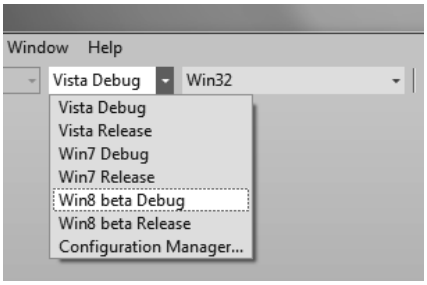
Three folder options should be changed:
- “Show hidden files, folders and drives” (enable)
- “Hide extensions for known file types” (disable)
- “Hide protected operating system files” (disable)

➔ 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 (KMDF”). The new project shall be named SmplDevice.



Compile this wizard generated driver template for Win8 beta Debug:



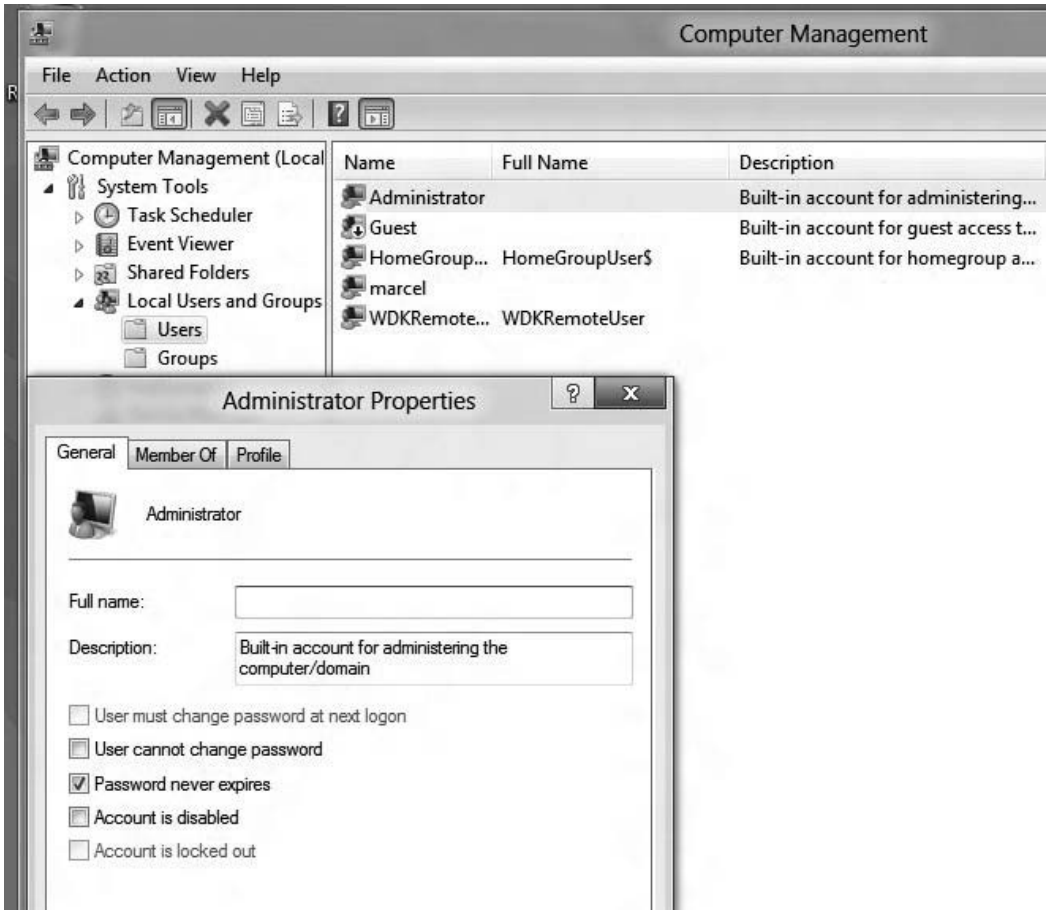
After compiling, the newly built driver executable (.sys file) is available in the appropriate subdirectory.

1.2 Setting up the test environment

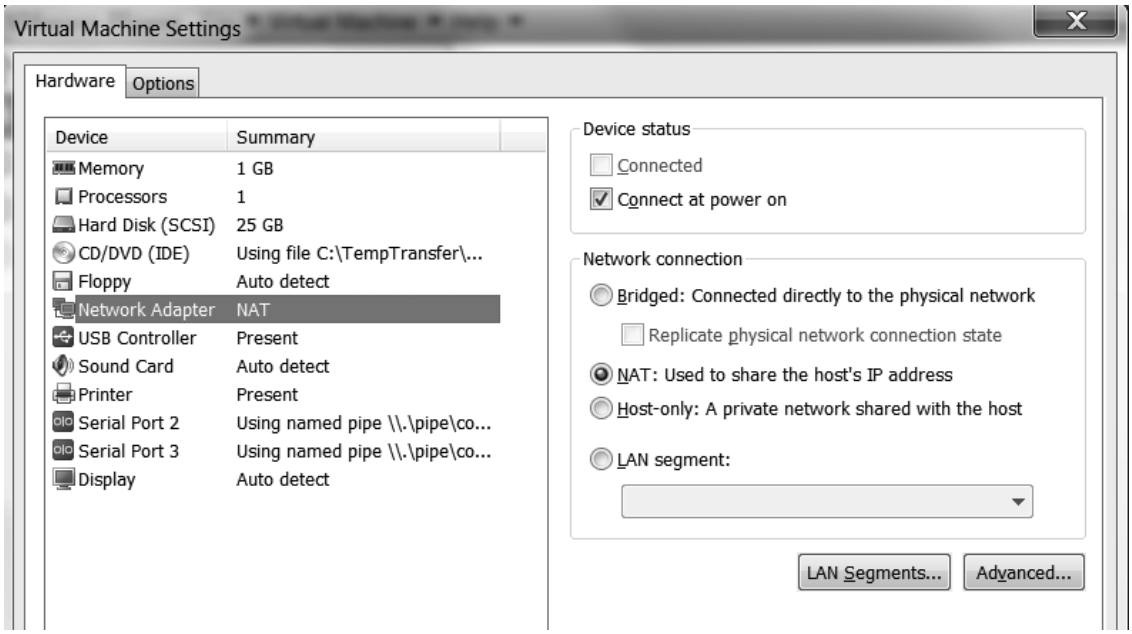
This chapter makes the students familiar with the WDK test environment in Visual Studio 11.

➔ 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.

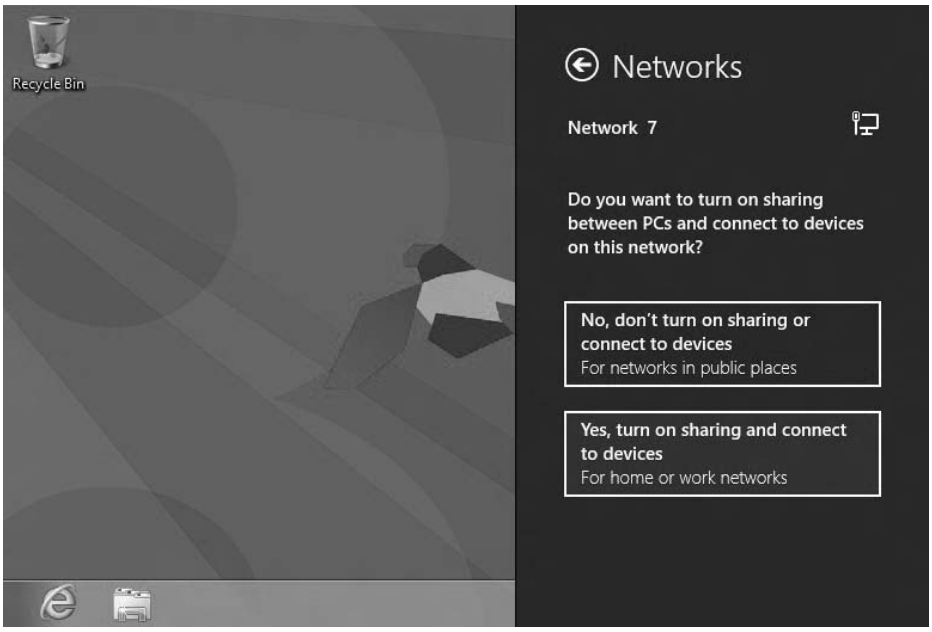


In VMWare “Virtual Machine Settings” set the network adapter to NAT (Network Address Translation) and make sure that a network connection exists between Host and virtual Target machine (using ipconfig and ping commands).



