Министерство образования Республики Беларусь
Учреждение образования
«Брестский государственный технический университет»
Кафедра ИИТ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ ПО ДИСЦИПЛИНЕ
«Системное программирование»
Тема: «Драйвер виртуального монитора для ОС Windows»

КП.ПО-9.1-40-01-01

Листов: 23

Выполнил:
Студент 3-го курса,
ФЭИС,
Группы ПО-9
Мисиюк А. С.
Нормоконтроль:
Козик И.Д.
Проверил:
Козик И.Д.

Брест 2023

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
КАФЕДРА «ИНТЕЛЛЕКТУАЛЬНЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»
Факультет ЭЛЕКТРОННО-ИНФОРМАЦИОННЫХ СИСТЕМ

"УТВЕРЖДАЮ"
Заведующий кафедрой ИИТ _____
(подпись)
«___» _____ 20__ год

З А Д А Н И Е
по курсовому проектированию

Студенту Мисиюку Алексею Сергеевичу **группа ПО-9**

1.Тема проекта: «Драйвер виртуального монитора для ОС Windows».

2. Сроки сдачи студентом законченного курсового проекта: 08.12.2023г.

3. Исходные данные к курсовому проекту (КП):

3.1. Документация драйверов на ОС Windows

3.2. Требования к оформлению пояснительной записки - см. 4.1.

4. Перечень методического обеспечения по курсовому проектированию:

4.1. Хвещук В.И, Крапивин Ю.Б., Муравьев Г.Л. Методическое пособие по курсовому проектированию, БрГТУ, ИИТ, 2012.- 76с. Заказ № 1106.
4.2. Краткие рекомендации по содержанию пояснительной записки КП. - ЛВС кафедры ИИТ, диск К – LOOK - каталог ББД 2017
4.3. ГОСТ ЕСПД 19.502-2000. Описание применения
4.4. ГОСТ ЕСПД 19.502-2000. Программа и методика испытаний.
4.5. ГОСТ ЕСПД 19.401. Текст программы.
4.6. Сайт Microsoft с документацией написания драйверов на ОС Windows Microsoft Learn - https://learn.microsoft.com/ru-ru/windows-hardware/drivers/

6. Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов):

ВЕДЕНИЕ (постановка задачи).

1. ЧТО ТАКОЕ ДРАЙВЕР
1.1. Введение в драйвер
1.2. Пользовательский режим и режим ядра

2. МОДЕЛЬ WINDOWS DISPLAY DRIVER MODEL (WDDM)
2.1. Архитектура WDDM
2.2. Поток операций windows Display Driver Model (WDDM)
2.3. Преимущества WDDM

3. ДАРЙВЕР КОСВЕННОГО ОТОБРАЖЕНИЯ (IDD)

4. РЕЗУЛЬТАТЫ НАПИСАНИЯ И ИСПЫТАНИЯ IDD

ЗАКЛЮЧЕНИЕ (результаты полученные в КП)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

ПРИЛОЖЕНИЯ:
А. ИНСТРУКЦИЯ ПО УСТАНОВКЕ И ПРОВЕРКЕ IDD
Б. КОД ПРОГРАММЫ

7. Консультанты по проекту: по расписанию консультаций для группы

8. График аттестации КП: по приказу:

1-я аттестация – раздел 1
2-я аттестация – раздел 2

9. Дата выдачи курсового проекта - 04.09.2023 г.

10. Календарный график работы над проектом на весь период проектирования (с указанием сроков выполнения и трудоемкости отдельных этапов)

№ п/п	Содержание курсового проекта	Сроки выполнения проекта	Объем работы в %
1	Изучение документации (графических) драйверов	7 недель	50 %
2	Программирование, отладка, тестирование драйвера	5 недель	36 %
3	Оформление проекта	1 неделя	7 %
4	Аттестация курсового проекта	1 неделя	7 %

Руководитель _____

(подпись)

Задание принял к исполнению _____

(дата и подпись студента)

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Курсовой проект на тему "*Драйвер виртуального монитора для операционной системы Windows*" представляет собой ценный опыт в области разработки драйверов и программного обеспечения под Windows. Изучение данной темы позволяет погрузиться в глубины операционной системы, а также освоить тонкости взаимодействия с аппаратными компонентами компьютера. В рамках проекта будет рассмотрен частично процесс создания графического драйвера, начиная с анализа модели и структур графических драйверов и заканчивая тестированием и испытанием его работы на реальном стационарном компьютере. Такой опыт не только расширит понимание работы операционной системы, но и позволит освоить навыки разработки на более низком уровне абстракции, что является важным компонентом в профессиональной карьере в сфере разработки ПО.

Среда разработки – Microsoft Visual Studio, а также наборы инструментов: SDK и WDK 10.

Операционная система, под которую создается драйвер – Windows 10. Хотя драйвер создается по универсальной модели и подойдет для ОС Windows от 8-й версии.

Система контроля версий – Git. Проект, а также готовые файлы проекта (включая драйвер) и инструкцию по использованию, можно найти на публичном репозитории по адресу: *https://github.com/11ALX11/SP_kurz/*.

Альтернативно инструкция представлена в Приложении А.

1. ЧТО ТАКОЕ ДРАЙВЕР

1.1. Введение в драйвер

Многие считают что самому создать драйвер для Windows это что-то на грани фантастики. Но на самом деле это не так. Конечно, разработка драйвера для какого-то навороченного девайса бывает не простой задачей. Но ведь тоже самое можно сказать про создание сложных программ или игр.

Сложно дать одно точное определение термина "драйвер". В самом фундаментальном смысле драйвер — это программный компонент, который позволяет операционной системе и устройству взаимодействовать друг с другом.

Например, предположим, что приложению необходимо считывать некоторые данные с устройства. Приложение вызывает функцию, реализованную операционной системой, а операционная система вызывает функцию, реализованную драйвером. Драйвер, написанный той же компанией, которая разработала и изготовила устройство, знает, как взаимодействовать с оборудованием устройства для получения данных. После того как драйвер получает данные с устройства, он возвращает данные в операционную систему, которая возвращает их приложению (рис. 1).

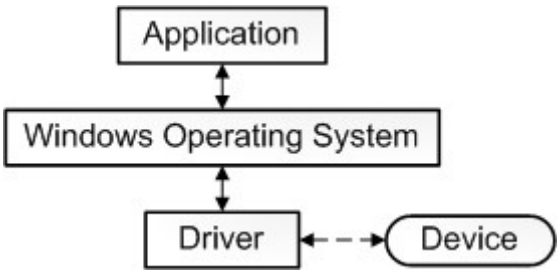

Рисунок 1 – простая схема взаимодействия драйвера и системы

Развертывание определения.

