**ПРИЛОЖЕНИЕ Б**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

# КАФЕДРА ИНТЕЛЛЕКТУАЛЬНЫХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

**ДРАЙВЕР ВИРТУАЛЬНОГО МОНИТОРА ДЛЯ ОС WINDOWS**

**КОД ПРОГРАММЫ**

**КП.ПО-9.1-40-01-01**

## Листов 19

|  |  |
| --- | --- |
|  |  |
| **Руководитель** | И. Д. Козик |
| **Выполнил** | А. С. Мисиюк |
| **Консультант** |  |
| **по ЕСПД** | И. Д. Козик |
|  |  |
|  |  |

**2023**

**Driver.h**

#pragma once

#define NOMINMAX

#include <windows.h>

#include <bugcodes.h>

#include <wudfwdm.h>

#include <wdf.h>

#include <iddcx.h>

#include <dxgi1\_5.h>

#include <d3d11\_2.h>

#include <avrt.h>

#include <wrl.h>

#include <memory>

#include <vector>

#include "Trace.h"

namespace Microsoft

{

namespace WRL

{

namespace Wrappers

{

// Adds a wrapper for thread handles to the existing set of WRL handle wrapper classes

typedef HandleT<HandleTraits::HANDLENullTraits> Thread;

}

}

}

namespace Microsoft

{

namespace IndirectDisp

{

/// <summary>

/// Manages the creation and lifetime of a Direct3D render device.

/// </summary>

struct Direct3DDevice

{

Direct3DDevice(LUID AdapterLuid);

Direct3DDevice();

HRESULT Init();

LUID AdapterLuid;

Microsoft::WRL::ComPtr<IDXGIFactory5> DxgiFactory;

Microsoft::WRL::ComPtr<IDXGIAdapter1> Adapter;

Microsoft::WRL::ComPtr<ID3D11Device> Device;

Microsoft::WRL::ComPtr<ID3D11DeviceContext> DeviceContext;

};

/// <summary>

/// Manages a thread that consumes buffers from an indirect display swap-chain object.

/// </summary>

class SwapChainProcessor

{

public:

SwapChainProcessor(IDDCX\_SWAPCHAIN hSwapChain, std::shared\_ptr<Direct3DDevice> Device, HANDLE NewFrameEvent);

~SwapChainProcessor();

private:

static DWORD CALLBACK RunThread(LPVOID Argument);

void Run();

void RunCore();

public:

IDDCX\_SWAPCHAIN m\_hSwapChain;

std::shared\_ptr<Direct3DDevice> m\_Device;

HANDLE m\_hAvailableBufferEvent;

Microsoft::WRL::Wrappers::Thread m\_hThread;

Microsoft::WRL::Wrappers::Event m\_hTerminateEvent;

};

/// <summary>

/// Provides a sample implementation of an indirect display driver.

/// </summary>

class IndirectDeviceContext

{

public:

IndirectDeviceContext(\_In\_ WDFDEVICE WdfDevice);

virtual ~IndirectDeviceContext();

void InitAdapter();

void FinishInit();

void CreateMonitor(unsigned int index);

void AssignSwapChain(IDDCX\_SWAPCHAIN SwapChain, LUID RenderAdapter, HANDLE NewFrameEvent);

void UnassignSwapChain();

protected:

WDFDEVICE m\_WdfDevice;

IDDCX\_ADAPTER m\_Adapter;

IDDCX\_MONITOR m\_Monitor;

IDDCX\_MONITOR m\_Monitor2;

std::unique\_ptr<SwapChainProcessor> m\_ProcessingThread;

public:

static const DISPLAYCONFIG\_VIDEO\_SIGNAL\_INFO s\_KnownMonitorModes[];

static const BYTE s\_KnownMonitorEdid[];

};

}

}

**Trace.h**

/\*++

Module Name:

Internal.h

Abstract:

This module contains the local type definitions for the

driver.

Environment:

Windows User-Mode Driver Framework 2

--\*/

//

// Define the tracing flags.

//

// Tracing GUID - b254994f-46e6-4718-80a0-0a3aa50d6ce4

//

#define WPP\_CONTROL\_GUIDS \

WPP\_DEFINE\_CONTROL\_GUID( \

MyDriver1TraceGuid, (b254994f,46e6,4718,80a0,0a3aa50d6ce4), \

\

WPP\_DEFINE\_BIT(MYDRIVER\_ALL\_INFO) \

WPP\_DEFINE\_BIT(TRACE\_DRIVER) \

WPP\_DEFINE\_BIT(TRACE\_DEVICE) \

WPP\_DEFINE\_BIT(TRACE\_QUEUE) \

)

#define WPP\_FLAG\_LEVEL\_LOGGER(flag, level) \

WPP\_LEVEL\_LOGGER(flag)

#define WPP\_FLAG\_LEVEL\_ENABLED(flag, level) \

(WPP\_LEVEL\_ENABLED(flag) && \

WPP\_CONTROL(WPP\_BIT\_ ## flag).Level >= level)

#define WPP\_LEVEL\_FLAGS\_LOGGER(lvl,flags) \

WPP\_LEVEL\_LOGGER(flags)

#define WPP\_LEVEL\_FLAGS\_ENABLED(lvl, flags) \

(WPP\_LEVEL\_ENABLED(flags) && WPP\_CONTROL(WPP\_BIT\_ ## flags).Level >= lvl)

//

// This comment block is scanned by the trace preprocessor to define our

// Trace function.