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

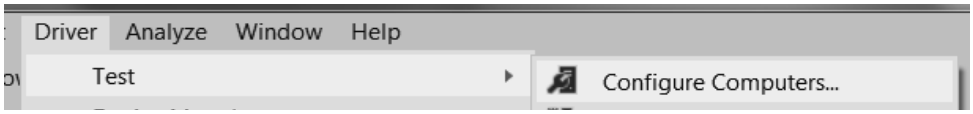
Enable file sharing;

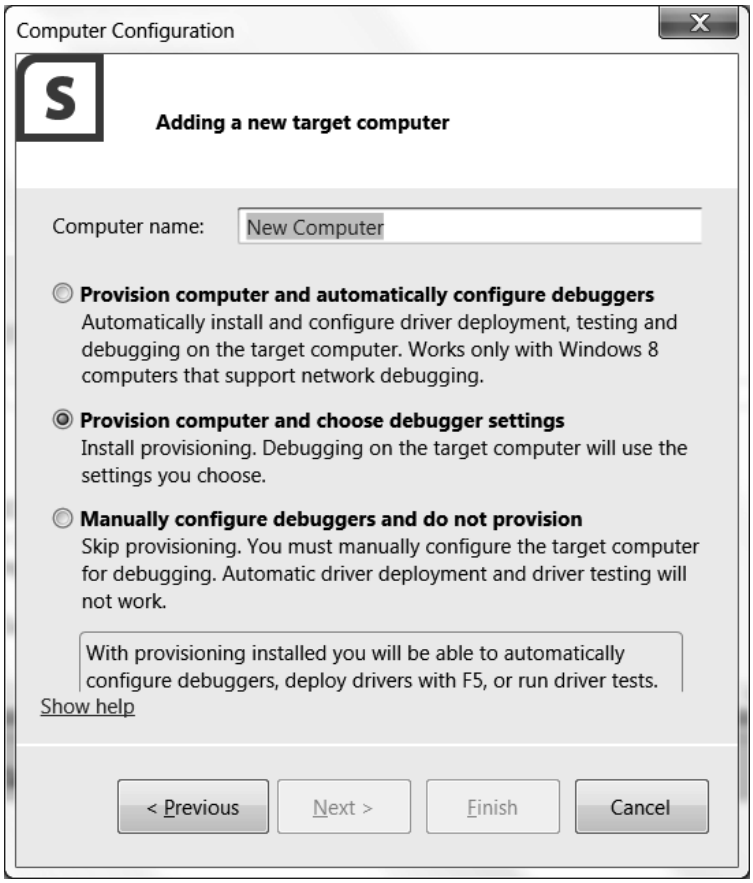


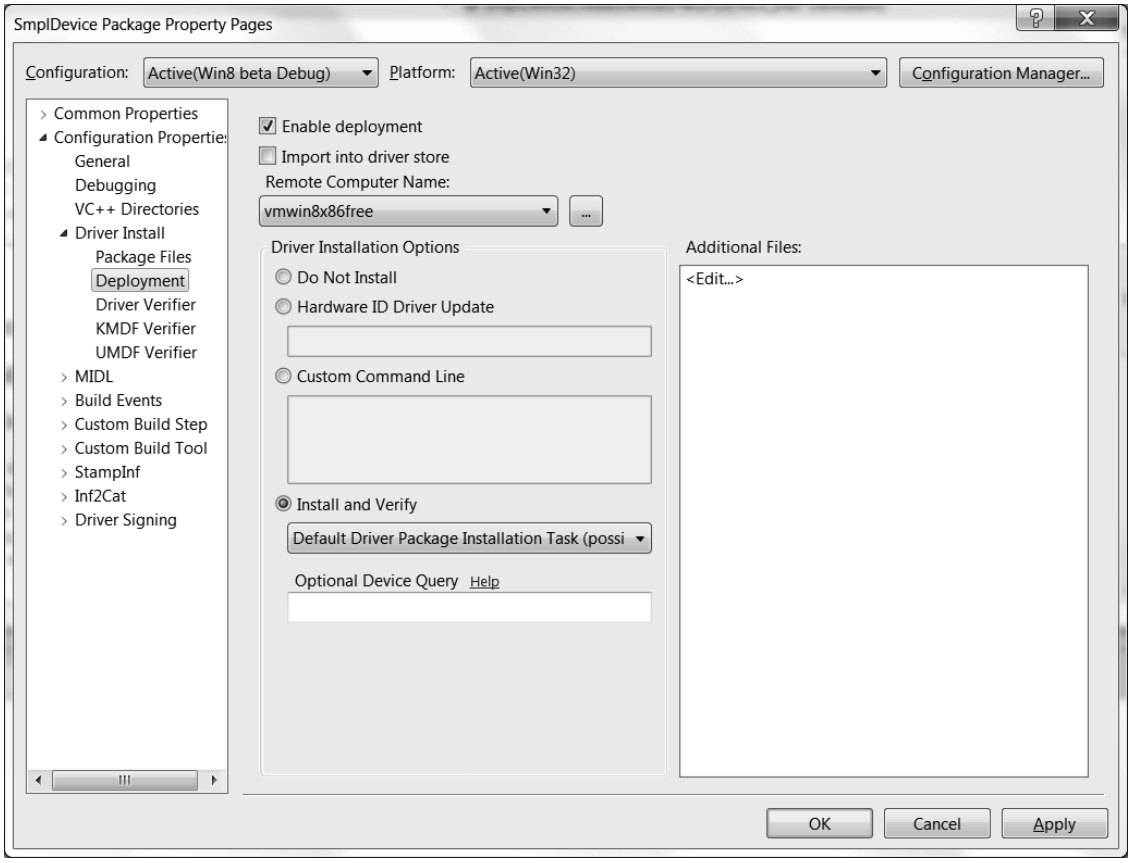
Make sure you can connect to the Target's share “users” from the host. In case the connection to the share is not successful, disconnect all other network connections on the host (e.g. company network, wireless, internet, etc.)

