Министерство образования и науки РФ
Санкт-Петербургский Политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

Курсовая работа

по дисциплине «Программирование драйверов периферийных устройств»

Выполнил студент
Группы 5130904/10101

Абраамян А. М.

Преподаватель

Иночкин Ф. М.

Санкт-Петербург
2024

Оглавление

Постановка задачи

Вариант 1. Драйвер-передатчик последовательного порта с циклическим опросом состояния в режиме ядра.

На базе примера «portio» WDK (winddk\src\general\portio). Драйвер устанавливает параметры порта (скорость, режим обмена) (обработчик EvtDeviceAdd). Программа в режиме пользователя передает блок данных драйверу при помощи API WriteFile. Драйвер получает блок данных (обработчик EvtIoWrite) и каждый байт блока данных записывается в регистр передатчика, готовность передачи перед отправкой байта опрашивается драйвером в непрерывном цикле. Запись/чтение из портов ввода-вывода можно реализовать при помощи ассемблерных вставок и команд in/out.

Исходные коды драйвера расположены в каталоге «sys», коды программы режима пользователя – в каталоге «gpdwrite».

Для отладки драйвера в виртуальной среде VirtualBox необходимо включить эмуляцию порта. Данные порта можно перенаправить в файл или воспользоваться программой эмуляции портов VSPE в хост-системе. В последнем случае потребуется создать структуру типа «мост», перенаправить вывод VirtualBox в первый порт, а ко второму подключить стандартную программу-терминал (например, стандартный HyperTerminal Windows).

Описание базового примера

Драйвер **Portio** — это пример драйвера, который показывает, как взаимодействовать с I/O портами на системе. Он предоставляет базовый способ работы с портами ввода-вывода, что полезно для взаимодействия с аппаратными устройствами, которые используют **порт I/O** (метод общения с устройствами через специфические адреса памяти, называемые I/O портами). Это может использоваться для работы с устаревшими устройствами, такими как **параллельные порты** или **серийные порты**.

В современных операционных системах доступ к I/O портам ограничен для приложений в **пользовательском режиме**, чтобы обеспечить безопасность и стабильность системы. Однако драйверы работают в **режиме ядра** и имеют необходимые привилегии для прямого доступа к этим ресурсам.

Основные шаги для понимания работы драйвера Portio:

1. **Обзор драйвера Portio**:
    - **Portio** — это простой пример драйвера, который демонстрирует, как выполнять базовые операции с I/O портами (например, чтение и запись данных в порты I/O) с использованием ядра Windows.
    - Драйвер работает в **режиме ядра** и использует такие функции, как `IoWritePortUchar` или `IoReadPortUchar` для отправки и получения данных с аппаратных устройств через I/O порты.
    - Обычно такие драйверы используют **порт I/O** для общения с **устаревшими** аппаратными устройствами, например, с **параллельными портами** или **серийными портами**.
2. **Точка входа в драйвер**:
    - Основной точкой входа для драйвера является функция `DriverEntry`, которая вызывается при загрузке драйвера в систему.
    - В этой функции обычно выполняются задачи инициализации, такие как выделение ресурсов, настройка объектов устройств и регистрация обработчиков для выполнения операций ввода-вывода.
3. **Обработка запросов ввода-вывода**:
    - Когда приложение в пользовательском режиме хочет взаимодействовать с аппаратным устройством, оно отправляет запрос на ввод-вывод в драйвер.
    - Запросы на ввод-вывод могут быть операциями чтения или записи в I/O порты.
    - Драйвер обрабатывает запросы, выполняя операции с портами ввода-вывода, используя такие функции как `IoReadPortUchar`, `IoWritePortUchar` и другие.
4. **Обработка доступа к I/O портам**:
    - **Portio** позволяет работать с аппаратными устройствами через **I/O порты** (обычно используя память, отображенную в адресное пространство). Эти порты являются специальными адресами памяти, которые напрямую связаны с конкретными аппаратными устройствами.
    - Для этого драйвер использует команды, такие как чтение/запись в память устройства или выполнение команд через шину данных.
5. **Пример работы драйвера Portio**:
    - Допустим, приложение хочет прочитать данные с устройства через параллельный порт.
    - Приложение вызывает API Windows для взаимодействия с драйвером (например, через `CreateFile` и `DeviceIoControl`).