//

// begin\_wpp config

// FUNC Trace{FLAG=MYDRIVER\_ALL\_INFO}(LEVEL, MSG, ...);

// FUNC TraceEvents(LEVEL, FLAGS, MSG, ...);

// end\_wpp

//

//

// Driver specific #defines

//

// Use a unique driver tracing ID here,

// see https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/adding-wpp-software-tracing-to-a-windows-driver

#define MYDRIVER\_TRACING\_ID L"Microsoft\\UMDF2.25\\IddSampleDriver v1.0"

**Driver.cpp**

/\*++

Copyright (c) Microsoft Corporation

Abstract:

This module contains a sample implementation of an indirect display driver. See the included README.md file and the

various TODO blocks throughout this file and all accompanying files for information on building a production driver.

MSDN documentation on indirect displays can be found at https://msdn.microsoft.com/en-us/library/windows/hardware/mt761968(v=vs.85).aspx.

Environment:

User Mode, UMDF

--\*/

#include "Driver.h"

#include "Driver.tmh"

using namespace std;

using namespace Microsoft::IndirectDisp;

using namespace Microsoft::WRL;

extern "C" DRIVER\_INITIALIZE DriverEntry;

EVT\_WDF\_DRIVER\_DEVICE\_ADD IddSampleDeviceAdd;

EVT\_WDF\_DEVICE\_D0\_ENTRY IddSampleDeviceD0Entry;

EVT\_IDD\_CX\_ADAPTER\_INIT\_FINISHED IddSampleAdapterInitFinished;

EVT\_IDD\_CX\_ADAPTER\_COMMIT\_MODES IddSampleAdapterCommitModes;

EVT\_IDD\_CX\_PARSE\_MONITOR\_DESCRIPTION IddSampleParseMonitorDescription;

EVT\_IDD\_CX\_MONITOR\_GET\_DEFAULT\_DESCRIPTION\_MODES IddSampleMonitorGetDefaultModes;

EVT\_IDD\_CX\_MONITOR\_QUERY\_TARGET\_MODES IddSampleMonitorQueryModes;

EVT\_IDD\_CX\_MONITOR\_ASSIGN\_SWAPCHAIN IddSampleMonitorAssignSwapChain;

EVT\_IDD\_CX\_MONITOR\_UNASSIGN\_SWAPCHAIN IddSampleMonitorUnassignSwapChain;

struct IndirectDeviceContextWrapper

{

IndirectDeviceContext\* pContext;

void Cleanup()

{

delete pContext;

pContext = nullptr;

}

};

// This macro creates the methods for accessing an IndirectDeviceContextWrapper as a context for a WDF object

WDF\_DECLARE\_CONTEXT\_TYPE(IndirectDeviceContextWrapper);

extern "C" BOOL WINAPI DllMain(

\_In\_ HINSTANCE hInstance,

\_In\_ UINT dwReason,

\_In\_opt\_ LPVOID lpReserved)

{

UNREFERENCED\_PARAMETER(hInstance);

UNREFERENCED\_PARAMETER(lpReserved);

UNREFERENCED\_PARAMETER(dwReason);

return TRUE;

}

\_Use\_decl\_annotations\_

extern "C" NTSTATUS DriverEntry(

PDRIVER\_OBJECT pDriverObject,

PUNICODE\_STRING pRegistryPath

)

{

WDF\_DRIVER\_CONFIG Config;

NTSTATUS Status;

WDF\_OBJECT\_ATTRIBUTES Attributes;

WDF\_OBJECT\_ATTRIBUTES\_INIT(&Attributes);

WDF\_DRIVER\_CONFIG\_INIT(&Config,

IddSampleDeviceAdd

);

Status = WdfDriverCreate(pDriverObject, pRegistryPath, &Attributes, &Config, WDF\_NO\_HANDLE);

if (!NT\_SUCCESS(Status))

{

return Status;

}

return Status;

}

\_Use\_decl\_annotations\_

NTSTATUS IddSampleDeviceAdd(WDFDRIVER Driver, PWDFDEVICE\_INIT pDeviceInit)

{

NTSTATUS Status = STATUS\_SUCCESS;

WDF\_PNPPOWER\_EVENT\_CALLBACKS PnpPowerCallbacks;

UNREFERENCED\_PARAMETER(Driver);

// Register for power callbacks - in this sample only power-on is needed

WDF\_PNPPOWER\_EVENT\_CALLBACKS\_INIT(&PnpPowerCallbacks);

PnpPowerCallbacks.EvtDeviceD0Entry = IddSampleDeviceD0Entry;

WdfDeviceInitSetPnpPowerEventCallbacks(pDeviceInit, &PnpPowerCallbacks);

IDD\_CX\_CLIENT\_CONFIG IddConfig;

IDD\_CX\_CLIENT\_CONFIG\_INIT(&IddConfig);

// If the driver wishes to handle custom IoDeviceControl requests, it's necessary to use this callback since IddCx

// redirects IoDeviceControl requests to an internal queue. This sample does not need this.