➔ Step 2:
Configure automatic driver deployment in Visual Studio 11

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







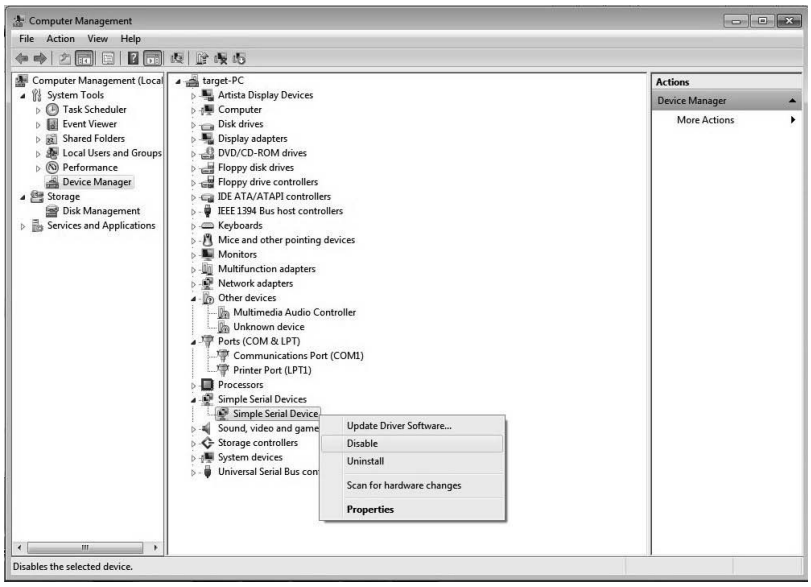
In the SmplDevice Package project properties, configure deployment to the virtual Target machine.

➔ **Step 3:**
Automatic driver deployment to the virtual Target machine

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

➔ **Step 4:**
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 (only path to own driver symbols, OS symbols NOT necessary)
- File menu → Source File Path

When compiling the driver, the debug symbols are compiled into a separate .pdb (Program Data Base) file.

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.

On Windows XP `DPFLTR_DEFAULT_ID` can be used with `DbgPrintEx(...)`.

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:	SmplDeviceKmdf\SmplDeviceKmdfLab1
Own new lab directory to create:	SmplDeviceKmdf\SmplDeviceKmdfLab2
The following files are involved:	Driver.c Device.c Driver.h Device.h SmplDevice.inf
The following files are new:	Queue.c Queue.h Public.h

Setup Application

Own new lab directory to create:	SmplDeviceKmdf\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`

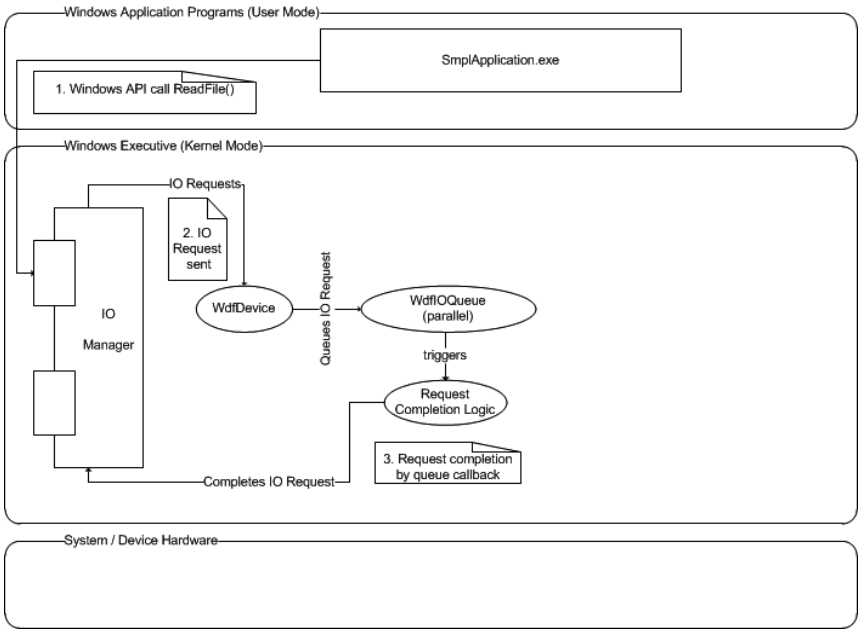
Applied KMDF event callbacks:

`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 Controlling Serial 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.

When the sample driver is installed on the serial port hardware of the PC it will receive an IO port address and an interrupt vector both already translated by the HAL (Hardware Abstraction Layer). The driver has to connect an interrupt service routine and must remember and save the register address of the IO port for later use writing to this register and reading from this register.

WDF Drivers get the information about their hardware resources within the `EvtDevicePrepareHardware` event callback function. From within this callback routine the registers have to be retrieved and if necessary they have to be mapped to virtual memory.

WDF drivers usually don't have to deal with the interrupt resource explicitly. This is implicitly done by the framework. The WDF driver just has to create an interrupt object using `WdfInterruptCreate()`. Then the framework will automatically find the interrupt resource and connect the interrupt service routine.

The power change callback `EvtDeviceD0Entry` is used to initialize the hardware

Goal

Writing a device driver which accesses its associated hardware resources via Plug'n'Play manager and HAL. This includes enabling hardware interrupts on the device and having a rudimentary interrupt service routine called.

Setup

The template is located in the following directory:	SmplDeviceKmdf\SmplDeviceKmdfLab2
Own new lab directory to create:	SmplDeviceKmdf\SmplDeviceKmdfLab4
The following files are involved:	Driver.c Driver.h Device.c Queue.c
The following files are new:	Interrupt.c Hw8250.h Hw8250.c

3.1 Driver setup with associated hardware resources

This lab section installs the driver as a Plug'n'Play Function Driver on top of a PDO for a serial port hardware device. First the .INF file needs to be modified to assign the driver to the existing serial port hardware. Then the driver for the serial port needs to be reinstalled replacing the system supplied driver by the SmplDevice driver.

Applied KMDF methods:

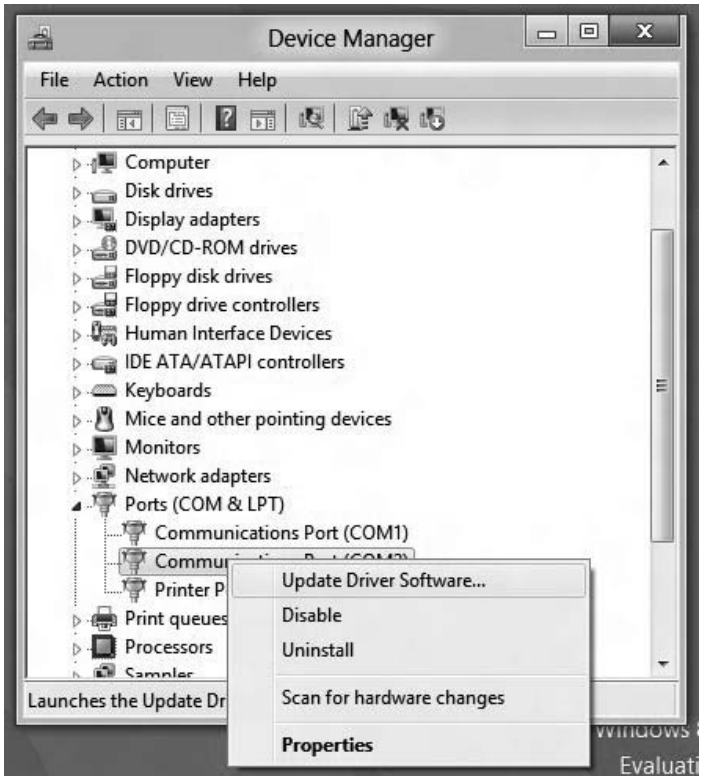
–

➔ Step 1: Changing the hardware ID in the .INF file

The Plug 'n Play Hardware ID of the COM port can be found in the Device Manager by looking in the device properties of the COM port. Change the hardware ID in the .INF file from Root\SmplDevice to the Hardware ID of the COM port.