- Запрос от приложения передается в драйвер, который выполняет операцию чтения с I/O порта устройства с помощью функции, например, `IoReadPortUchar`.
- Драйвер читает данные с порта и возвращает их обратно в приложение.
- Приложение получает данные и продолжает свою работу (например, отображает их на экране).

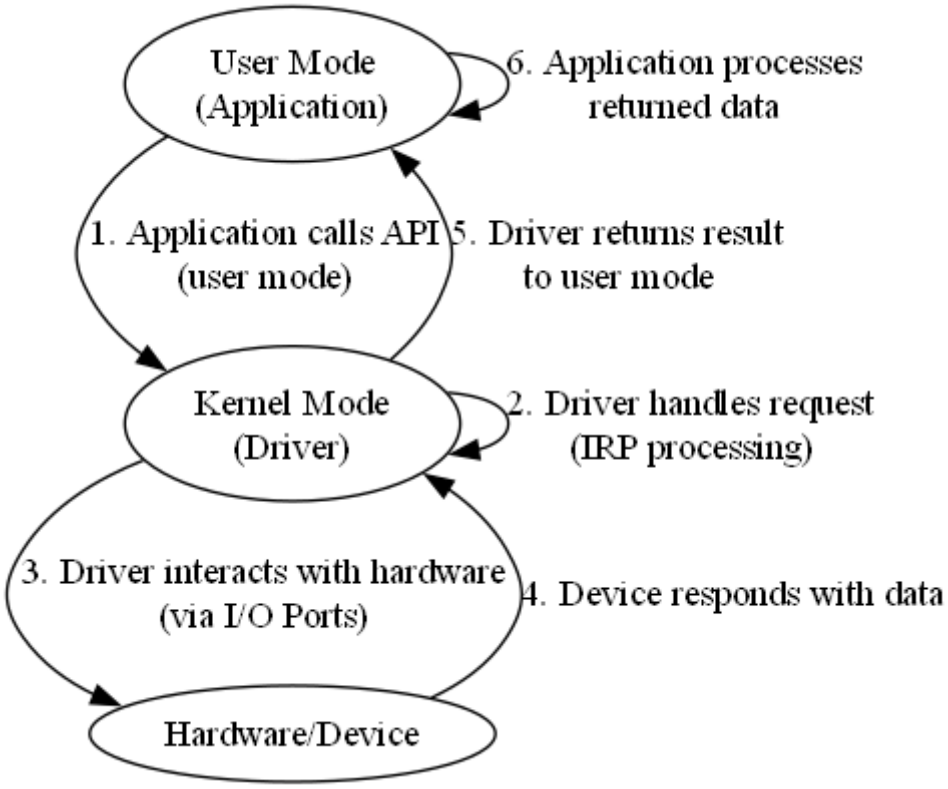

Рисунок 1 Диаграмма взаимодействия базового драйвера

Описание внесенных изменений

Основные отличия между исходным кодом и измененным кодом:

1. **Использование WDF и IOCTL**:
   - В **измененном коде** используется полноценная поддержка **IOCTL**, а также применяется **Windows Driver Framework (WDF)**. Это делает драйвер более гибким и безопасным в плане взаимодействия с устройствами. В отличие от исходного кода, где доступ к портам происходил напрямую через функции записи и чтения, в **измененном коде** добавлен механизм **IOCTL**, который позволяет драйверу получать команды и данные от приложений в пользовательском режиме.
2. **Структурированная работа с данными**:
   - В **измененном коде** добавлена структура **DEVICE_EXTENSION**, которая хранит данные в буфере и позволяет передавать их через IOCTL в устройство. Данные копируются в буфер устройства и обрабатываются функцией **CustomIoWrite**, которая затем передает их в серийный порт. В отличие от исходного кода, где данные передавались напрямую через операции с I/O портами, в **измененном коде** данные сначала обрабатываются через IOCTL и буферы, что делает код более безопасным и масштабируемым.
3. **Обработка IOCTL запросов**:

- В **измененном коде** добавлен обработчик **DeviceIoControlHandler**, который обрабатывает запросы от пользовательского режима. Когда приложение отправляет данные через **IOCTL**, драйвер получает эти данные в виде **SystemBuffer** и копирует их в буфер устройства. В исходном примере **Portio** таких запросов не было, и все операции происходили напрямую с I/O портами.

