1. Filtering Requests of the Keyboard Driver

# 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

The goal is writing a Keyboard Filter driver which can trace and/or modified each keystroke.

# 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

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 SmplFltr.

Remove public.h and all device interface related functionality. Temporarily disable WDFIOQUEUE object creation by commenting the call to SmplFltrQueueInitialize 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

* Remove the Class GUID, and the class guid sections
* Change the ClassGUID and class to keyboard. Take a hint from the original keyboard.inf file, on INF database (windows\inf)
* Change the hardware ID:

[Standard.NT$ARCH$]

;%SmplFltr.DeviceDesc%=SmplFltr\_Device, Root\SmplFltr ; TODO: edit hw-id

%SmplFltr.DeviceDesc%=SmplFltr\_Device, \*PNP0303

## 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):



Verify keyboard is still working

# 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.

# Completion routine for WdfRequestTypeRead

The driver has to subscribe for a completion routine to receive WDFREQUESTs 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.

# 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.