➔ Step 2: Driver installation/update on the Target

To install the driver, right-click on the COM2 port and select the Update Driver option.



After installing the driver package with the updated .INF file, please uninstall or disable the previous instance of SmplDevice driver which does not have any hardware resources.

3.2 Plug'n'Play Events

The driver has to subscribe for Plug'n'Play callbacks to receive its hardware resources.

Applied KMDF methods:

```
WdfDeviceInitSetPnpPowerEventCallbacks(…)
WdfCmResourceListGetCount(…)
WdfCmResourceListGetDescriptor(…)
```

Applied KMDF event callbacks:

```
EVT_WDF_DEVICE_PREPARE_HARDWARE
EVT_WDF_DEVICE_RELEASE_HARDWARE
```

Applied framework data structures in the driver:

```
WDF_PNPPOWER_EVENT_CALLBACKS
```

Applied framework macros in the driver:

```
WDF_PNPPOWER_EVENT_CALLBACKS_INIT
```

➔ Step 1:
Implementation of event callback functions for preparing the hardware and releasing the hardware

Two device event callback functions have to be declared in Device.h and implemented in Device.c. First, an empty implementation containing debug traces is enough.

```
EVT_WDF_DEVICE_PREPARE_HARDWARE SmplDeviceEvtPrepareHardware;
EVT_WDF_DEVICE_RELEASE_HARDWARE SmplDeviceEvtReleaseHardware;
```

➔ Step 2:
Subscribing for Plug 'n Play event callbacks

The driver has to subscribe for the above callbacks in the `EvtDriverDeviceAdd(...)` before calling `WdfDeviceCreate(…)` in the function `SmplDeviceCreateDevice(...)`

```
WDF_PNPPOWER_EVENT_CALLBACKS PnpPowerCallbacks;
...
WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&PnpPowerCallbacks);
PnpPowerCallbacks.EvtDevicePrepareHardware =
                                    SmplDeviceEvtPrepareHardware;
PnpPowerCallbacks.EvtDeviceReleaseHardware =
                                    SmplDeviceEvtReleaseHardware;
WdfDeviceInitSetPnpPowerEventCallbacks(    DeviceInit,
                                           &PnpPowerCallbacks);
```

➔ Step 3:
Testing functionality by disabling and enabling in Device Manager

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

➔ 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 that the driver uses. Device hardware configuration parameters are an important subset of device global variables. The device object context needs to be extended by the variables below:

```
typedef struct _DEVICE_CONTEXT
{
       ...
       ULONG PortAddressLength;
       PUCHAR pPortAddress;

} DEVICE_CONTEXT, *PDEVICE_CONTEXT;
```

➔ Step 5:
Inspection of the accessor method for the Object Context Area

The accessor method `WdfObjectGet_DEVICE_CONTEXT(...)` is defined globally. This method is 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 6:
Saving the HAL translated IO space register address

The function `SmplDeviceEvtPrepareHardware` shall take apart the resource list which is already translated by the HAL. The list must contain one interrupt and one IO port address. The port address has to be saved in the device context.

The following WDF functions can be used to retrieve the resource list:

```
WdfCmResourceListGetCount(…)
WdfCmResourceListGetDescriptor(…)
```

The basic approach of this routine is shown by the code fragment below:

```
for (index = 0; index < CountResource; index++)
{
  .
  .
  .

   switch (pCMResourceDescPartial->Type)
   {
      case CmResourceTypeMemory:
         …
      case CmResourceTypePort:
         …
      case CmResourceTypeInterrupt:
         …
      default:
         …
   }
}
```

3.3 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 Driver.h and implemented in Device.c:

```
EVT_WDF_DEVICE_D0_ENTRY SmplDeviceEvtD0Entry;
EVT_WDF_DEVICE_D0_EXIT SmplDeviceEvtD0Exit;
```

➔ 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.

3.4 Hardware resources and device register access

This lab section increases the driver with routines to read, to save and to access its hardware resources. The registers residing in port address space will be made accessible via the HAL.

Applied HAL macros in the driver:

```
READ_PORT_UCHAR(…)
WRITE_PORT_UCHAR(…)
```

➔ Step 1: Integration of the prepared hardware access routines

For easy access to the serial port hardware, the files Hw8250.c Hw8250.h shall be used. These have been coded for the sole purpose of easily accessing the serial port hardware during this lab. For hardware access they are using the HAL macros `READ_PORT_UCHAR(…)` and `WRITE_PORT_UCHAR(…)`.

➔ Step 2: Hardware initialization in the EvtDeviceD0Entry callback

Basic initialization of the serial interface chip is done in `EvtDeviceD0Entry` by the following routine from the file Hw8250.c.

```
Hw8250Init(pSmplDeviceContext->pPortAddress,1200,7,1,0)
```

3.5 Interrupts

This lab section implements a minimum interrupt service routine. This routine does a check if the interrupt is coming from the right device by reading an interrupt status register of the device. In case it is not, this is indicated to Windows by returning `FALSE` and no action is taken. In case the interrupt was fired by the right device, the interrupt is handled. In this case data is read from the port and the interrupt condition on the device is reset. After this, the successful completion is signalled to Windows.

Applied KMDF methods:

```
WdfInterruptCreate(…)
WdfInterruptGetDevice(…)
```

Applied KMDF event callbacks:

```
EVT_WDF_INTERRUPT_ENABLE
EVT_WDF_INTERRUPT_DISABLE
EVT_WDF_INTERRUPT_ISR
```

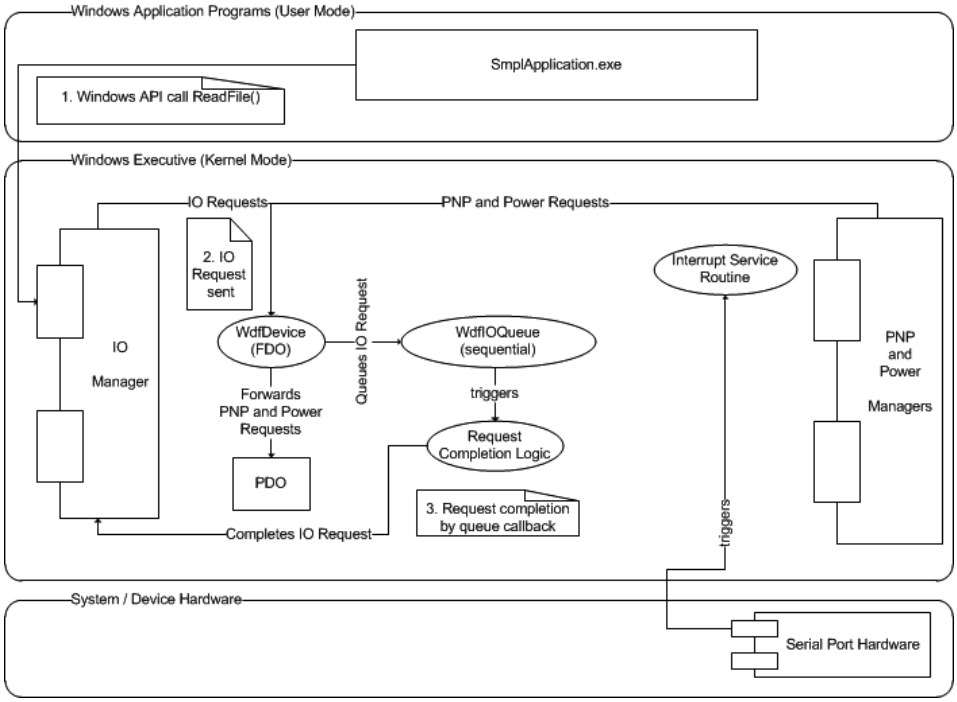
Applied framework data structures in the driver:

```
WDF_OBJECT_ATTRIBUTES
WDF_INTERRUPT_CONFIG
```

Applied framework macros in the driver:

```
WDF_INTERRUPT_CONFIG_INIT
```

Overview diagram:



➔ Step 1: Creating the event callback functions for the WDFINTERRUPT object

A new file Interrupt.c shall be created. The following callbacks of the interrupt object have to be declared and implemented:

```
EVT_WDF_INTERRUPT_ENABLE SmplInterruptEvtEnable;
EVT_WDF_INTERRUPT_DISABLE SmplInterruptEvtDisable;
EVT_WDF_INTERRUPT_ISR SmplInterruptEvtIsr;
```

For now, these event callback function implementation can remain empty. They should only contain trace outputs for invocation test.