4. **Работа с IRP (I/O Request Packets)**:
   - В **измененном коде** используется структура **IRP** для обработки запросов на ввод-вывод. В частности:
     - **CreateCloseHandler** — обрабатывает запросы на создание/закрытие устройства.
     - **DeviceIoControlHandler** — обрабатывает запросы на устройство (например, отправку данных через COM порт).
   - В отличие от исходного примера **Portio**, где операции с портами происходили напрямую и без сложной обработки, в **измененном коде** обработка запросов происходит через IRP, что делает взаимодействие более безопасным и организованным.
5. **Интерфейс с пользовательским приложением**:
   - В **измененном коде** создается **символическая ссылка** с помощью `IoCreateSymbolicLink`, что позволяет пользовательским приложениям взаимодействовать с драйвером через **\DosDevices\IoctlDevice**. Это дает возможность приложению работать с драйвером в пользовательском режиме. В исходном коде **Portio** такого механизма не было, и взаимодействие с драйвером происходило напрямую через I/O порты без использования символических ссылок.
6. **Логирование и отладка**:
   - В **измененном коде** добавлено множество функций отладки с помощью **DbgPrint**, что позволяет отслеживать, что происходит в драйвере в реальном времени. Например, выводятся данные о процессе записи в порт и получении данных. В исходном коде **Portio** такого логирования не было, что делает **измененный код** более удобным для отладки и тестирования.

Основные этапы взаимодействия с COM портом в измененном коде:

1. **Настройка COM порта** (функция `ConfigureSerialPort`):
   - На самом начале драйвер конфигурирует серийный порт, например, COM2, с помощью функции `ConfigureSerialPort`.
   - В этом процессе выполняется несколько ключевых шагов, чтобы правильно настроить порт для передачи данных:
     - **Отключение прерываний**: Мы записываем в регистр `IER` (Interrupt Enable Register) значение `0x00`, чтобы отключить все прерывания для порта. Это необходимо, чтобы контролировать данные вручную.
     - **Настройка скорости передачи (baud rate)**: Для установки скорости передачи данных (например, 115200 бод) используется делитель (divisor). Для этого устанавливается **DLAB (Divisor Latch Access Bit)** через запись в регистр `LCR` (Line Control Register).
     - **Настройка параметров передачи данных**: Устанавливаются параметры: 8 бит данных, без четности, 1 стоп-бит. Это также настраивается через регистр `LCR`.
     - **Настройка FIFO (First In, First Out) буфера**: Включается FIFO для улучшения производительности. Это важно для обработки данных, если их много.

- **Включение передачи и приема данных**: Настроены модемные управляющие линии, чтобы разрешить передачу и прием данных.

Этот этап настраивает порт для дальнейшего использования — определяются параметры порта, такие как скорость передачи, количество бит данных и другие характеристики.

2. **Проверка готовности передатчика** (функция `IsTransmitterReady`):
   - После того как мы настроили COM порт, необходимо удостовериться, что передатчик готов к отправке данных.
   - В этой функции происходит чтение из **LSR (Line Status Register)**, который сообщает, готов ли передатчик отправить новый байт. Конкретно, проверяется бит **THRE** (Transmitter Holding Register Empty), который указывает, что передатчик пуст и готов принять новый байт.
   - Если бит **THRE** установлен (значение `0x20`), это означает, что передатчик готов для следующего байта данных.
3. **Ожидание готовности передатчика** (функция `WaitForTransmitterReady`):
   - В этой функции драйвер ждет, пока передатчик станет готовым к передаче данных. Если передатчик еще не готов (бит **THRE** не установлен), драйвер делает паузу с помощью функции `KeDelayExecutionThread`, чтобы не перегружать процессор бесконечными проверками.
   - Это важно для обеспечения синхронности — не нужно сразу пытаться записывать данные, если передатчик еще не готов.
4. **Запись данных в COM порт** (функция `CustomIoWrite`):
   - Когда передатчик готов, начинается процесс записи данных в COM порт.
   - В этой функции драйвер получает данные от пользователя через **IRP** (Input/Output Request Packet), который содержит буфер с данными, которые нужно отправить.
   - После этого данные передаются поочередно в порт. Для каждого байта:
     - Проверяется готовность передатчика.
     - Если передатчик готов, байт данных записывается в **COM2** через функцию `WRITE_PORT_UCHAR`, которая записывает данные в порт по его базовому адресу (например, `0x2F8` для COM2).
   - Этот процесс продолжается до тех пор, пока все данные не будут отправлены в порт.
5. **Передача данных по IOCTL**:
   - В **измененном коде** также используется механизм **IOCTL**, чтобы передавать данные в драйвер. Пользовательское приложение может отправить команду через **DeviceIoControl**, и драйвер получает данные через **IRP**.
   - Данные, которые передаются через IOCTL, сначала копируются в **DataBuffer** в структуре **DEVICE_EXTENSION**, а затем передаются через функцию **CustomIoWrite** в COM порт.