Наше объяснение до сих пор упрощено несколькими способами:

- **Не все драйверы должны быть написаны компанией, которая разработала устройство.**

Во многих случаях устройство разработано в соответствии с опубликованным стандартом оборудования. Таким образом, драйвер может быть написан корпорацией Майкрософт, и конструктор устройств не должен предоставлять драйвер.

- **Не все драйверы взаимодействуют напрямую с устройством.**

Для заданного запроса ввода-вывода (например, чтения данных с устройства) в стеке драйверов часто имеется несколько драйверов, участвующих в запросе. Обычный способ визуализации стека — первый участник вверху и последний участник в нижней части, как показано на этой схеме. Некоторые драйверы в стеке могут участвовать в преобразовании запроса из одного формата в другой. Эти драйверы не взаимодействуют напрямую с устройством; они просто управляют запросом и передают запрос драйверам, которые находятся ниже в стеке (рис. 2).

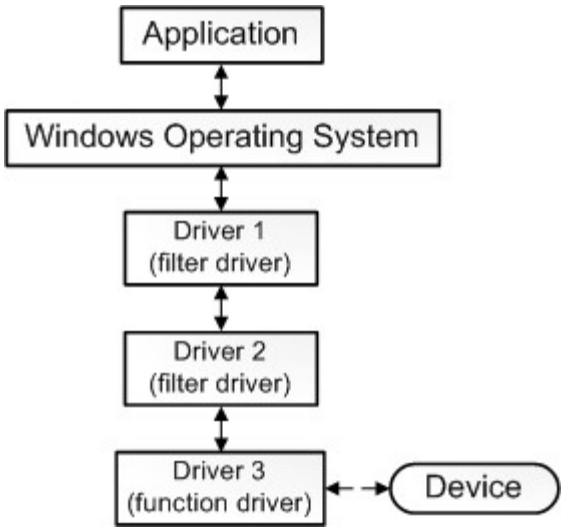

Рисунок 2 – схема взаимодействия драйверов (стека драйверов) с системой

Драйвер функции. Драйвер функции в стеке, который напрямую взаимодействует с устройством, называется драйвером функции.

Драйвер фильтра. Драйверы, выполняющие вспомогательную обработку, называются драйверами фильтров.

- Некоторые драйверы фильтров отслеживают и записывают сведения о запросах ввода-вывода, но не участвуют в них активно. Например, некоторые драйверы фильтров выступают в качестве проверяющих, чтобы убедиться, что другие драйверы в стеке обрабатывают запрос ввода-вывода правильно.

Мы можем расширить определение драйвера , сказав, что драйвер — это любой программный компонент, который наблюдает или участвует в обмене данными между операционной системой и устройством.

Драйверы программного обеспечения

Расширенное определение является достаточно точным, но по-прежнему неполным, так как некоторые драйверы вообще не связаны с каким-либо аппаратным устройством.

Например, предположим, что вам нужно написать средство, которое имеет доступ к основным структурам данных операционной системы. Доступ к этим структурам можно получить только с помощью кода, выполняемого в режиме ядра. Это можно сделать, разделив средство на два компонента. Первый компонент выполняется в пользовательском режиме и представляет пользовательский интерфейс. Второй компонент выполняется в режиме ядра и имеет доступ к данным основной операционной системы. Компонент, работающий в пользовательском режиме, называется приложением, а компонент, работающий в режиме ядра, называется программным драйвером. Программный драйвер не связан с аппаратным устройством.

На рисунке 3 показана схема, на которой приложение в пользовательском режиме взаимодействует с программным драйвером в режиме ядра.

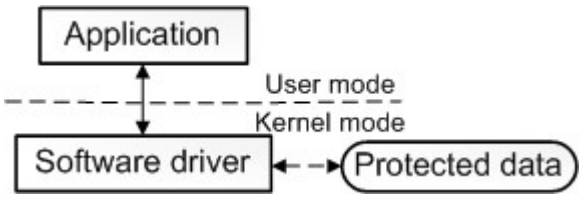

Рисунок 3 – взаимодействие пользовательского режима с режимом ядра

Программные драйверы всегда работают в режиме ядра. Главной причиной написания программного драйвера является получение доступа к защищенным данным, доступным только в режиме ядра. Однако драйверам устройств не всегда требуется доступ к данным и ресурсам в режиме ядра. Поэтому некоторые драйверы устройств работают в пользовательском режиме.

Дополнительные сведения о драйверах функций

Наше объяснение до сих пор упрощено определение драйвера функции. Мы сказали, что драйвер-функция для устройства — это драйвер в стеке, который напрямую взаимодействует с устройством. Это верно для устройства, которое подключается напрямую к шине PCI. Драйвер функции для устройства PCI получает адреса, сопоставленные с ресурсами порта и памяти на устройстве. Драйвер функции напрямую взаимодействует с устройством, записывая их на эти адреса.

Однако во многих случаях устройство не подключается напрямую к шине PCI. Вместо этого устройство подключается к адаптеру шины узла, который подключен к шине PCI. Например, USB-тостер подключается к адаптеру шины узла (называемому контроллером узла USB), который подключен к шине PCI. Usb-тостер имеет драйвер функции, а USB-контроллер узла также имеет драйвер функции. Драйвер функции для тостера косвенно взаимодействует с тостером, отправляя запрос драйверу функции для контроллера узла USB. Затем драйвер функции для хост-контроллера USB напрямую взаимодействует с оборудованием USB-контроллера узла, которое взаимодействует с тостером.

1.2. Пользовательский режим и режим ядра

Процессор на компьютере под управлением Windows имеет два разных режима: пользовательский режим и режим ядра.

Процессор переключается между двумя режимами в зависимости от типа кода, выполняемого на процессоре. Приложения выполняются в пользовательском режиме, а основные компоненты операционной системы — в режиме ядра. Хотя многие драйверы работают в режиме ядра, некоторые драйверы могут работать в пользовательском режиме.

Пользовательский режим

При запуске приложения в пользовательском режиме Windows создает процесс для приложения. Процесс предоставляет приложению частное виртуальное адресное пространство и таблицу частных дескрипторов. Так как виртуальное адресное пространство приложения является частным, одно приложение не может изменять данные, принадлежащие другому приложению. Каждое приложение выполняется изолированно, и в случае сбоя приложения сбой ограничивается одним приложением. Другие приложения и операционная система не затрагиваются сбоем.

Помимо частного, виртуальное адресное пространство приложения в пользовательском режиме ограничено. Процесс, выполняющийся в пользовательском режиме, не может получить доступ к виртуальным адресам, зарезервированным для операционной системы. Ограничение виртуального адресного пространства приложения в пользовательском режиме предотвращает изменение и, возможно, повреждение критически важных данных операционной системы.