➔ Step 2:
Creating the WDFINTERRUPT object in the EvtDriverDeviceAdd routine

The driver calls the `WDF_INTERRUPT_CONFIG_INIT` macro to initialize the `WDF_INTERRUPT_CONFIG` structure with a pointer to the `EvtInterruptIsr` callback function that the driver implements for the interrupt. The driver also needs to subscribe `EvtInterruptEnable` and `EvtInterruptDisable` callback functions upon calling `WdfInterruptCreate(…)`.

➔ Step 3:
EvtInterruptIsr implementation identifying the interrupt source

First of all, the Interrupt Service Routine must read the interrupt status register of the device. It has to determine if the interrupt source is the correct hardware device. If this isn't the case the ISR has to return immediately without taking any action indicating status FALSE.

```
UCHAR IntId;
.
.
.

IntId=READ_INTERRUPT_IDENTIFICATION(pSmplDeviceContext->pPortAddress);

if (IntId & IIR_NOT_FIRED)
{
   return FALSE;
}
```

The `WDFDEVICE` object can be retrieved by calling the `WdfInterruptGetDevice(...)`.

➔ Step 4:
EvtInterruptIsr implementation handling the interrupt

After making sure that the interrupt originated from the correct hardware, the interrupt has to be confirmed to that hardware and the hardware interrupt condition has to be reset. Windows will reset the interrupt condition on the interrupt controller upon successful return from the ISR later on.

Reading the receiver buffer, confirms and resets the interrupt condition on the serial interface controller.

```
if (IntId == IIR_RECEIVER_FULL)
{
   UCHAR rcvd;
   rcvd = READ_RECEIVER_BUFFER(pSmplDeviceContext>pPortAddress);
}
```

➔ Step 5:
EvtInterruptIsr loop implementation checking for interrupt condition

The above logic needs to be implemented in a loop.

```
BOOLEAN bReturnValue = FALSE;

#pragma warning(push)
#pragma warning(disable:4127) // conditional expression is constant
    while(TRUE)
#pragma warning(pop)
    {   // First check: Is it ours???
        UCHAR rcvd;

        UCHAR IntId = READ_INTERRUPT_IDENTIFICATION(
                                    pSmplDeviceContext->pPortAddress);
        if (IntId & IIR_NOT_FIRED)
        {
            return bReturnValue;
        }

        DbgPrintEx(DPFLTR_IHVDRIVER_ID, 1234,"Interrupt\n");
        bReturnValue = TRUE;
...
```

This loop is necessary for the serial port on VMWare (real serial ports or serial ports on Microsoft Virtual PC do not need this loop).

➔ Step 6:
Enabling and disabling device interrupts

The interrupt is enabled and disabled by writing to a specific hardware register in `SmplDeviceInterruptEnable(…)` and `SmplDeviceInterruptDisable(…)`. For this purpose, the file Hw8250.h contains a macro encapsulating the call to `WRITE_PORT_UCHAR(…)`. Interrupts are enabled by calling:

```
WRITE_INTERRUPT_ENABLE(  pSmplDeviceContext->pPortAddress,
                         IER_RECEIVER_FULL       );
```

The device interrupt is disabled by invoking:

```
WRITE_INTERRUPT_ENABLE(pSmplDeviceContext->pPortAddress, 0);
```

➔ Step 7:
Test

Add a named pipe on the virtual machine on the second virtual serial port (COM2). Name the pipe “\\.\pipe\vpcserial“. A simple application sending random one-byte data at a time through the pipe can be created.

`DbgPrintEx(...)` can be used within the interrupt service routine to verify in the debugger if and when it is invoked.

S0 idle power management needs to be disabled in Device Manager's property sheet.

4 Simple Application IO with Hardware

Overview

This lab demonstrates basic (yet incomplete) 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 serial port is received from within the interrupt service routine. WdfRequests cannot be completed within the interrupt service routine. Windows interrupt architecture requires a DPC (Deferred Procedure Call) to be queued for doing IO operations like e.g. completion of WdfRequests. Such IO operations can only be done at or below `IRQL_DSPATCH_LEVEL`.

From within the DPC 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:	SmplDeviceKmdf\SmplDeviceKmdfLab4
Own new lab directory to create:	SmplDeviceKmdf\SmplDeviceKmdfLab5
The following files are involved:	Driver.c Driver.h Device.c Interrupt.c Queue.c Hw8250.h Hw8250.c
The following files are new:	-

4.1 Creating the Deferred Procedure Calls (DPC) routine callback

In this lab section, the driver shall be extended by a DPC routine for interrupt post processing. The first DPC routine implementation shall still be empty without any functionality. From within the ISR it will be scheduled for later execution. With the help of the kernel debugger, it can be evaluated if and when the DPC will be invoked.

Applied KMDF methods:

`WdfInterruptQueueDpcForIsr(…)`

Applied KMDF event callbacks:

`EVT_WDF_INTERRUPT_DPC`

➔ Step 1: Declaration and implementation of a DPC event handler

First an empty DPC callback shall be defined and implemented. It should only contain debug traces and have the following prototype:

```
EVT_WDF_INTERRUPT_DPC SmplInterruptEvtDpc;
```

➔ Step 2: Subscription for the DPC event callback routine

DPC event callback routine can be subscribed upon Interrupt object creation in `EvtDriverDeviceAdd`. The `SmplInterruptEvtDpc` callback shall be assigned to the `WDF_INTERRUPT_CONFIG` structure using the macro `WDF_INTERRUPT_CONFIG_INIT(...)`. Since there is already an existing call to this macro, just add the callback for DPC next to the ISR callback parameter. It should look like this:

```
WDF_INTERRUPT_CONFIG_INIT(  &interruptConfig,
                            SmplInterruptEvtIsr,
                            SmplInterruptEvtDpc);
```

➔ Step 3: Triggering a DPC from the interrupt service routine

From within the interrupt service routine, a DPC has to be triggered. This is done via the following function: `WdfInterruptQueueDpcForIsr`(…)

This function should be called right after receiving the data in the ISR.

➔ Step 4: Checking the invocation of the DPC routine

With the kernel debugger, it can be verified if and when the DPC routine is invoked.