Подробное описание взаимодействия с COM портом:

1. **Настройка порта**:
   - В начале драйвер конфигурирует COM порт, устанавливая необходимые параметры (baud rate, количество бит, паритет и т. д.).
   - Это делается через запись в специальные порты управления и настройки, такие как регистры `LCR`, `IER`, `LSR` и другие.
2. **Отправка данных**:

- Когда драйвер получает данные через **IRP** (обычно от приложения в пользовательском режиме), он поочередно отправляет эти данные в серийный порт.
- Для этого каждый байт данных записывается в передатчик порта только после того, как передатчик станет готов (проверяется регистр состояния порта).

3. **Задержки и синхронизация**:
   - Для правильной синхронизации драйвер использует механизм ожидания готовности передатчика с помощью функции **WaitForTransmitterReady**. Это помогает избежать ошибок при передаче данных и гарантирует, что передатчик всегда готов принять новый байт.
4. **IOCTL для взаимодействия с пользовательским приложением**:
   - Пользовательское приложение может отправить запрос через **IOCTL**, и драйвер передаст данные в COM порт.
   - Этот механизм позволяет драйверу работать с пользовательскими приложениями, обеспечивая интерфейс для отправки данных в порт.

Решение

Подготовка

На основной машине

- WinDbg - С помощью него мы будем отлаживать при необходимости код и смотреть дебаг сообщения ядра
- Visual Studio - Нужен для компиляции кода для общения с драйвером
- VirtualBox - Запускаем винду XP
- VSCode - Что нибудь чтобы писать код

Виртуальная машина

После установки всего необходимого ПО – была проведена настройка образа виртуальной машины. Были выставлены следующие параметры для COM портов:

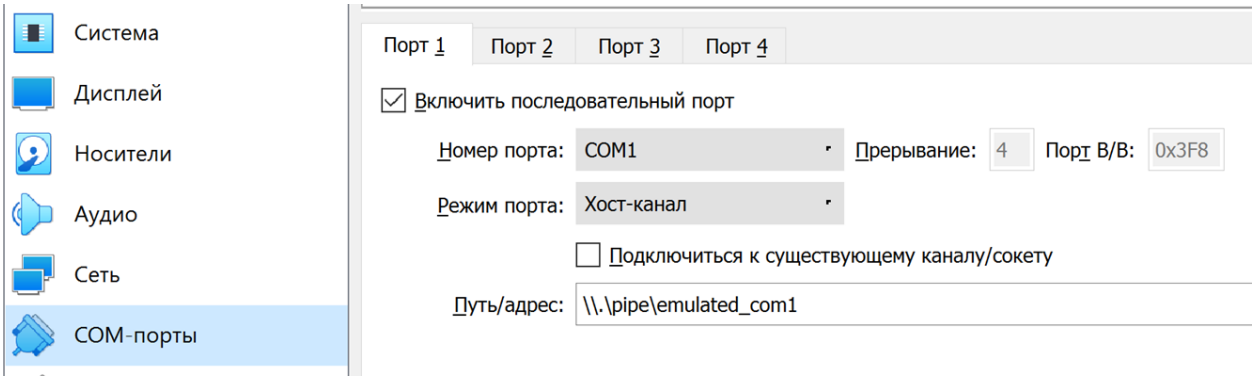

Рисунок 2 Настройка портов машины

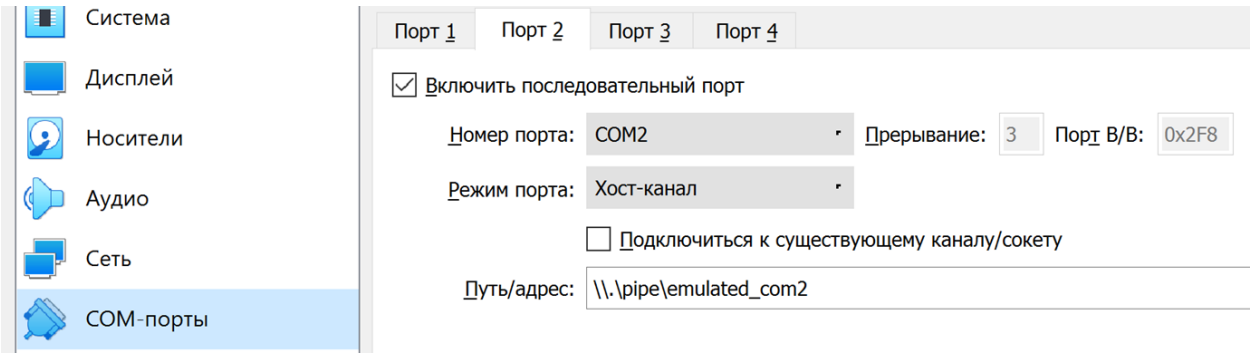

Рисунок 3 Настройка портов машины

Дальше настроили вывод дебаг информации системы в COM1 порт с частотой 115200. Запускаем WinDbg, перезапускаем систему и видим следующую картину:

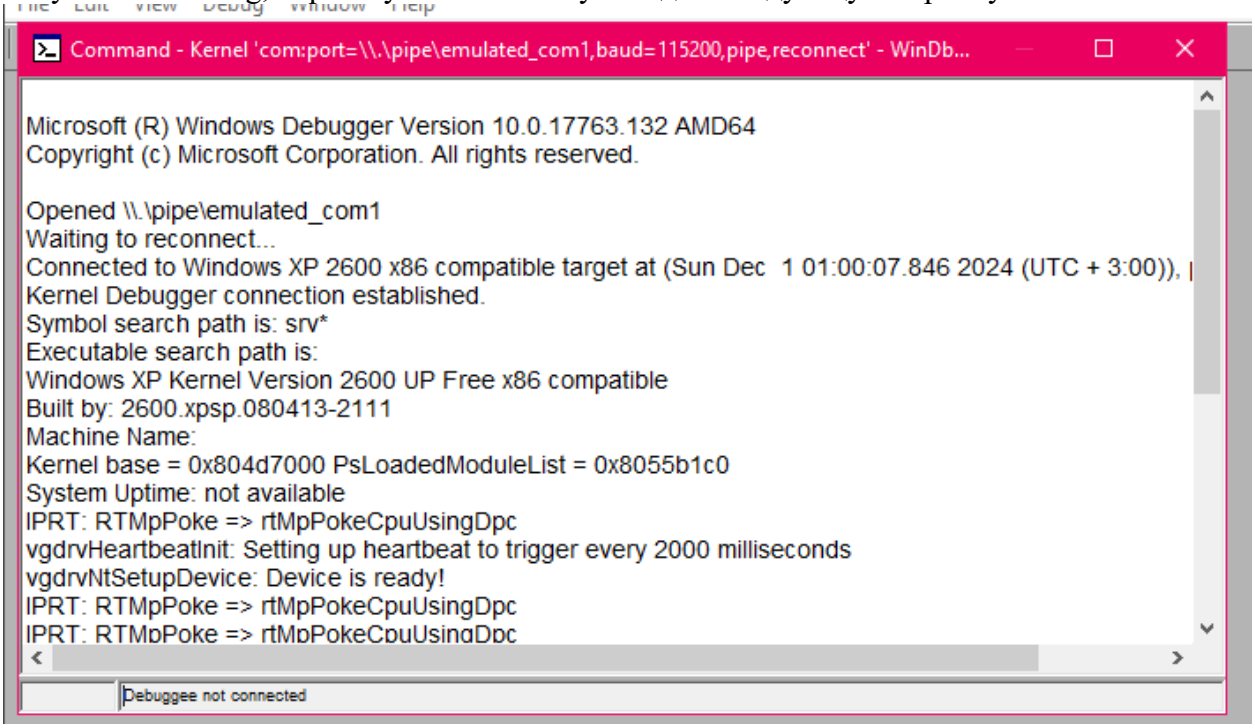

Рисунок 4 WinDbg

Собираем код

Создадим общую папку на виртуальной машине и поместим туда код. Перейдем туда и напишем команду bulid -ceZ для того чтобы собрать его.

При запуске наш драйвер создаст две сущности:

- \Device\IoctlDevice
- \DosDevices\IoctlDevice

Первое - это непосредственно имя нашего драйвера. Оно будет доступно только в режиме ядра - что значит пользователи не могут его использовать.

Второе - это символическая ссылка на наш драйвер. Пользователи должны работать именно с ней.