Режим ядра

Весь код, который выполняется в режиме ядра, использует одно виртуальное адресное пространство. Таким образом, драйвер в режиме ядра не изолирован от других драйверов и самой операционной системы. Если драйвер, работающий в режиме ядра, случайно выполняет запись на неправильный виртуальный адрес, данные, принадлежащие операционной системе или другому драйверу, могут быть скомпрометированы. При сбое драйвера в режиме ядра происходит сбой всей операционной системы.

На рисунке 4 изображена схема, на которой показано взаимодействие между компонентами пользовательского режима и режима ядра.

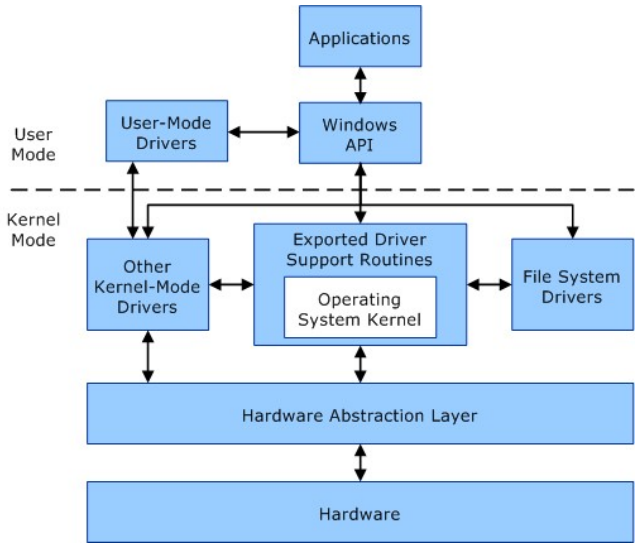

Рисунок 4 - взаимодействие между компонентами пользовательского и режима ядра

2. МОДЕЛЬ WINDOWS DISPLAY DRIVER MODEL (WDDM)

2.1. Архитектура WDDM

Модель драйвера дисплея Windows (WDDM) — это архитектура драйвера графического дисплея, представленная в Windows Vista (WDDM 1.0).

Для модели windows Display Driver Model (WDDM) требуется, чтобы поставщик графического оборудования предоставил сопряженный драйвер отображения пользовательского режима (UMD) и драйвер отображения в режиме ядра (KMD; также называется драйвером минипорта дисплея).

Архитектура модели драйвера дисплея WDDM состоит из частей режима пользователя и режима ядра. На рисунке 5 показана архитектура, необходимая для поддержки WDDM.

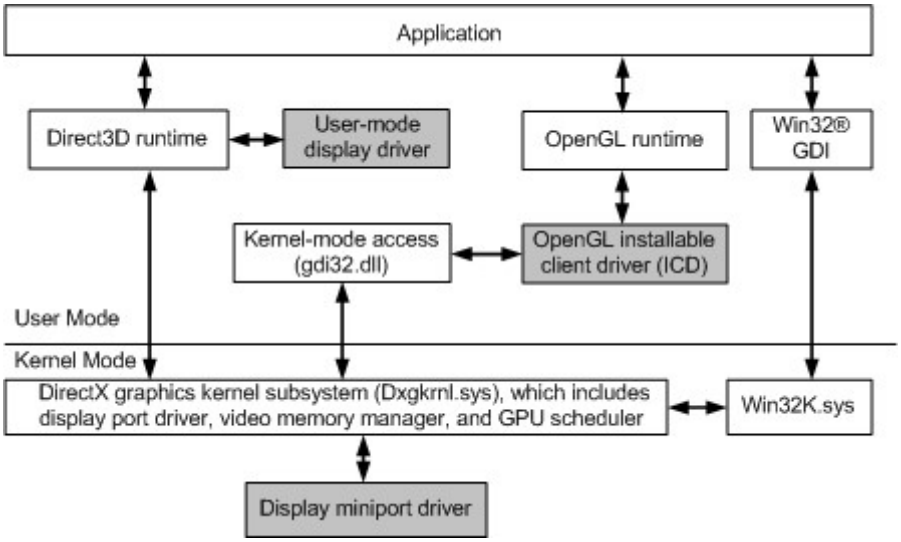

Рисунок 5 – архитектура WDDM

Поставщик графического оборудования должен предоставить драйвер дисплея в пользовательском режиме и драйвер мини-порта дисплея (также известный как драйвер отображения в режиме ядра или KMD).

- Драйвер отображения в пользовательском режиме — это библиотека динамической компоновки (DLL), загружаемая средой выполнения Direct3D;
- Драйвер мини-порта дисплея взаимодействует с подсистемой ядра графики DirectX.

2.2. Поток операций windows Display Driver Model (WDDM)

На следующей схеме показан поток операций WDDM, выполняемых с момента создания устройства отрисовки до отображения содержимого на дисплее. Сведения, приведенные на схеме, более подробно описывают упорядоченную последовательность потока операций.