// IddConfig.EvtIddCxDeviceIoControl = IddSampleIoDeviceControl;

IddConfig.EvtIddCxAdapterInitFinished = IddSampleAdapterInitFinished;

IddConfig.EvtIddCxParseMonitorDescription = IddSampleParseMonitorDescription;

IddConfig.EvtIddCxMonitorGetDefaultDescriptionModes = IddSampleMonitorGetDefaultModes;

IddConfig.EvtIddCxMonitorQueryTargetModes = IddSampleMonitorQueryModes;

IddConfig.EvtIddCxAdapterCommitModes = IddSampleAdapterCommitModes;

IddConfig.EvtIddCxMonitorAssignSwapChain = IddSampleMonitorAssignSwapChain;

IddConfig.EvtIddCxMonitorUnassignSwapChain = IddSampleMonitorUnassignSwapChain;

Status = IddCxDeviceInitConfig(pDeviceInit, &IddConfig);

if (!NT\_SUCCESS(Status))

{

return Status;

}

WDF\_OBJECT\_ATTRIBUTES Attr;

WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE(&Attr, IndirectDeviceContextWrapper);

Attr.EvtCleanupCallback = [](WDFOBJECT Object)

{

// Automatically cleanup the context when the WDF object is about to be deleted

auto\* pContext = WdfObjectGet\_IndirectDeviceContextWrapper(Object);

if (pContext)

{

pContext->Cleanup();

}

};

WDFDEVICE Device = nullptr;

Status = WdfDeviceCreate(&pDeviceInit, &Attr, &Device);

if (!NT\_SUCCESS(Status))

{

return Status;

}

Status = IddCxDeviceInitialize(Device);

// Create a new device context object and attach it to the WDF device object

auto\* pContext = WdfObjectGet\_IndirectDeviceContextWrapper(Device);

pContext->pContext = new IndirectDeviceContext(Device);

return Status;

}

\_Use\_decl\_annotations\_

NTSTATUS IddSampleDeviceD0Entry(WDFDEVICE Device, WDF\_POWER\_DEVICE\_STATE PreviousState)

{

UNREFERENCED\_PARAMETER(PreviousState);

// This function is called by WDF to start the device in the fully-on power state.

auto\* pContext = WdfObjectGet\_IndirectDeviceContextWrapper(Device);

pContext->pContext->InitAdapter();

return STATUS\_SUCCESS;

}

#pragma region Direct3DDevice

Direct3DDevice::Direct3DDevice(LUID AdapterLuid) : AdapterLuid(AdapterLuid)

{

}

Direct3DDevice::Direct3DDevice()

{

AdapterLuid = LUID{};

}

HRESULT Direct3DDevice::Init()

{

// The DXGI factory could be cached, but if a new render adapter appears on the system, a new factory needs to be

// created. If caching is desired, check DxgiFactory->IsCurrent() each time and recreate the factory if !IsCurrent.

HRESULT hr = CreateDXGIFactory2(0, IID\_PPV\_ARGS(&DxgiFactory));

if (FAILED(hr))

{

return hr;

}

// Find the specified render adapter

hr = DxgiFactory->EnumAdapterByLuid(AdapterLuid, IID\_PPV\_ARGS(&Adapter));

if (FAILED(hr))

{

return hr;

}

// Create a D3D device using the render adapter. BGRA support is required by the WHQL test suite.

hr = D3D11CreateDevice(Adapter.Get(), D3D\_DRIVER\_TYPE\_UNKNOWN, nullptr, D3D11\_CREATE\_DEVICE\_BGRA\_SUPPORT, nullptr, 0, D3D11\_SDK\_VERSION, &Device, nullptr, &DeviceContext);

if (FAILED(hr))

{

// If creating the D3D device failed, it's possible the render GPU was lost (e.g. detachable GPU) or else the

// system is in a transient state.

return hr;

}

return S\_OK;

}

#pragma endregion

#pragma region SwapChainProcessor

SwapChainProcessor::SwapChainProcessor(IDDCX\_SWAPCHAIN hSwapChain, shared\_ptr<Direct3DDevice> Device, HANDLE NewFrameEvent)

: m\_hSwapChain(hSwapChain), m\_Device(Device), m\_hAvailableBufferEvent(NewFrameEvent)

{

m\_hTerminateEvent.Attach(CreateEvent(nullptr, FALSE, FALSE, nullptr));

// Immediately create and run the swap-chain processing thread, passing 'this' as the thread parameter

m\_hThread.Attach(CreateThread(nullptr, 0, RunThread, this, 0, nullptr));

}

SwapChainProcessor::~SwapChainProcessor()

{

// Alert the swap-chain processing thread to terminate

SetEvent(m\_hTerminateEvent.Get());

if (m\_hThread.Get())

{

// Wait for the thread to terminate

WaitForSingleObject(m\_hThread.Get(), INFINITE);

}

}

DWORD CALLBACK SwapChainProcessor::RunThread(LPVOID Argument)

{

reinterpret\_cast<SwapChainProcessor\*>(Argument)->Run();

return 0;

}

void SwapChainProcessor::Run()

{

// For improved performance, make use of the Multimedia Class Scheduler Service, which will intelligently

// prioritize this thread for improved throughput in high CPU-load scenarios.

