1. Lab 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: kmdf\lab1

New lab directory to create: kmdf\lab2

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: kmdf\lab2

The following files are involved:

The following files are new: SmplApplicationConsole.cpp

# 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

CloseHandle

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”. It’s 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.

# 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 includes 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

# 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.

# 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.