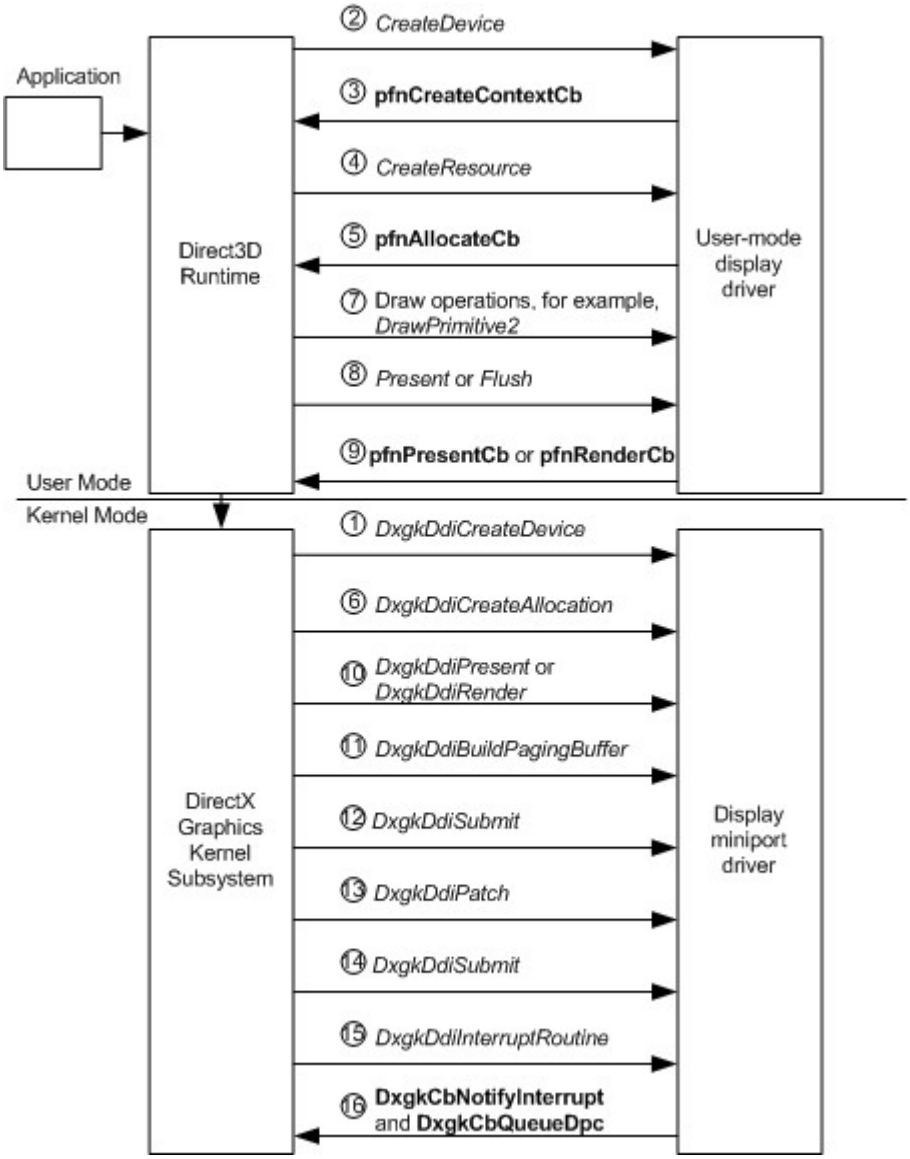

Рисунок 6 – поток операций WDDM

- **Создание устройства отрисовки**

После того, как приложение запросит создание устройства отрисовки, выполните приведенные далее действия.

1. Подсистема ядра графики DirectX (Dxgkrnl) вызывает функцию DxgkDdiCreateDevice драйвера минипорта дисплея (KMD). KMD инициализирует прямой доступ к памяти (DMA), возвращая указатель на заполненную структуру DXGK_DEVICEINFO в элементе pInfoструктуры DXGKARG_CREATEDEVICE;

2. Если вызов DxgkDdiCreateDevice завершается успешно, среда выполнения Direct3D вызывает функцию CreateDevice драйвера отображения пользовательского режима (UMD);

3. В вызове CreateDevice UMD должен явным образом вызвать функцию pfnCreateContextCb среды выполнения для создания одного или нескольких контекстов GPU, которые являются потоками выполнения GPU на только что созданном устройстве. Среда выполнения возвращает сведения в UMD в элементах pCommandBuffer и CommandBufferSize структуры D3DDDICB_CREATECONTEXT для инициализации буфера команд;

- **Создание поверхностей для устройства**

После того как приложение запросит создание поверхностей для устройства отрисовки, выполните следующие действия.

4. Среда выполнения Direct3D вызывает функцию CreateResource UMD;

5. CreateResource вызывает предоставленную средой выполнения функцию pfnAllocateCb;

6. Среда выполнения вызывает функцию DXGkDdiCreateAllocation KMD, указывая количество и типы создаваемых выделений. DxgkDdiCreateAllocation возвращает сведения о выделениях в массиве DXGK_ALLOCATIONINFO структур в элементе pAllocationInfoструктуры DXGKARG_CREATEALLOCATION;

- **Отправка буфера команд в режим ядра**

После того, как приложение запросит рисование на поверхность:

7. Среда выполнения Direct3D вызывает функцию UMD, связанную с операцией рисования, например DrawPrimitive2;

8. Среда выполнения Direct3D вызывает функцию UMD Present или Flush, чтобы вызвать отправку буфера команд в режим ядра. Примечание. UMD также отправляет буфер команд, когда буфер команд заполнен;

9. В ответ на шаг 8 UMD вызывает одну из следующих функций, предоставляемых средой выполнения:

- Функция pfnPresentCb среды выполнения, если был вызван метод Present;
- Функция pfnRenderCb среды выполнения, если был вызван метод Flush или буфер команд заполнен.

10. функция DxgkDdiPresent в KMD вызывается, если был вызван pfnPresentCb , либо dxgkDdiRender или DxgkDdiRenderKm , если был вызван pfnRenderCb . KMD проверяет буфер команд, записывает в буфер DMA в формате оборудования и создает список выделения, описывающий используемые поверхности;

- **Отправка буфера DMA на оборудование**

11. Dxgkrnl вызывает функцию DXGkDdiBuildPagingBuffer KMD для создания буферов DMA специального назначения, которые перемещают выделения, указанные в списке выделения, в память, доступную для GPU, и из нее. Эти специальные буферы DMA называются буферами разбиения по страницам. DxgkDdiBuildPagingBuffer не вызывается для каждого кадра;

12. Dxgkrnl вызывает функцию DXGkDdiSubmitCommand KMD для постановки буферов подкачки в единицу выполнения GPU;

13. Dxgkrnl вызывает функцию DXGkDdiPatch KMD для назначения физических адресов ресурсам в буфере DMA;

14. Dxgkrnl вызывает функцию DXGkDdiSubmitCommand KMD для постановки буфера DMA в единицу выполнения GPU. Каждый буфер DMA, отправленный в GPU, содержит идентификатор ограждения, который является числом. После того как GPU завершит обработку буфера DMA, GPU создает прерывание;

15. KMD получает уведомление о прерывании в функции DxgkDdiInterruptRoutine . KMD должен считывать из GPU идентификатор ограждения только что завершенного буфера DMA;

16. KMD должен вызывать функцию DxgkCbNotifyInterrupt , чтобы уведомить DXGK о завершении буфера DMA. KMD также должен вызывать функцию DxgkCbQueueDpc для постановки в очередь отложенного вызова процедуры (DPC).

2.3 Преимущества WDDM

Создавать драйверы графики и дисплея проще с помощью WDDM, а не модели XDDM в Windows 2000, так как в ней реализованы следующие улучшения. Кроме того, драйверы WDDM способствуют повышению стабильности и безопасности операционной системы. Меньше кода драйвера выполняется в режиме ядра, где он может получить доступ к системным адресным пространствам и, возможно, вызвать сбои.