DWORD AvTask = 0;

HANDLE AvTaskHandle = AvSetMmThreadCharacteristicsW(L"Distribution", &AvTask);

RunCore();

// Always delete the swap-chain object when swap-chain processing loop terminates in order to kick the system to

// provide a new swap-chain if necessary.

WdfObjectDelete((WDFOBJECT)m\_hSwapChain);

m\_hSwapChain = nullptr;

AvRevertMmThreadCharacteristics(AvTaskHandle);

}

void SwapChainProcessor::RunCore()

{

// Get the DXGI device interface

ComPtr<IDXGIDevice> DxgiDevice;

HRESULT hr = m\_Device->Device.As(&DxgiDevice);

if (FAILED(hr))

{

return;

}

IDARG\_IN\_SWAPCHAINSETDEVICE SetDevice = {};

SetDevice.pDevice = DxgiDevice.Get();

hr = IddCxSwapChainSetDevice(m\_hSwapChain, &SetDevice);

if (FAILED(hr))

{

return;

}

// Acquire and release buffers in a loop

for (;;)

{

ComPtr<IDXGIResource> AcquiredBuffer;

// Ask for the next buffer from the producer

IDARG\_OUT\_RELEASEANDACQUIREBUFFER Buffer = {};

hr = IddCxSwapChainReleaseAndAcquireBuffer(m\_hSwapChain, &Buffer);

// AcquireBuffer immediately returns STATUS\_PENDING if no buffer is yet available

if (hr == E\_PENDING)

{

// We must wait for a new buffer

HANDLE WaitHandles [] =

{

m\_hAvailableBufferEvent,

m\_hTerminateEvent.Get()

};

DWORD WaitResult = WaitForMultipleObjects(ARRAYSIZE(WaitHandles), WaitHandles, FALSE, 16);

if (WaitResult == WAIT\_OBJECT\_0 || WaitResult == WAIT\_TIMEOUT)

{

// We have a new buffer, so try the AcquireBuffer again

continue;

}

else if (WaitResult == WAIT\_OBJECT\_0 + 1)

{

// We need to terminate

break;

}

else

{

// The wait was cancelled or something unexpected happened

hr = HRESULT\_FROM\_WIN32(WaitResult);

break;

}

}

else if (SUCCEEDED(hr))

{

AcquiredBuffer.Attach(Buffer.MetaData.pSurface);

// ==============================

// Process the frame here

//

// This is the most performance-critical section of code in an IddCx driver. It's important that whatever

// is done with the acquired surface be finished as quickly as possible. This operation could be:

// \* a GPU copy to another buffer surface for later processing (such as a staging surface for mapping to CPU memory)

// \* a GPU encode operation

// \* a GPU VPBlt to another surface

// \* a GPU custom compute shader encode operation

// ==============================

AcquiredBuffer.Reset();

hr = IddCxSwapChainFinishedProcessingFrame(m\_hSwapChain);

if (FAILED(hr))

{

break;

}

// ==============================

// Report frame statistics once the asynchronous encode/send work is completed

//

// Drivers should report information about sub-frame timings, like encode time, send time, etc.

// ==============================

// IddCxSwapChainReportFrameStatistics(m\_hSwapChain, ...);

}

else

{

// The swap-chain was likely abandoned (e.g. DXGI\_ERROR\_ACCESS\_LOST), so exit the processing loop

break;

}

}

}

#pragma endregion

#pragma region IndirectDeviceContext

const UINT64 MHZ = 1000000;

const UINT64 KHZ = 1000;

// A list of modes exposed by the sample monitor EDID - FOR SAMPLE PURPOSES ONLY

const DISPLAYCONFIG\_VIDEO\_SIGNAL\_INFO IndirectDeviceContext::s\_KnownMonitorModes[] =

{

// 800 x 600 @ 60Hz

{

40 \* MHZ, // pixel clock rate [Hz]

{ 40 \* MHZ, 800 + 256 }, // fractional horizontal refresh rate [Hz]

{ 40 \* MHZ, (800 + 256) \* (600 + 28) }, // fractional vertical refresh rate [Hz]

{ 800, 600 }, // (horizontal, vertical) active pixel resolution

{ 800 + 256, 600 + 28 }, // (horizontal, vertical) total pixel resolution

{ { 255, 0 }}, // video standard and vsync divider

DISPLAYCONFIG\_SCANLINE\_ORDERING\_PROGRESSIVE

},

// 640 x 480 @ 60Hz

{

25175 \* KHZ, // pixel clock rate [Hz]

{ 25175 \* KHZ, 640 + 160 }, // fractional horizontal refresh rate [Hz]

{ 25175 \* KHZ, (640 + 160) \* (480 + 46) }, // fractional vertical refresh rate [Hz]

{ 640, 480 }, // (horizontal, vertical) active pixel resolution

{ 640 + 160, 480 + 46 }, // (horizontal, vertical) blanking pixel resolution

{ { 255, 0 } }, // video standard and vsync divider

DISPLAYCONFIG\_SCANLINE\_ORDERING\_PROGRESSIVE

},

// 800 x 600 @ 60Hz

{

40 \* MHZ, // pixel clock rate [Hz]

{ 40 \* MHZ, 800 + 256 }, // fractional horizontal refresh rate [Hz]

{ 40 \* MHZ, (800 + 256) \* (600 + 28) }, // fractional vertical refresh rate [Hz]