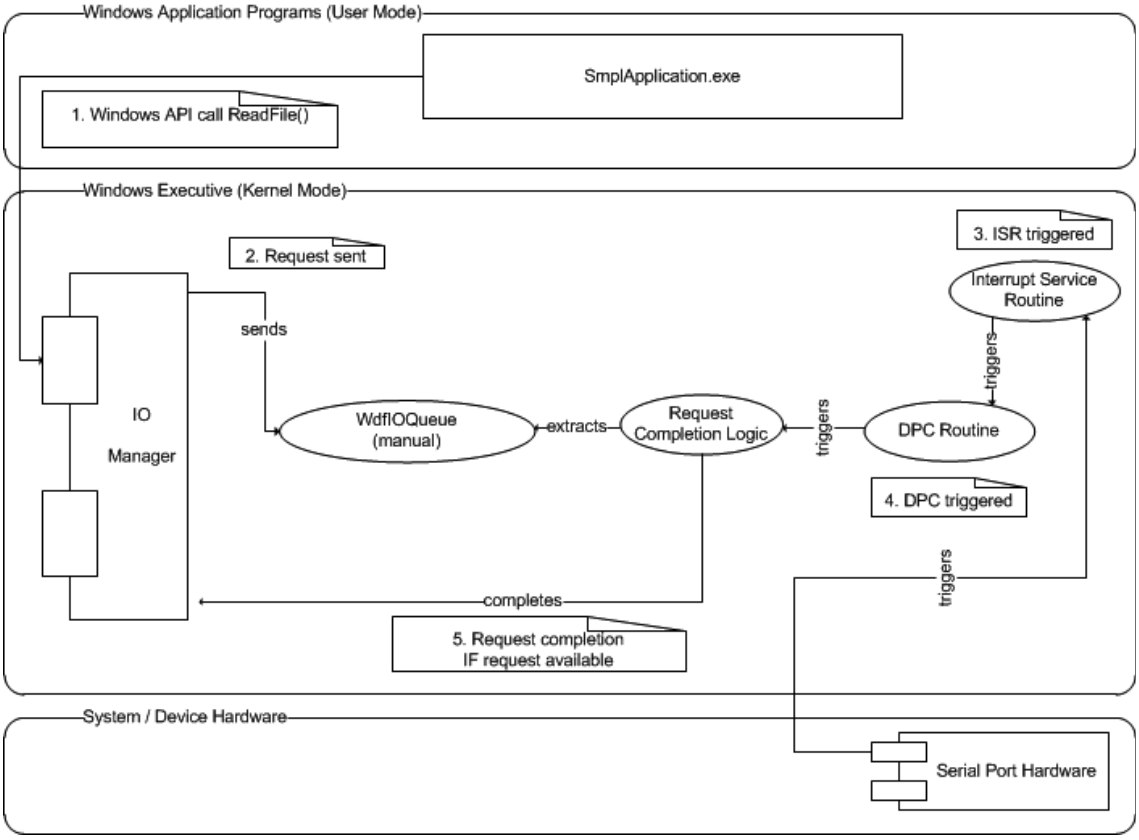
4.2 Simple Request completion upon hardware action

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

Applied KMDF methods:

`WdfIoQueueRetrieveNextRequest(…)`
`WdfInterruptGetDevice(…)`

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:
Retrieving Requests from the queue in DPC callback function

After receiving the data from the hardware in the ISR, the DPC callback function is invoked to process the data and complete an available pending Request from the queue. The 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 4: Running the user mode application for testing

The test application created in the previous exercise can be used again to test this driver. Whenever the data simulator writes data to the pipe (\\.\Pipe\vpcserial), the test application's `ReadFile(...)` should be completed. Multiple applications shall be tested concurrently at the same time.

5 Data Buffering and Synchronization

Overview

This lab implements data buffering and synchronization. It teaches how to transfer device input data from the Interrupt Service Routine via a DPC routine to the user mode application. To ensure correct operation, the DPC routine has to be synchronized against the Interrupt Service Routine when moving data between them. This is done by using interrupt Spinlocks.

Goal

Implementation of a Windows device driver which an incoming data from the COM port is placed into a circular buffer and transferring data to one or more Win32 applications at the same time.

Setup

The template is located in the following directory:	SmplDeviceKmdf\SmplDeviceKmdfLab5
Own new lab directory to create:	SmplDeviceKmdf\SmplDeviceKmdfLab6
The following files are involved:	Driver.c Driver.h Device.c Interrupt.c Queue.c Hw8250.h Hw8250.c
The following files are new:	CircularBuffer.c

5.1 Implementation of a circular buffer

The goal of this exercise is to be able to implement an intermediate buffering of incoming serial data.

Applied KMDF methods:

`WdfDeviceInitRegisterPnpStateChangeCallback(...)`

Applied KMDF event callbacks:

`EVT_WDF_DEVICE_PNP_STATE_CHANGE_NOTIFICATION`

➔ Step 1: Implementation of a circular buffer

The DPC routine is triggered after one or more Interrupts have already occurred. Every data received in an interrupt should be processed one at a time with a corresponding Read Request. To avoid loss of data, a circular buffer shall be implemented. This buffer will hold the input from the device until it is ready to be processed.

A circular buffer shall be implemented in a separate file named CircularBuffer.c. The circular buffer shall encapsulate the private data structure `SMPLDEVICE_RINGBUFFER`:

```
typedef struct _SMPLDEVICE_RINGBUFFER
{         ULONG MaxSize;
          ULONG FreeSpace;
          ULONG ReadPos;
          ULONG WritePos;
          UCHAR Buffer[1];

} SMPLDEVICE_RINGBUFFER,*PSMPLDEVICE_RINGBUFFER;
```

The following public functions shall be implemented by the ringbuffer:

```
PSMPLDEVICE_RINGBUFFER RingbufferInitialize(ULONG BufSize)

VOID
RingbufferFree(PSMPLDEVICE_RINGBUFFER pRingBuffer);

PSMPLDEVICE_RINGBUFFER RingbufferInitialize(IN ULONG BufSize);

BOOLEAN
RingbufferInsertCharacter(
          PSMPLDEVICE_RINGBUFFER pRingBuffer,
          IN UCHAR Character);

BOOLEAN
RingbufferRemoveCharacter(
          PSMPLDEVICE_RINGBUFFER pRingBuffer,
          OUT PUCHAR pCharacter);

ULONG
RingbufferGetCountCharacters(
          PSMPLDEVICE_RINGBUFFER pRingBuffer);
```

To save time, please ask your trainer for a ready to run implementation of this circular buffer.

➔ Step 2: Ringbuffer allocation, initialization and cleanup

In the `EvtDriverDeviceAdd` callback function a circular buffer has to be allocated and initialized.

For circular buffer deallocation a new Plug'n'Play callback of type `EvtDevicePnpStateChange` has to be registered for `WdfDevStatePnpRemoved` notification type `StateNotificationEnterState`. Registration is done by calling

`WdfDeviceInitRegisterPnpStateChangeCallback` in `EvtDriverDeviceAdd` before the device object is created.

Inside this newly declared and registered Plug'n'Play callback function the circular buffer has to be freed.

5.2 Circular buffer synchronization between ISR and DPC

In this lab section, the previously implemented circular buffer will be integrated into the driver. The access to the incoming data buffer shall be synchronized between the DPC routine and the Interrupt Service Routine. This synchronization is achieved by using Interrupt Spinlocks.

Applied KMDF methods:

```
WdfInterruptAcquireLock(…)
WdfInterruptReleaseLock(…)
```

➔ Step 1: Circular buffer operation

Every data received in the `EvtInterruptIsr` callback function should immediately be inserted into the circular buffer. This data shall be removed later from within the DPC routine. At this stage the extracted data shall not be processed yet after being removed from the circular buffer.