- Среда выполнения Direct3D и подсистема ядра графики DirectX (Dxgkrnl) выполняют больше операций обработки отображения; то есть в среде выполнения и подсистеме находится больше кода, а не в драйверах. Эта обработка включает код, который управляет видеопамятью и планирует буферы прямого доступа к памяти (DMA) для GPU;
- Для создания Surface требуется меньше этапов в режиме ядра. Для создания Surface в операционных системах до Windows Vista требовались следующие последовательные вызовы режима ядра:
    1. DdCanCreateSurface;
    2. DdCreateSurface;
    3. D3dCreateSurfaceEx.

Для создания Surface в WDDM требуется только вызов драйвера отображения в пользовательском режиме CreateResource , который, в свою очередь, вызывает функцию pfnAllocateCb среды выполнения. Этот вызов приводит к тому, что Dxgkrnl вызывает функцию DxgkDdiCreateAllocation драйвера режима ядра;

- Вызовы, которые создают и уничтожают поверхности, а также блокируют и разблокируют ресурсы, более равномерно связаны;
- WDDM одинаково обрабатывает видеопамяти, системную память и управляемые поверхности. Операционные системы до Windows Vista обрабатывали эти компоненты тонко разными способами;
- Преобразование шейдера выполняется в пользовательском режиме части драйверов дисплея.

Этот подход устраняет следующие сложности, возникающие при выполнении преобразования шейдера в режиме ядра:

- Аппаратные модели, которые не соответствуют абстракциям интерфейса драйвера устройства (DDI);
- Сложная технология компилятора, используемая в переводе.

Так как обработка шейдера выполняется полностью для каждого процесса и доступ к оборудованию не требуется, обработка шейдера в режиме ядра не требуется. Таким образом, код перевода шейдера можно обрабатывать в пользовательском режиме.

Необходимо написать код try/except вокруг кода перевода в пользовательском режиме. Ошибки перевода должны привести к возврату к обработке приложения.

Фоновый перевод (т. е. код перевода, который выполняется в отдельном потоке от других потоков обработки экрана) проще писать в пользовательском режиме.

3. ДРАЙВЕР КОСВЕННОГО ОТОБРАЖЕНИЯ (IDD)

В качестве драйвера для создания виртуального монитора мною был выбран драйвер косвенного отображения. Почему? В моих глазах этот драйвер наиболее подходит для поставленной задачи несмотря на то, что традиционные драйвера WDDM также подойдут. IDD (или **Indirect Display Driver**) – драйвер пользовательского режима, что послужило определенным плюсом из-за простоты и безопасности его тестирования. Также логическое применение, так как при создании виртуального монитора не подразумевается никакого отображения, поэтому косвенное отображение очень даже подходит.

Модель драйвера непрямого отображения (IDD) предоставляет простую модель драйвера в пользовательском режиме для поддержки мониторов, которые не подключены к традиционным выводам дисплея GPU. Например, аппаратный ключ, подключенный к компьютеру через USB, к которому подключен обычный монитор (VGA, DVI, HDMI, DP и т.д.).

Реализация IDD

IdD — это сторонний драйвер UMDF для устройства. Он разработан с использованием функциональных возможностей, предоставляемых IddCx (Класс косвенного драйвера дисплея eXtension) для взаимодействия с графическими подсистемами Windows следующими способами:

- Создание графического адаптера, представляющего устройство непрямого отображения;
- Мониторы отчетов подключены и отключены от системы;
- Укажите описания подключенных мониторов;
- Предоставление доступных режимов отображения;
- Поддержка других функциональных возможностей дисплея, таких как аппаратный курсор мыши, гамма-связь, I2C-связь и защищенное содержимое;
- Обработка изображений рабочего стола для отображения на мониторе;

Так как IDD является драйвером UMDF, он также отвечает за реализацию всех функций UMDF , таких как обмен данными с устройствами, управление питанием, plug and play и т. д.

IdD выполняется в сеансе 0 без каких-либо компонентов, запущенных в пользовательском сеансе, поэтому любая нестабильность драйвера не повлияет на стабильность системы в целом.

На рисунке 7 представлена схема, на которой обозревается архитектура:

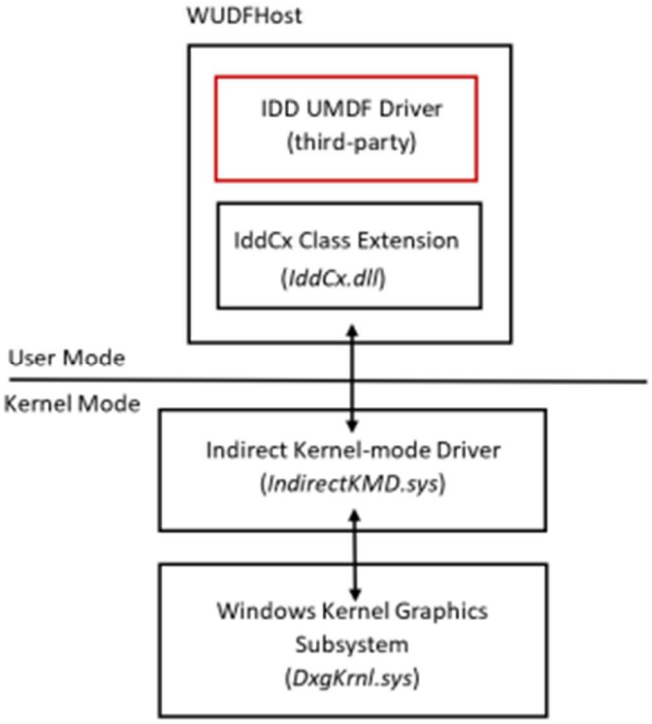

Рисунок 7 – обзор архитектуры IDD

Модель пользовательского режима

IdD — это модель только в пользовательском режиме без поддержки компонентов режима ядра. Таким образом, драйвер может использовать любые API DirectX для обработки образа рабочего стола. На самом деле IddCx предоставляет образ рабочего стола для кодирования в поверхности DirectX.

Примечание. Драйвер не должен вызывать ИНТЕРФЕЙСы API пользовательского режима, которые не подходят для использования драйверами, например GDI, ИНТЕРФЕЙСы API окон, OpenGL или Vulkan.

IdD следует создавать как универсальный драйвер Windows , чтобы его можно было использовать на нескольких платформах Windows.

Во время сборки idD UMDF объявляет версию IddCx, для которую она была создана, а ОПЕРАЦИОННая система гарантирует, что при загрузке драйвера будет загружена правильная версия IddCx.

4. РЕЗУЛЬТАТЫ НАПИСАНИЯ И ИСПЫТАНИЯ IDD

К непосредственно написанию кода драйвера можно приступать после установки необходимого ПО. Нужно установить последнюю версию Microsoft Visual Studio, наборы инструментов: SDK и WDK.