{ 1920, 1280 }, // (horizontal, vertical) active pixel resolution

{ 1920 + 256, 1280 + 28 }, // (horizontal, vertical) total pixel resolution

{ { 255, 0 }}, // video standard and vsync divider

DISPLAYCONFIG\_SCANLINE\_ORDERING\_PROGRESSIVE

},

{

229009 \* KHZ, // pixel clock rate [Hz]

{ 229009 \* KHZ, 2560 + 40 }, // fractional horizontal refresh rate [Hz]

{ 229009 \* KHZ, (2560 + 40) \* (1440 + 28) }, // fractional vertical refresh rate [Hz]

{ 2560, 1440 }, // (horizontal, vertical) active pixel resolution

{ 2560 + 40, 1440 + 28 }, // (horizontal, vertical) total pixel resolution

{ { 255, 0 }}, // video standard and vsync divider

DISPLAYCONFIG\_SCANLINE\_ORDERING\_PROGRESSIVE

},

{

509367 \* KHZ, // pixel clock rate [Hz]

{ 509367 \* KHZ, 3840 + 40 }, // fractional horizontal refresh rate [Hz]

{ 509367 \* KHZ, (3840 + 40) \* (2160 + 28) }, // fractional vertical refresh rate [Hz]

{ 3840, 2160 }, // (horizontal, vertical) active pixel resolution

{ 3840 + 40, 2160 + 28 }, // (horizontal, vertical) total pixel resolution

{ { 255, 0 }}, // video standard and vsync divider

DISPLAYCONFIG\_SCANLINE\_ORDERING\_PROGRESSIVE

},

};

// This is a sample monitor EDID - FOR SAMPLE PURPOSES ONLY

const BYTE IndirectDeviceContext::s\_KnownMonitorEdid[] =

{

0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x31, 0xD8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x05, 0x16, 0x01, 0x03, 0x6D, 0x32, 0x1C, 0x78, 0xEA, 0x5E, 0xC0, 0xA4, 0x59, 0x4A, 0x98, 0x25,

0x20, 0x50, 0x54, 0x00, 0x00, 0x00, 0xD1, 0xC0, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,

0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x02, 0x3A, 0x80, 0x18, 0x71, 0x38, 0x2D, 0x40, 0x58, 0x2C,

0x45, 0x00, 0xF4, 0x19, 0x11, 0x00, 0x00, 0x1E, 0x00, 0x00, 0x00, 0xFF, 0x00, 0x4C, 0x69, 0x6E,

0x75, 0x78, 0x20, 0x23, 0x30, 0x0A, 0x20, 0x20, 0x20, 0x20, 0x00, 0x00, 0x00, 0xFD, 0x00, 0x3B,

0x3D, 0x42, 0x44, 0x0F, 0x00, 0x0A, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x00, 0x00, 0x00, 0xFC,

0x00, 0x4C, 0x69, 0x6E, 0x75, 0x78, 0x20, 0x46, 0x48, 0x44, 0x0A, 0x20, 0x20, 0x20, 0x00, 0x05

};

IndirectDeviceContext::IndirectDeviceContext(\_In\_ WDFDEVICE WdfDevice) :

m\_WdfDevice(WdfDevice)

{

m\_Adapter = {};

}

IndirectDeviceContext::~IndirectDeviceContext()

{

m\_ProcessingThread.reset();

}

#define NUM\_VIRTUAL\_DISPLAYS 1

void IndirectDeviceContext::InitAdapter()

{

// ==============================

// Update the below diagnostic information in accordance with the target hardware. The strings and version

// numbers are used for telemetry and may be displayed to the user in some situations.

//

// This is also where static per-adapter capabilities are determined.

// ==============================

IDDCX\_ADAPTER\_CAPS AdapterCaps = {};

AdapterCaps.Size = sizeof(AdapterCaps);

// Declare basic feature support for the adapter (required)

AdapterCaps.MaxMonitorsSupported = NUM\_VIRTUAL\_DISPLAYS;

AdapterCaps.EndPointDiagnostics.Size = sizeof(AdapterCaps.EndPointDiagnostics);

AdapterCaps.EndPointDiagnostics.GammaSupport = IDDCX\_FEATURE\_IMPLEMENTATION\_NONE;

AdapterCaps.EndPointDiagnostics.TransmissionType = IDDCX\_TRANSMISSION\_TYPE\_WIRED\_OTHER;

// Declare your device strings for telemetry (required)

AdapterCaps.EndPointDiagnostics.pEndPointFriendlyName = L"IddSample Device";

AdapterCaps.EndPointDiagnostics.pEndPointManufacturerName = L"Microsoft";

AdapterCaps.EndPointDiagnostics.pEndPointModelName = L"IddSample Model";

// Declare your hardware and firmware versions (required)

IDDCX\_ENDPOINT\_VERSION Version = {};

Version.Size = sizeof(Version);

Version.MajorVer = 1;

AdapterCaps.EndPointDiagnostics.pFirmwareVersion = &Version;

AdapterCaps.EndPointDiagnostics.pHardwareVersion = &Version;

// Initialize a WDF context that can store a pointer to the device context object

WDF\_OBJECT\_ATTRIBUTES Attr;

WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE(&Attr, IndirectDeviceContextWrapper);

IDARG\_IN\_ADAPTER\_INIT AdapterInit = {};

AdapterInit.WdfDevice = m\_WdfDevice;

AdapterInit.pCaps = &AdapterCaps;

AdapterInit.ObjectAttributes = &Attr;

// Start the initialization of the adapter, which will trigger the AdapterFinishInit callback later

IDARG\_OUT\_ADAPTER\_INIT AdapterInitOut;

NTSTATUS Status = IddCxAdapterInitAsync(&AdapterInit, &AdapterInitOut);

if (NT\_SUCCESS(Status))

{

// Store a reference to the WDF adapter handle

m\_Adapter = AdapterInitOut.AdapterObject;

// Store the device context object into the WDF object context

auto\* pContext = WdfObjectGet\_IndirectDeviceContextWrapper(AdapterInitOut.AdapterObject);

pContext->pContext = this;

}

}

void IndirectDeviceContext::FinishInit()

{

for (unsigned int i = 0; i < NUM\_VIRTUAL\_DISPLAYS; i++) {

CreateMonitor(i);

}

}

void IndirectDeviceContext::CreateMonitor(unsigned int index) {

// ==============================

// In a real driver, the EDID should be retrieved dynamically from a connected physical monitor. The EDID

// provided here is purely for demonstration, as it describes only 640x480 @ 60 Hz and 800x600 @ 60 Hz. Monitor

// manufacturers are required to correctly fill in physical monitor attributes in order to allow the OS to optimize

// settings like viewing distance and scale factor. Manufacturers should also use a unique serial number every

// single device to ensure the OS can tell the monitors apart.

// ==============================

WDF\_OBJECT\_ATTRIBUTES Attr;

WDF\_OBJECT\_ATTRIBUTES\_INIT\_CONTEXT\_TYPE(&Attr, IndirectDeviceContextWrapper);

IDDCX\_MONITOR\_INFO MonitorInfo = {};

MonitorInfo.Size = sizeof(MonitorInfo);

MonitorInfo.MonitorType = DISPLAYCONFIG\_OUTPUT\_TECHNOLOGY\_HDMI;

MonitorInfo.ConnectorIndex = index;

MonitorInfo.MonitorDescription.Size = sizeof(MonitorInfo.MonitorDescription);

MonitorInfo.MonitorDescription.Type = IDDCX\_MONITOR\_DESCRIPTION\_TYPE\_EDID;

MonitorInfo.MonitorDescription.DataSize = sizeof(s\_KnownMonitorEdid);

MonitorInfo.MonitorDescription.pData = const\_cast<BYTE\*>(s\_KnownMonitorEdid);

// ==============================

// The monitor's container ID should be distinct from "this" device's container ID if the monitor is not

// permanently attached to the display adapter device object. The container ID is typically made unique for each

// monitor and can be used to associate the monitor with other devices, like audio or input devices. In this

// sample we generate a random container ID GUID, but it's best practice to choose a stable container ID for a

// unique monitor or to use "this" device's container ID for a permanent/integrated monitor.

// ==============================

// Create a container ID

CoCreateGuid(&MonitorInfo.MonitorContainerId);

IDARG\_IN\_MONITORCREATE MonitorCreate = {};

MonitorCreate.ObjectAttributes = &Attr;

MonitorCreate.pMonitorInfo = &MonitorInfo;

// Create a monitor object with the specified monitor descriptor

IDARG\_OUT\_MONITORCREATE MonitorCreateOut;

NTSTATUS Status = IddCxMonitorCreate(m\_Adapter, &MonitorCreate, &MonitorCreateOut);

if (NT\_SUCCESS(Status))

{

m\_Monitor = MonitorCreateOut.MonitorObject;

// Associate the monitor with this device context

auto\* pContext = WdfObjectGet\_IndirectDeviceContextWrapper(MonitorCreateOut.MonitorObject);

pContext->pContext = this;

// Tell the OS that the monitor has been plugged in

IDARG\_OUT\_MONITORARRIVAL ArrivalOut;

Status = IddCxMonitorArrival(m\_Monitor, &ArrivalOut);

}

}

void IndirectDeviceContext::AssignSwapChain(IDDCX\_SWAPCHAIN SwapChain, LUID RenderAdapter, HANDLE NewFrameEvent)

{

m\_ProcessingThread.reset();

auto Device = make\_shared<Direct3DDevice>(RenderAdapter);

if (FAILED(Device->Init()))

{

// It's important to delete the swap-chain if D3D initialization fails, so that the OS knows to generate a new

// swap-chain and try again.