Загрузка драйвера

В диспетчере устройств удалим текущий драйвер COM порта от компании Microsoft и с помощью команды c:\WinDDK\7600.16385.0\tools\devcon\i386\devcon.exe INSTALL .\genport.inf "root\portio" установим наш. Заглядываем в WinDbg и видим сообщение

COM2 port configured successfully.

Проверка работы драйвера

Для проверки работы драйвера – нам понадобятся две программы:

- Скрипт на питоне который будет читать из \\.\pipe\emulated_com2
- Скомпилированная программа для winXP32 которая будет отправлять драйверу данные.

Для начала запускаем скрипт, следом запускаем вторую программу на виртуальной машине. В результате если всё правильно сделать – в консоли где запущен скрипт на питоне мы увидим сообщение посланное от нашей программы, в данном случае “Hello, Serial Port!”

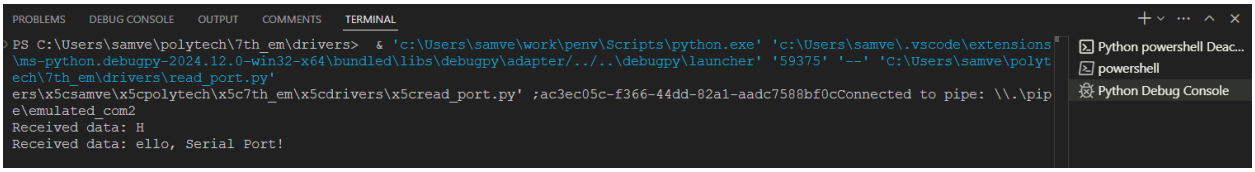

Рисунок 5 Чтение из COM порта

Приведем вывод из IrpTracker:

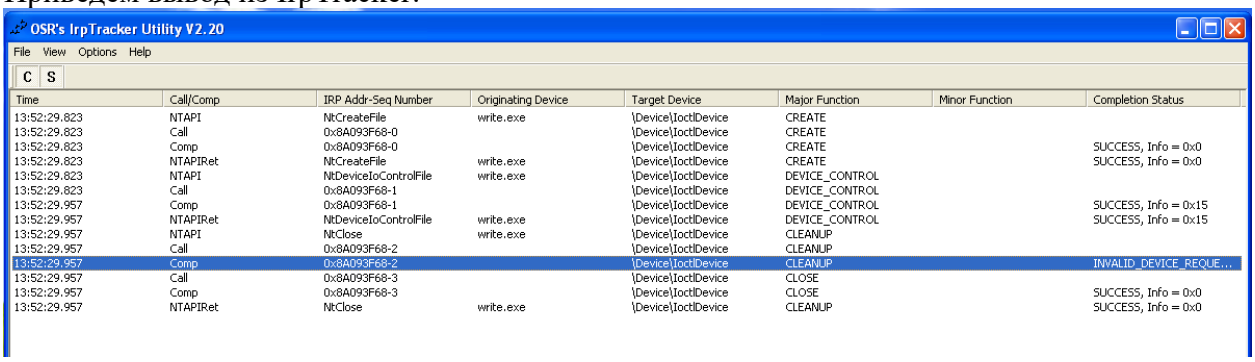

Рисунок 6 IrpTracker

Видим разные системные вызовы и общение нашей программы с драйвером, значит всё работает отлично