Прежде чем писать код с нуля, лучше попытаться найти образец уже готового драйвера для изучения работы структур и процессов на готовом примере, а не по сухой документации, и уже его основе реализовать требуемый функционал.

Код проекта находится в *Приложении Б.*

Состоит из 2 разделений:

- Приложение для запуска виртуального монитора временно (без установки как устройства, монитор существует до закрытия приложения);
- Непосредственно проекта самого *драйвера косвенного отображения* (*IDD*).

Дальше я укажу пару моментов, которые отличают драйвер виртуального монитора от настоящего драйвера (драйвера косвенного отображения):

- Обработка кадра (в этом проекте ее просто нет);
- Отчет по статистике кадра;
- Программно прописанный EDID и конфигурация монитора (понятное дело, это драйвер должен получать от самого монитора динамически);
- Динамическая логика контейнера вместо статического (как если бы монитор был подключен вечно, что не верно в случае с физическими мониторами, но вполне подходит в случае с виртуальным монитором).

Испытать данный драйвер не очевидная задача.

Даже если все будет работать правильно, ничего не измениться визуально. Так как мы создали монитор, которого “не существует”.

Достаточно провести мышку к границе справа (обычно именно сюда Windows помещает новые мониторы) и вести ее дальше. Если при попытке вернуть ее на экран она появляется не сразу, как если бы она была дальше, а не на границе, то можно подтвердить работоспособность данного драйвера виртуального монитора.

Хотя лучше зайти в **диспетчер устройств**, выбрать пункты **Видеоадаптеры** и **Мониторы**, где среди видеоадаптеров можно заметить наш адаптер, созданный драйвером, а среди мониторов появится новый монитор (как правило, их будет 2) (рис. 8).

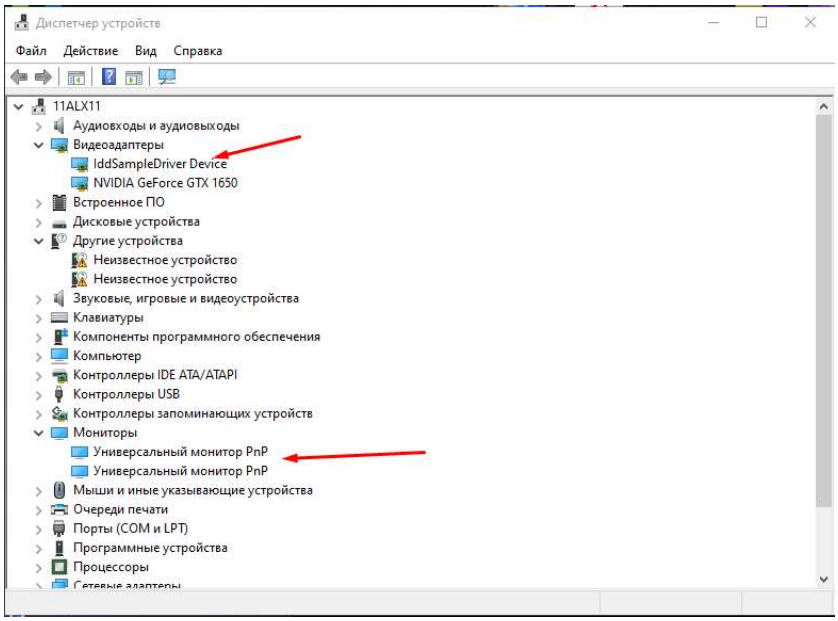

Рисунок 8 – проверка обнаружения виртуальных устройств

Также можно перейти в **Параметры** и найти **Дисплей**. Тут можно заметить, что Windows определяет 2 монитора, и при желании, 2-ому можно даже поменять разрешение на указанное программно в драйвере (рис. 9).

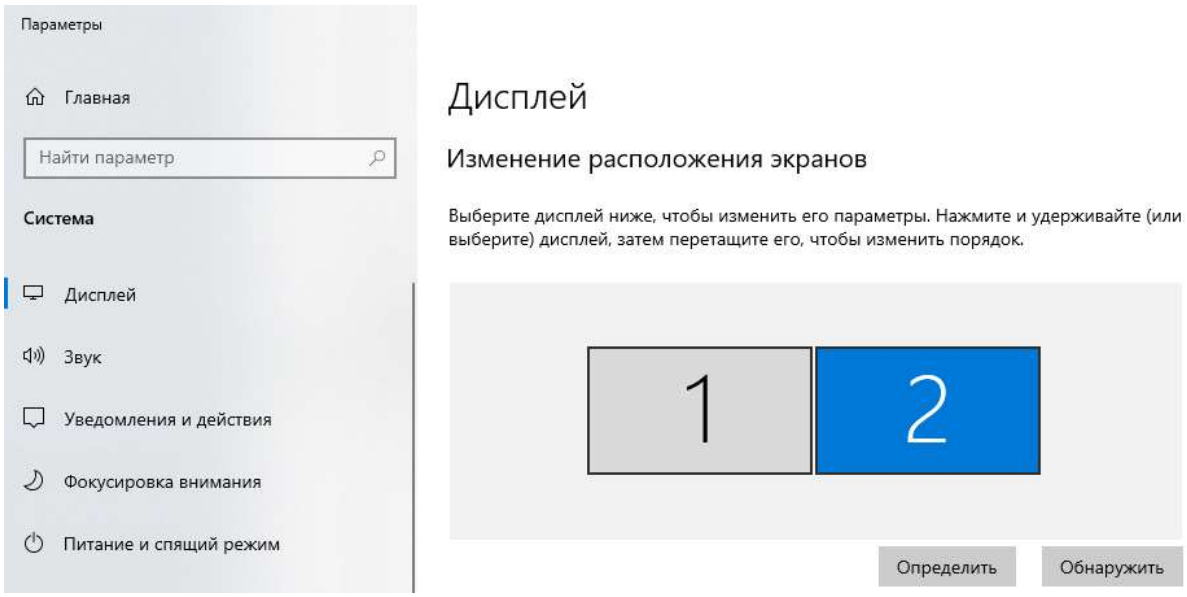

Рисунок 9 – проверка наличия 2-го дисплея

Ну и наконец, можно использовать программы захвата экрана (такие как OBS) для взаимодействия с новым экраном (рис. 10).