WdfObjectDelete(SwapChain);

}

else

{

// Create a new swap-chain processing thread

m\_ProcessingThread.reset(new SwapChainProcessor(SwapChain, Device, NewFrameEvent));

}

}

void IndirectDeviceContext::UnassignSwapChain()

{

// Stop processing the last swap-chain

m\_ProcessingThread.reset();

}

#pragma endregion

#pragma region DDI Callbacks

\_Use\_decl\_annotations\_

NTSTATUS IddSampleAdapterInitFinished(IDDCX\_ADAPTER AdapterObject, const IDARG\_IN\_ADAPTER\_INIT\_FINISHED\* pInArgs)

{

// This is called when the OS has finished setting up the adapter for use by the IddCx driver. It's now possible

// to report attached monitors.

auto\* pContext = WdfObjectGet\_IndirectDeviceContextWrapper(AdapterObject);

if (NT\_SUCCESS(pInArgs->AdapterInitStatus))

{

pContext->pContext->FinishInit();

}

return STATUS\_SUCCESS;

}

\_Use\_decl\_annotations\_

NTSTATUS IddSampleAdapterCommitModes(IDDCX\_ADAPTER AdapterObject, const IDARG\_IN\_COMMITMODES\* pInArgs)

{

UNREFERENCED\_PARAMETER(AdapterObject);

UNREFERENCED\_PARAMETER(pInArgs);

// For the sample, do nothing when modes are picked - the swap-chain is taken care of by IddCx

// ==============================

// TODO: In a real driver, this function would be used to reconfigure the device to commit the new modes. Loop

// through pInArgs->pPaths and look for IDDCX\_PATH\_FLAGS\_ACTIVE. Any path not active is inactive (e.g. the monitor

// should be turned off).

// ==============================

return STATUS\_SUCCESS;

}

\_Use\_decl\_annotations\_

NTSTATUS IddSampleParseMonitorDescription(const IDARG\_IN\_PARSEMONITORDESCRIPTION\* pInArgs, IDARG\_OUT\_PARSEMONITORDESCRIPTION\* pOutArgs)

{

// ==============================

// In a real driver, this function would be called to generate monitor modes for an EDID by parsing it. In

// this sample driver, we hard-code the EDID, so this function can generate known modes.

// ==============================

pOutArgs->MonitorModeBufferOutputCount = ARRAYSIZE(IndirectDeviceContext::s\_KnownMonitorModes);

if (pInArgs->MonitorModeBufferInputCount < ARRAYSIZE(IndirectDeviceContext::s\_KnownMonitorModes))

{

// Return success if there was no buffer, since the caller was only asking for a count of modes

return (pInArgs->MonitorModeBufferInputCount > 0) ? STATUS\_BUFFER\_TOO\_SMALL : STATUS\_SUCCESS;

}

else

{

// Copy the known modes to the output buffer

for (DWORD ModeIndex = 0; ModeIndex < ARRAYSIZE(IndirectDeviceContext::s\_KnownMonitorModes); ModeIndex++)

{

pInArgs->pMonitorModes[ModeIndex].Size = sizeof(IDDCX\_MONITOR\_MODE);

pInArgs->pMonitorModes[ModeIndex].Origin = IDDCX\_MONITOR\_MODE\_ORIGIN\_MONITORDESCRIPTOR;

pInArgs->pMonitorModes[ModeIndex].MonitorVideoSignalInfo = IndirectDeviceContext::s\_KnownMonitorModes[ModeIndex];

}

// Set the preferred mode as represented in the EDID

pOutArgs->PreferredMonitorModeIdx = 0;

return STATUS\_SUCCESS;

}

}

\_Use\_decl\_annotations\_

NTSTATUS IddSampleMonitorGetDefaultModes(IDDCX\_MONITOR MonitorObject, const IDARG\_IN\_GETDEFAULTDESCRIPTIONMODES\* pInArgs, IDARG\_OUT\_GETDEFAULTDESCRIPTIONMODES\* pOutArgs)

{

UNREFERENCED\_PARAMETER(MonitorObject);

UNREFERENCED\_PARAMETER(pInArgs);

UNREFERENCED\_PARAMETER(pOutArgs);

// Should never be called since we create a single monitor with a known EDID in this sample driver.

// ==============================

// TODO: In a real driver, this function would be called to generate monitor modes for a monitor with no EDID.

// Drivers should report modes that are guaranteed to be supported by the transport protocol and by nearly all

// monitors (such 640x480, 800x600, or 1024x768). If the driver has access to monitor modes from a descriptor other

// than an EDID, those modes would also be reported here.

// ==============================

return STATUS\_NOT\_IMPLEMENTED;

}

/// <summary>

/// Creates a target mode from the fundamental mode attributes.

/// </summary>

void CreateTargetMode(DISPLAYCONFIG\_VIDEO\_SIGNAL\_INFO& Mode, UINT Width, UINT Height, UINT VSync)

{

Mode.totalSize.cx = Mode.activeSize.cx = Width;

Mode.totalSize.cy = Mode.activeSize.cy = Height;

Mode.AdditionalSignalInfo.vSyncFreqDivider = 1;

Mode.AdditionalSignalInfo.videoStandard = 255;

Mode.vSyncFreq.Numerator = VSync;

Mode.vSyncFreq.Denominator = Mode.hSyncFreq.Denominator = 1;

Mode.hSyncFreq.Numerator = VSync \* Height;

Mode.scanLineOrdering = DISPLAYCONFIG\_SCANLINE\_ORDERING\_PROGRESSIVE;

Mode.pixelRate = VSync \* Width \* Height;

}

void CreateTargetMode(IDDCX\_TARGET\_MODE& Mode, UINT Width, UINT Height, UINT VSync)

{

Mode.Size = sizeof(Mode);

CreateTargetMode(Mode.TargetVideoSignalInfo.targetVideoSignalInfo, Width, Height, VSync);

}

\_Use\_decl\_annotations\_

NTSTATUS IddSampleMonitorQueryModes(IDDCX\_MONITOR MonitorObject, const IDARG\_IN\_QUERYTARGETMODES\* pInArgs, IDARG\_OUT\_QUERYTARGETMODES\* pOutArgs)

{

UNREFERENCED\_PARAMETER(MonitorObject);

vector<IDDCX\_TARGET\_MODE> TargetModes(6);

// Create a set of modes supported for frame processing and scan-out. These are typically not based on the

// monitor's descriptor and instead are based on the static processing capability of the device. The OS will

// report the available set of modes for a given output as the intersection of monitor modes with target modes.