Структурная диаграмма

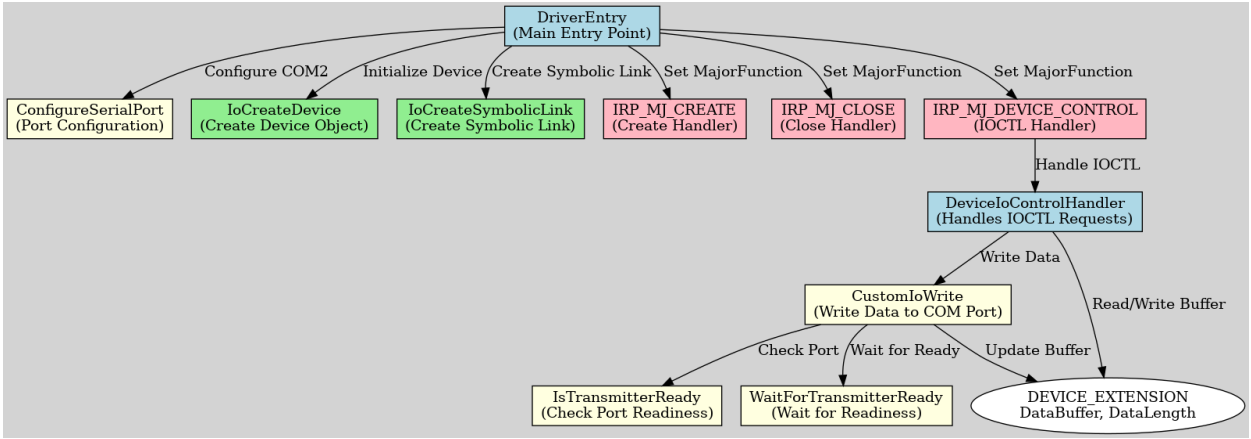

Диаграмма взаимодействия

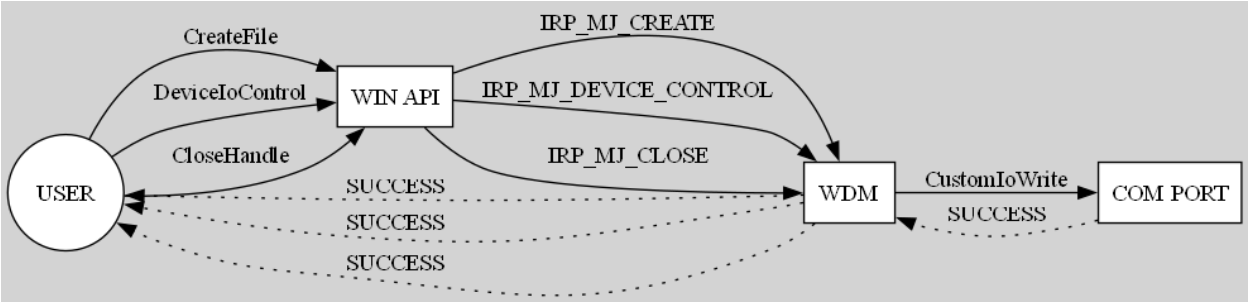

Рисунок 7 Диаграмма взаимодействия нового драйвера

Исходный код программ

Driver.c

```
#include <ntddk.h>
#include <wdf.h>

#define IOCTL_SEND_DATA_TO_PORT CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED,
FILE_WRITE_ACCESS)

#define MAX_BUFFER_SIZE 256
#define COM2_PORT_BASE_ADDRESS 0x2F8 // Example base address for COM2

VOID EvtIoWrite(
  __in PDEVICE_OBJECT DeviceObject,
  __in PIRP Irp,
  __in size_t Length
);

typedef struct _DEVICE_EXTENSION {
  CHAR DataBuffer[MAX_BUFFER_SIZE];
  ULONG DataLength;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

// Function to configure the serial port
NTSTATUS ConfigureSerialPort(USHORT PortBase)
{
  UCHAR divisor;
  NTSTATUS status;

  // Reset the COM port (disable interrupts)
  WRITE_PORT_UCHAR(PortBase + 4, 0x00);  // Disable interrupts (IER)

  // Set the baud rate - Divisor Latch Access Bit (DLAB = 1)
  WRITE_PORT_UCHAR(PortBase + 3, 0x80);  // Enable DLAB
  divisor = 1; // Example divisor for 115600 baud rate
  WRITE_PORT_UCHAR(PortBase, divisor);   // Set LSB of divisor
  WRITE_PORT_UCHAR(PortBase + 1, divisor); // Set MSB of divisor

  // Set 8 data bits, no parity, 1 stop bit
  WRITE_PORT_UCHAR(PortBase + 3, 0x03);  // 8 data bits, no parity, 1 stop bit
```

```
    // Enable FIFO for serial port
    WRITE_PORT_UCHAR(PortBase + 2, 0xC7);  // Enable FIFO, clear them, 14-byte threshold

    // Enable modem control
    WRITE_PORT_UCHAR(PortBase + 4, 0x0F);  // Enable both receiver and transmitter

    DbgPrint("COM2 port configured successfully.\n");

    return STATUS_SUCCESS;
}

// Check if the transmitter is ready
BOOLEAN IsTransmitterReady()
{
    UCHAR lsr;

    // Read the Line Status Register (LSR) from COM2 (PortBase + 5)
    lsr = READ_PORT_UCHAR(COM2_PORT_BASE_ADDRESS + 5);  // LSR is at offset 5

    // Check if the Transmitter Holding Register Empty (THRE) bit is set (bit 5)
    DbgPrint("LSR Register: 0x%x\n", lsr);
    return (lsr & 0x20) != 0;
}

// Wait for the transmitter to be ready to send data
NTSTATUS WaitForTransmitterReady()
{
    LARGE_INTEGER delayTime;
    NTSTATUS status;

    // Set a timeout for checking (e.g., 100 milliseconds)
    delayTime.QuadPart = -10000 * 100;  // 100ms in 100-nanosecond units

    DbgPrint("Waiting for transmitter to be ready...\n");

    while (!IsTransmitterReady()) {
        // Wait for the timeout period before checking again
        KeDelayExecutionThread(KernelMode, FALSE, &delayTime);
    }

    DbgPrint("Transmitter is ready.\n");
    return STATUS_SUCCESS;
}

// Write data to the serial port
VOID CustomIoWrite(
    __in PDEVICE_OBJECT DeviceObject,
    __in PIRP Irp,
    __in size_t Length
)
{
    PDEVICE_EXTENSION deviceExtension = (PDEVICE_EXTENSION)DeviceObject->DeviceExtension;