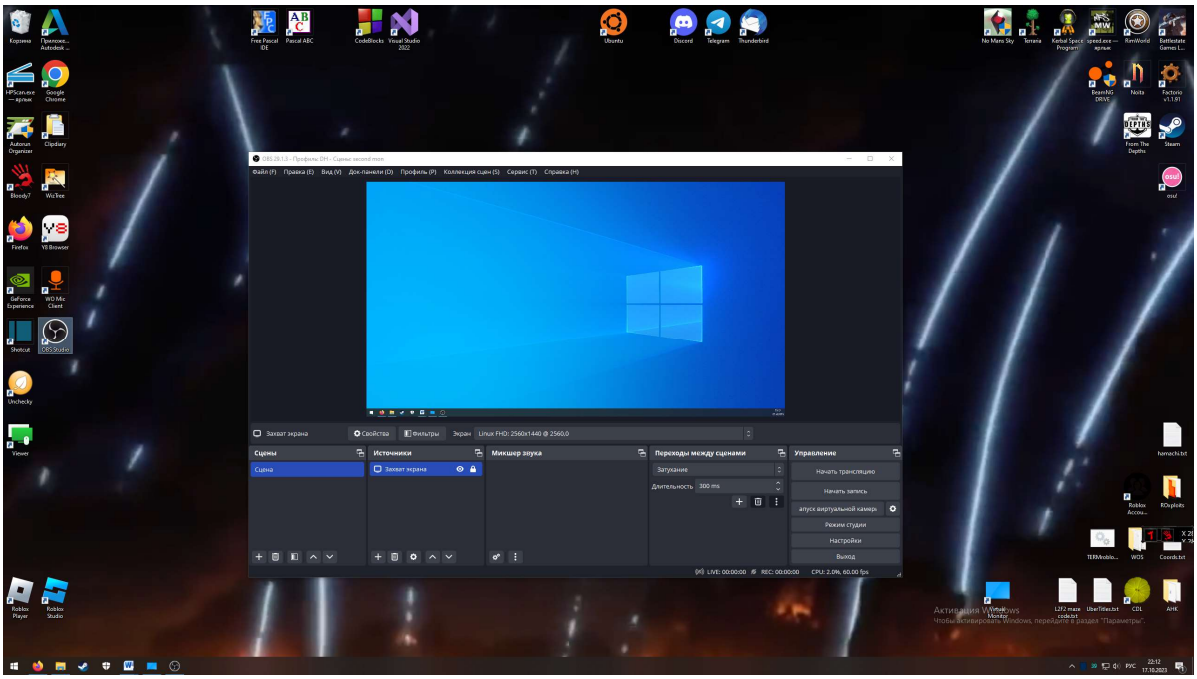

Рисунок 10 – обнаружение и проецирование 2-го дисплея программой OBS

Инструкция по использованию представлена в Приложении А.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта был разработан драйвер виртуального монитора для операционной системы Windows 10. Результаты данного проекта позволили углубить знания в области разработки драйверов и программного обеспечения, а также приобрести ценный опыт взаимодействия с аппаратными компонентами системы.

Основные достижения в рамках данного проекта включают:

1. **Разработка драйвера**: Был создан эффективный и гибкий драйвер, способный эмулировать работу дополнительного монитора в ОС Windows;
2. **Изучение операционной системы**: Работа над проектом потребовала глубокого понимания внутреннего устройства операционной системы Windows, а также принципов работы с драйверами;
3. **Навыки разработки на низком уровне**: Разработка драйвера позволила освоить навыки работы на более низком уровне абстракции, что является важным компонентом профессионального роста в области разработки ПО.

Таким образом, выполнение данного курсового проекта не только позволило успешно реализовать поставленную цель, но и обогатило набор профессиональных навыков, необходимых в сфере разработки программного обеспечения под операционную систему Windows.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

- Драйвер — это просто / Хабр: https://habr.com/ru/articles/145926/ (дата доступа: 28.10.2023)
- Что такое драйвер? - Windows drivers | Microsoft Learn: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/gettingstarted/what-is-a-driver- (дата доступа: 28.10.2023)
- Пользовательский режим и режим ядра - Windows drivers | Microsoft Learn: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode (дата доступа: 28.10.2023)
- Написание первого драйвера - Windows drivers | Microsoft Learn: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/gettingstarted/writing-your-first-driver (дата доступа: 28.10.2023)
- Написание универсального драйвера Windows (UMDF 2) на основе шаблона - Windows drivers | Microsoft Learn: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/gettingstarted/writing-a-umdf-driver-based-on-a-template (дата доступа: 28.10.2023)
- Написание Hello World драйвера Windows (KMDF) - Windows drivers | Microsoft Learn: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/gettingstarted/writing-a-very-small-kmdf--driver (дата доступа: 28.10.2023)
- Написание универсального драйвера Windows (KMDF) на основе шаблона - Windows drivers | Microsoft Learn: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/gettingstarted/writing-a-kmdf-driver-based-on-a-template (дата доступа: 28.10.2023)
- Подготовка компьютера для развертывания и тестирования драйверов (WDK 10) - Windows drivers | Microsoft Learn: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/gettingstarted/provision-a-target-computer-wdk-8-1 (дата доступа: 28.10.2023)
- Multiple-Monitor Support in the Display Driver - Windows drivers | Microsoft Learn: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/multiple-monitor-support-in-the-display-driver (дата доступа: 28.10.2023)
- Driver design guides for display, graphics, and compute accelerator devices - Windows drivers | Microsoft Learn: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/ (дата доступа: 28.10.2023)
- Road map for the Windows Display Driver Model (WDDM) - Windows drivers | Microsoft Learn: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/roadmap-for-developing-drivers-for-the-windows-vista-display-driver-mo (дата доступа: 28.10.2023)
- Initializing Display Miniport and User-Mode Display Drivers - Windows drivers | Microsoft Learn: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/initializing-display-miniport-and-user-mode-display-drivers (дата доступа: 28.10.2023)

- Installing Test-Signed Driver Packages - Windows drivers | Microsoft Learn: https://learn.microsoft.com/en-us/windows-hardware/drivers/install/installing-test-signed-driver-packages (дата доступа: 28.10.2023)
- Загрузка тестового подписанного кода - Windows drivers | Microsoft Learn: https://learn.microsoft.com/ru-ru/windows-hardware/drivers/install/the-testsigning-boot-configuration-option (дата доступа: 28.10.2023)
- Indirect display driver model overview - Windows drivers | Microsoft Learn: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/indirect-display-driver-model-overview (дата доступа: 28.10.2023)

ПРИЛОЖЕНИЕ А

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

КАФЕДРА ИНТЕЛЛЕКТУАЛЬНЫХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ДРАЙВЕР ВИРТУАЛЬНОГО МОНИТОРА ДЛЯ ОС WINDOWS

ИНСТРУКЦИЯ ПО УСТАНОВКЕ И ПРОВЕРКЕ IDD

КП.ПО-9.1-40-01-01

Листов 3

Руководитель И. Д. Козик

Выполнил А. С. Мисиюк

Консультант по ЕСПД И. Д. Козик

2023

Готовые к использованию файлы проекта можно найти на репозитории https://github.com/11ALX11/SP_kurz в релизах (https://github.com/11ALX11/SP_kurz/releases), где следует выбирать последнюю доступную версию. Тут вы найдете архив, содержащий необходимые файлы. Драйвер расчитан на **Windows 10+** (в теории с Windows 8).