CreateTargetMode(TargetModes[0], 3840, 2160, 60);

CreateTargetMode(TargetModes[1], 2560, 1440, 60);

CreateTargetMode(TargetModes[2], 1920, 1080, 60);

CreateTargetMode(TargetModes[3], 1024, 768, 60);

CreateTargetMode(TargetModes[4], 800, 600, 60);

CreateTargetMode(TargetModes[5], 640, 480, 60);

pOutArgs->TargetModeBufferOutputCount = (UINT)TargetModes.size();

if (pInArgs->TargetModeBufferInputCount >= TargetModes.size())

{

copy(TargetModes.begin(), TargetModes.end(), pInArgs->pTargetModes);

}

return STATUS\_SUCCESS;

}

\_Use\_decl\_annotations\_

NTSTATUS IddSampleMonitorAssignSwapChain(IDDCX\_MONITOR MonitorObject, const IDARG\_IN\_SETSWAPCHAIN\* pInArgs)

{

auto\* pContext = WdfObjectGet\_IndirectDeviceContextWrapper(MonitorObject);

pContext->pContext->AssignSwapChain(pInArgs->hSwapChain, pInArgs->RenderAdapterLuid, pInArgs->hNextSurfaceAvailable);

return STATUS\_SUCCESS;

}

\_Use\_decl\_annotations\_

NTSTATUS IddSampleMonitorUnassignSwapChain(IDDCX\_MONITOR MonitorObject)

{

auto\* pContext = WdfObjectGet\_IndirectDeviceContextWrapper(MonitorObject);

pContext->pContext->UnassignSwapChain();

return STATUS\_SUCCESS;

}

#pragma endregion

**main.cpp** *(Приложение для вызова драйвера)*

#include <iostream>

#include <vector>

#include <windows.h>

#include <swdevice.h>

#include <conio.h>

#include <wrl.h>

VOID WINAPI

CreationCallback(

\_In\_ HSWDEVICE hSwDevice,

\_In\_ HRESULT hrCreateResult,

\_In\_opt\_ PVOID pContext,

\_In\_opt\_ PCWSTR pszDeviceInstanceId

)

{

HANDLE hEvent = \*(HANDLE\*) pContext;

SetEvent(hEvent);

UNREFERENCED\_PARAMETER(hSwDevice);

UNREFERENCED\_PARAMETER(hrCreateResult);

UNREFERENCED\_PARAMETER(pszDeviceInstanceId);

}

int \_\_cdecl main(int argc, wchar\_t \*argv[])

{

UNREFERENCED\_PARAMETER(argc);

UNREFERENCED\_PARAMETER(argv);

HANDLE hEvent = CreateEvent(nullptr, FALSE, FALSE, nullptr);

HSWDEVICE hSwDevice;

SW\_DEVICE\_CREATE\_INFO createInfo = { 0 };

PCWSTR description = L"Idd Sample Driver";

// These match the Pnp id's in the inf file so OS will load the driver when the device is created

PCWSTR instanceId = L"IddSampleDriver";

PCWSTR hardwareIds = L"IddSampleDriver\0\0";

PCWSTR compatibleIds = L"IddSampleDriver\0\0";

createInfo.cbSize = sizeof(createInfo);

createInfo.pszzCompatibleIds = compatibleIds;

createInfo.pszInstanceId = instanceId;

createInfo.pszzHardwareIds = hardwareIds;

createInfo.pszDeviceDescription = description;

createInfo.CapabilityFlags = SWDeviceCapabilitiesRemovable |

SWDeviceCapabilitiesSilentInstall |

SWDeviceCapabilitiesDriverRequired;

// Create the device

HRESULT hr = SwDeviceCreate(L"IddSampleDriver",

L"HTREE\\ROOT\\0",

&createInfo,

0,

nullptr,

CreationCallback,

&hEvent,

&hSwDevice);

if (FAILED(hr))

{

printf("SwDeviceCreate failed with 0x%lx\n", hr);

return 1;

}

// Wait for callback to signal that the device has been created

printf("Waiting for device to be created....\n");

DWORD waitResult = WaitForSingleObject(hEvent, 10\*1000);

if (waitResult != WAIT\_OBJECT\_0)

{

printf("Wait for device creation failed\n");

return 1;

}

printf("Device created\n\n");

// Now wait for user to indicate the device should be stopped

printf("Press 'x' to exit and destory the software device\n");

bool bExit = false;

do

{

// Wait for key press

int key = \_getch();

if (key == 'x' || key == 'X')

{

bExit = true;

}

}while (!bExit);

// Stop the device, this will cause the sample to be unloaded

SwDeviceClose(hSwDevice);

return 0;

}