Request processing shall remain in place exactly as it was implemented before in the previous chapter.

➔ Step 2: Acquiring and releasing the interrupt lock in the DPC

To prevent the DPC routine from concurrently modifying the circular buffer at the same time as the Interrupt Service Routine (ISR), an Interrupt lock must be acquired by the DPC This can be implemented using:

```
WdfInterruptAcquireLock(…)
WdfInterruptReleaseLock(…)
```

5.3 Partial synchronization of Request processing between queue and DPC

If there is a Request available and data is present in the circular buffer, then this Request shall be completed. To achieve this, the DPC has to be synchronized against itself and against the WdfIoQueue.

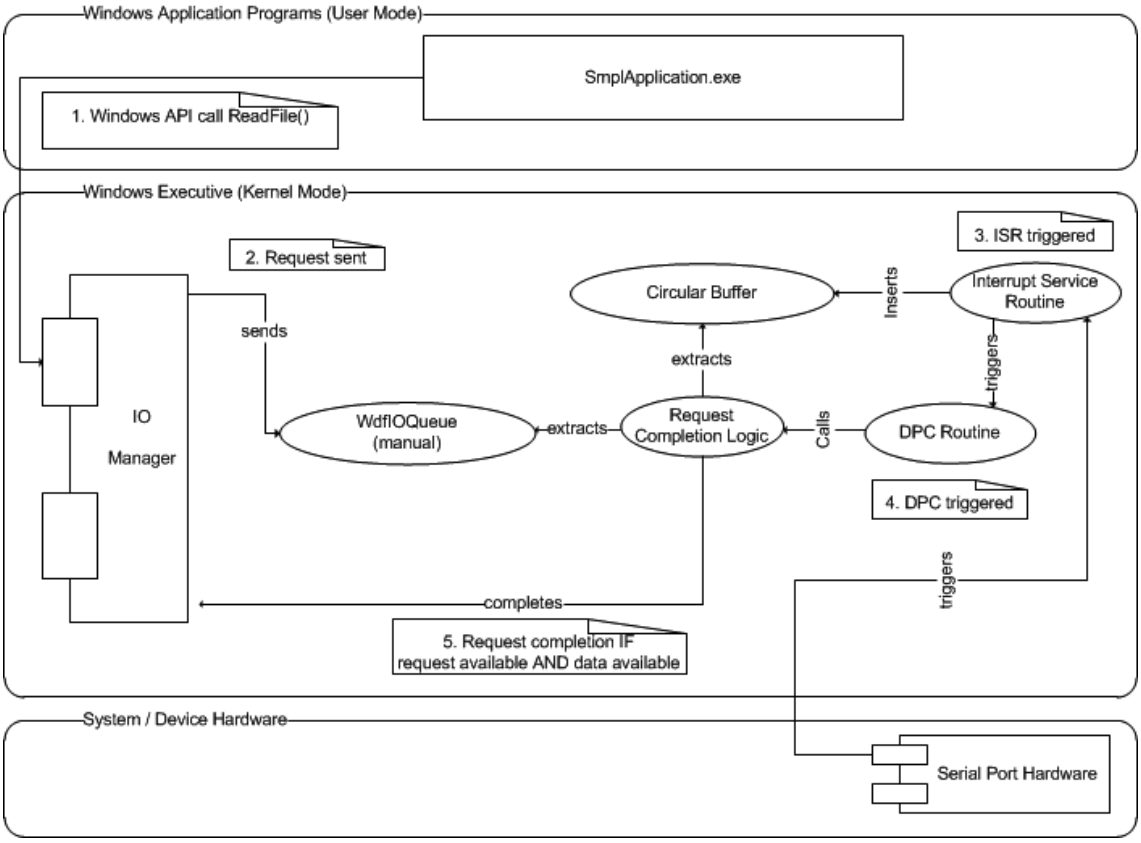
Applied KMDF methods:

```
WdfObjectAcquireLock(…)
WdfObjectReleaseLock(…)
```

Applied framework data structures in the driver:

```
WDF_OBJECT_ATTRIBUTES
```

Overview diagram:



➔ Step 1:
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 2:
Request completion synchronized by the queue object lock

The DPC now has to check for two conditions to be able to do Request processing:
- Are there Requests in the queue?
- Is data available in the circular buffer?

The check for both conditions has to be done in a synchronized way. Explicitly acquiring the Read Queue object lock shall implicitly prevent both of the above conditions from changing.

The following completion logic needs to be implemented:
- Acquire Queue lock
- Acquire Interrupt lock
- Peek circular buffer if data available
- Release Interrupt lock

- IF no data, release Queue lock => return
- Retrieve next Request from Queue
- IF no request, then release Queue lock => return
- Acquire Interrupt lock
- Copy data from circular buffer to Request
- Release Interrupt lock
- Complete Request
- Release Queue lock

5.4 Synchronized Request processing (final)

Request processing will be implemented. Requests shall only be completed if there is data available in the circular buffer. To achieve this the DPC has to be synchronized against itself and the WdfIoQueue.

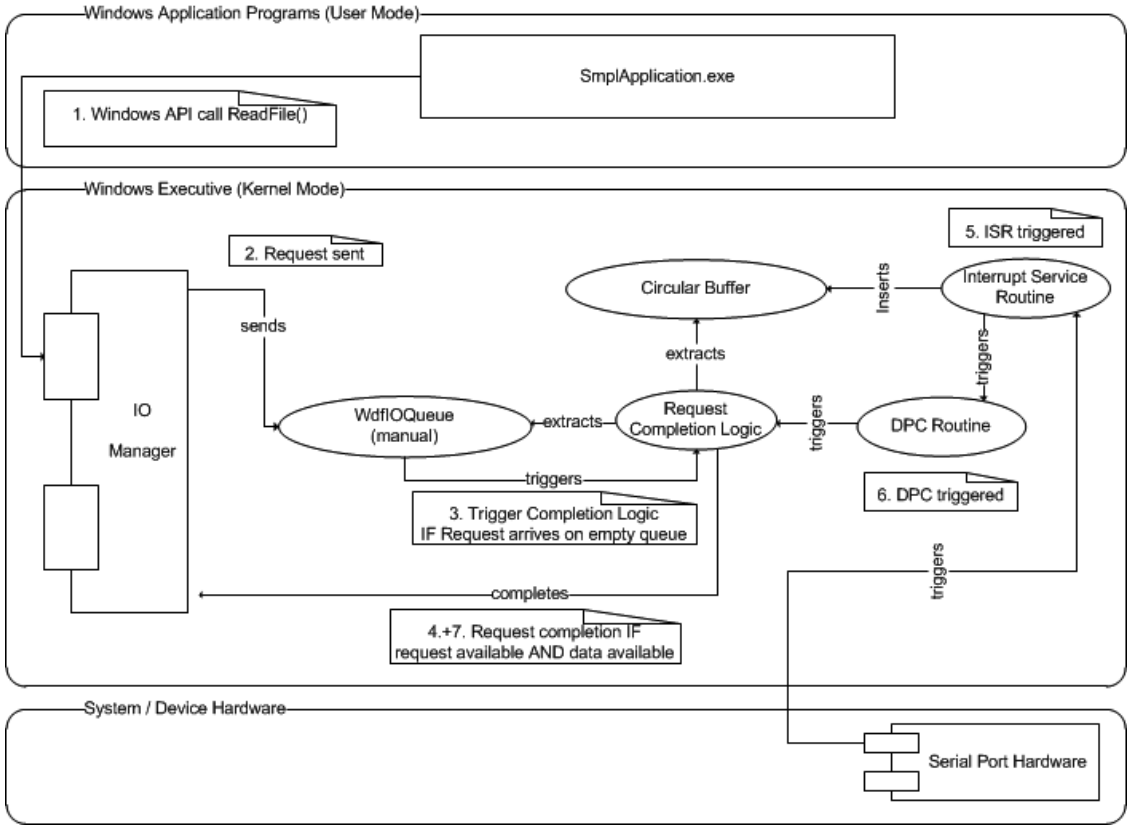
Applied KMDF methods:

`WdfIoQueueReadyNotify`

Applied KMDF event callbacks:

`EVT_WDF_IO_QUEUE_STATE`

Overview diagram:



➔ Step 1: Extending the Device Object Context Area structure

First the device object context structure has to be extended. A new variable is needed to hold the interrupt object.

```
typedef struct _SMPL_DEVICE_CONTEXT
{
.
   WDFINTERRUPT SmplInterrupt;
.
```

```
} SMPL_DEVICE_CONTEXT, *PSMPL_DEVICE_CONTEXT;
```

This new variable has to be initialized upon creation of the interrupt object.

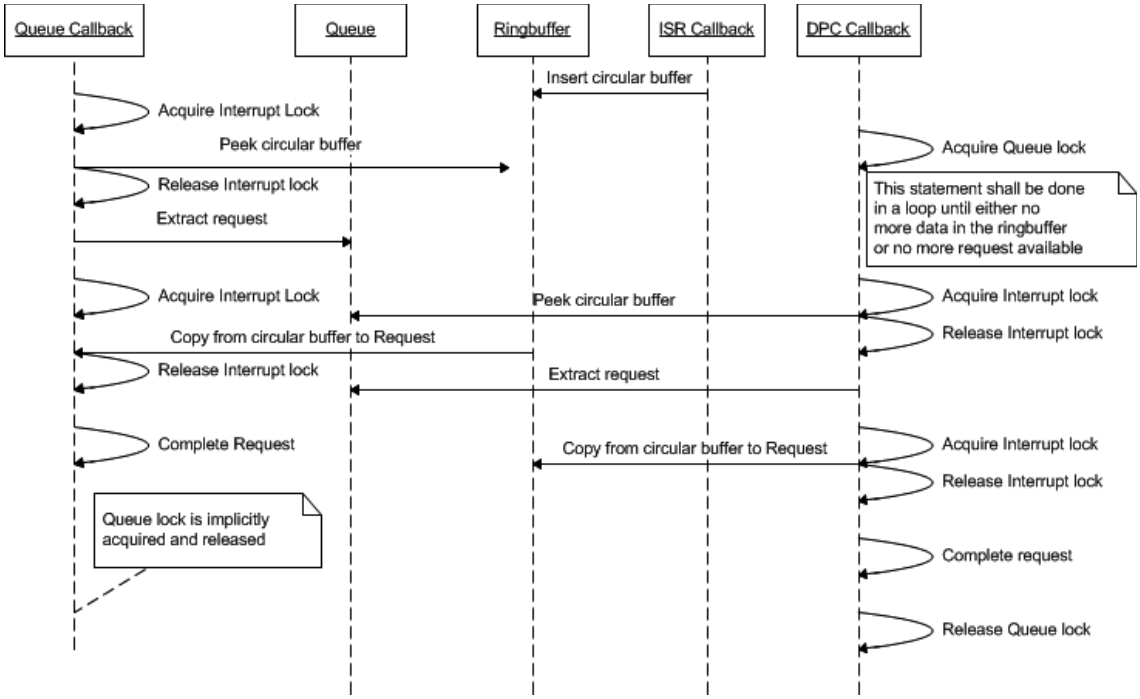
➔ Step 2: Implementing a separate function for Request completion

The code that loops checking ringbuffer and completing Requests shall be moved from the `EvtInterruptDpc` callback to a separate function. This is necessary for reusing it from other places later.

➔ Step 3: EVT_WDF_IO_QUEUE_STATE implementation

Since the `EvtIoRead` has already been removed in the previous lab, a new callback function shall be created to be invoked whenever the Read Queue changes state from empty to non-empty. It can be done by calling `WdfIoQueueReadyNotify(…)`. Name the callback function `SmplQueueEvtState(…)`. Here, available Read Request can be completed right away if the circular buffer is not empty.

Below is a UML Sequence Diagram for the implementation of the synchronization:



5.5 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.

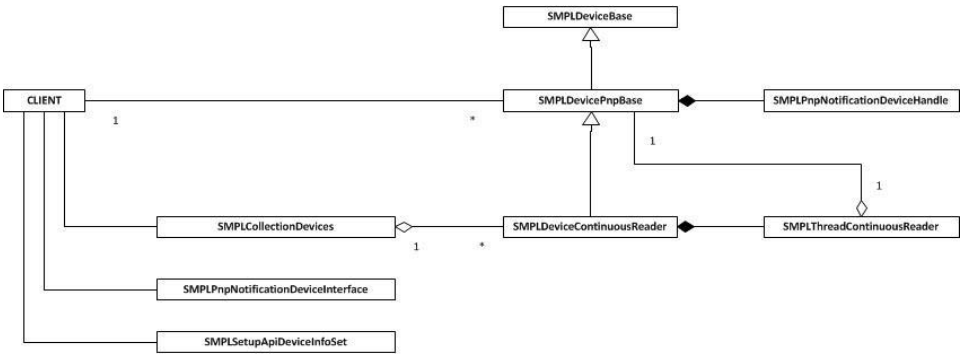
➔ Step 2:
Testing Device Handle notification for DBT_DEVICEQUERYREMOVE

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

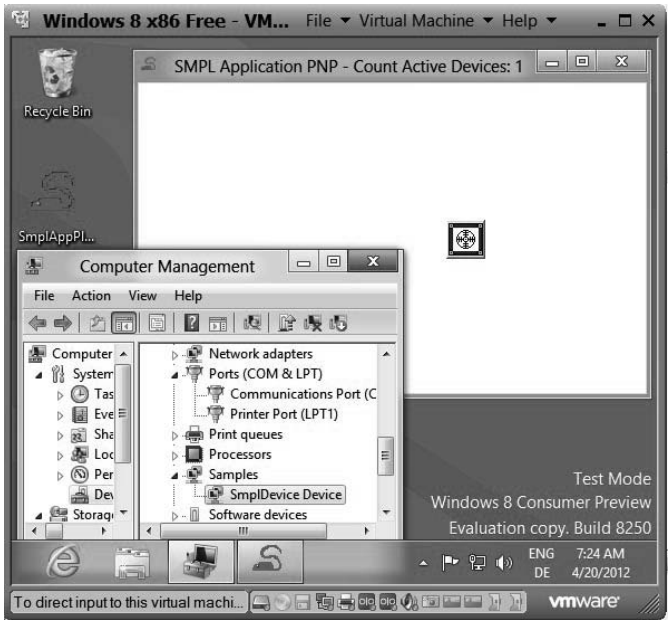
From within the handler for DBT_DEVICEARRIVAL the application needs to open a handle to the device. Device Interface symbolic link name is a parameter of the notification.

➔ 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 Appendix

6.1 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)
```

6.2 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\SmplDevice.sys
C:\TrainingWdfKmdf\SmplDeviceKmdfLabX\objchk_win7_x86\SmplDevice.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
```

6.3 WinDbg WMI trace output

➔ **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 SmplDevice.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\SmplDevice\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 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.