Перед использованием нужно подписать и установить драйвер (рис. 1 и рис. 2):

- С правами админа запустите .bat;
- Файл .inf нужно установить.

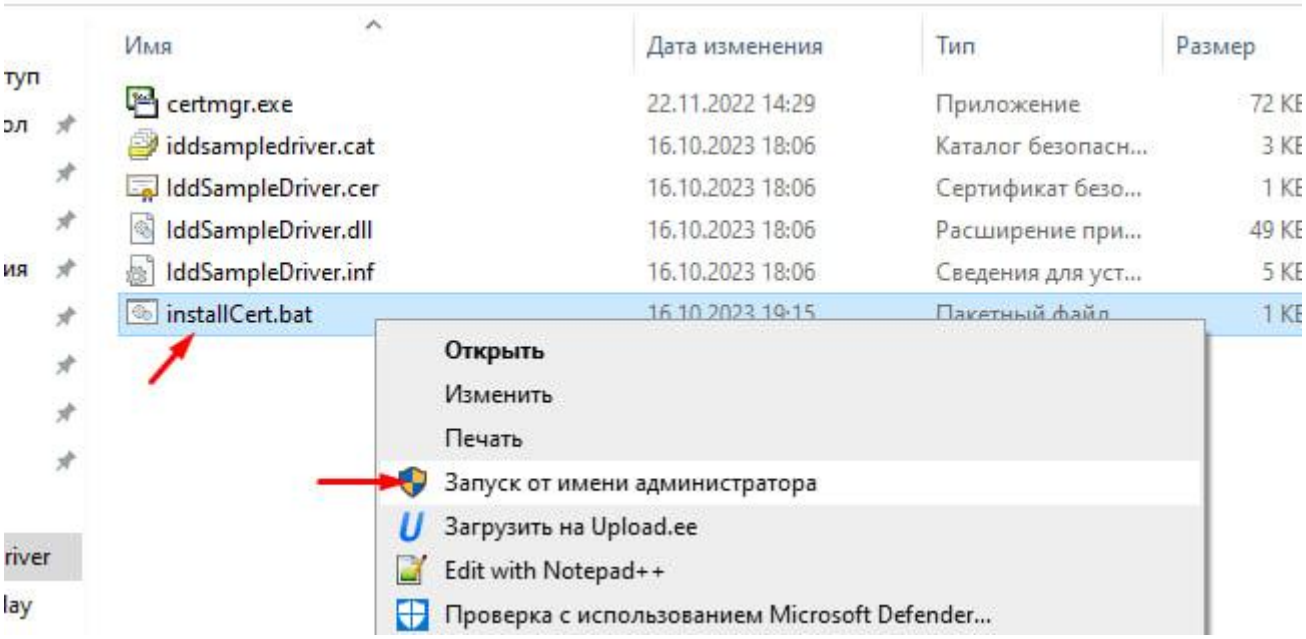

Рисунок 1 – запуск от имени администратора скрипт подписи драйвера

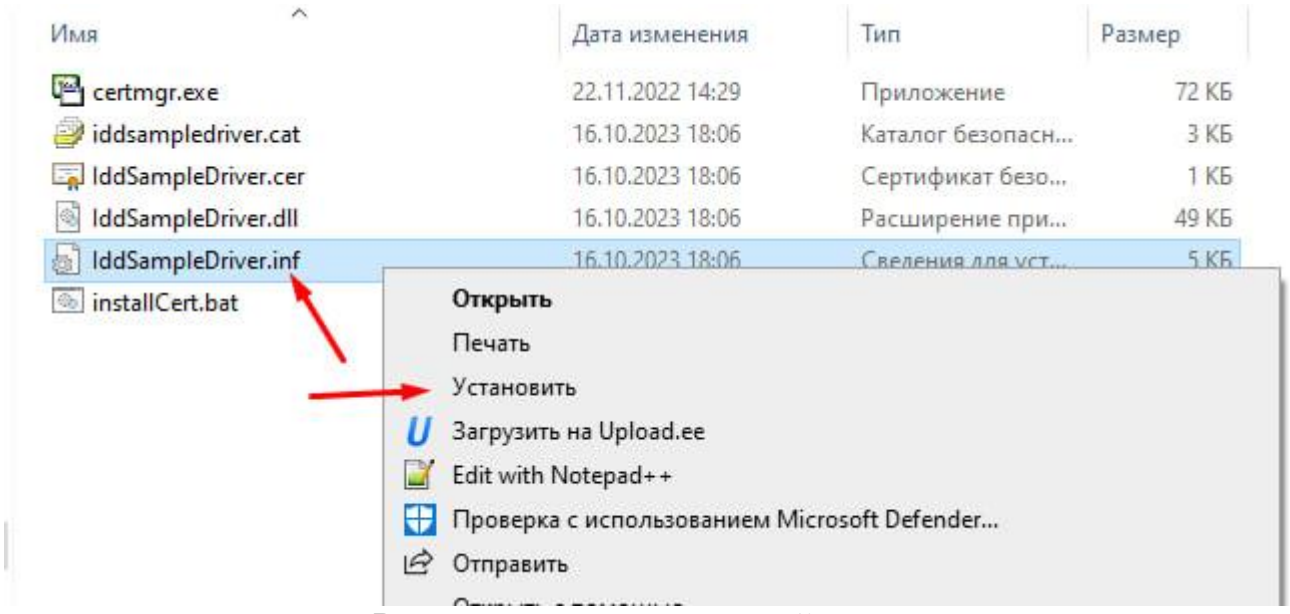

Рисунок 2 – установка драйвера

С помощью *IddSampleApp.exe* можно временно добавить монитор. (запуск от имени администратора).

Альтернативно, можно установить видеоадаптер (на основе драйвера). Зайдите в **диспетчере устройств**. Кликните по любому устройству, выберите в панели меню (сверху) **Действие -> Установить старое устройство**. Дальше нас интересует **установка вручную** (но на самом деле без разницы). Дальше ищем в списке **Видеоадаптеры**. Теперь выбираем **Установить с диска** и ищем наш **.inf** драйвер.

Важно!

Если есть желание убрать виртуальный монитор после такой установки, в **диспетчере устройств** -> **видеоадаптеры** ищите **IddSampleDriver Device**. **Удалите** этот фиктивный адаптер (пункт **Удалить устройство**).

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;
}
```