    NTSTATUS status = STATUS_SUCCESS;
    size_t bytesWritten = 0;
    PVOID buffer = NULL;

    DbgPrint("CustomIoWrite: Start writing to port\n");

    // Check that Length is valid and non-zero
    if (Length == 0) {
        DbgPrint("CustomIoWrite: Invalid length 0\n");
        WdfRequestComplete(Irp, STATUS_INVALID_PARAMETER);
```

```
        return;
    }

    // Retrieve the input buffer from the IRP (this is the user data sent to the driver)
    buffer = Irp->AssociatedIrp.SystemBuffer;
    if (buffer == NULL) {
        DbgPrint("CustomIoWrite: Buffer is NULL\n");
        WdfRequestComplete(Irp, STATUS_INVALID_PARAMETER);
        return;
    }

    // Start sending bytes one by one to the serial port
    for (bytesWritten = 0; bytesWritten < Length; bytesWritten++) {
        DbgPrint("CustomIoWrite: Writing byte %zu: 0x%02X\n", bytesWritten,
((PUCHAR)buffer)[bytesWritten]);

        // Wait for the transmitter to be ready
        status = WaitForTransmitterReady();
        if (!NT_SUCCESS(status)) {
            DbgPrint("CustomIoWrite: Transmitter not ready. Status: 0x%x\n", status);
            WdfRequestComplete(Irp, status);
            return;
        }

        // Send the byte to the serial port (COM2 for example)
        WRITE_PORT_UCHAR(COM2_PORT_BASE_ADDRESS, ((PUCHAR)buffer)[bytesWritten]);
        DbgPrint("CustomIoWrite: Sent byte 0x%x to COM2\n", ((PUCHAR)buffer)[bytesWritten]);
    }

    // Complete the IRP after all bytes are written
    DbgPrint("CustomIoWrite: Completed writing to serial port\n");
}

// IOCTL handler for sending data to serial port
NTSTATUS DeviceIoControlHandler(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
)
{
    PDEVICE_EXTENSION deviceExtension = (PDEVICE_EXTENSION)DeviceObject->DeviceExtension;
    PIO_STACK_LOCATION ioStack = IoGetCurrentIrpStackLocation(Irp);
    NTSTATUS status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG bytesToCopy = 0;

    // Check if the correct IOCTL code was used
    if (ioStack->Parameters.DeviceIoControl.IoControlCode == IOCTL_SEND_DATA_TO_PORT) {
        bytesToCopy = min(ioStack->Parameters.DeviceIoControl.InputBufferLength, MAX_BUFFER_SIZE);

        // Copy the input buffer to the device extension data buffer
        RtlCopyMemory(deviceExtension->DataBuffer, Irp->AssociatedIrp.SystemBuffer, bytesToCopy);
        deviceExtension->DataLength = bytesToCopy;

        // Debug: Print received data and length
        DbgPrint("DeviceIoControlHandler: Received %lu bytes, data: %.*s\n",
            bytesToCopy, bytesToCopy, deviceExtension->DataBuffer);

        // Pass data to the serial port write function
        CustomIoWrite(DeviceObject, Irp, bytesToCopy);
    } else {
        DbgPrint("DeviceIoControlHandler: Invalid IOCTL code\n");
        status = STATUS_INVALID_DEVICE_REQUEST;
    }
```

```
    status = STATUS_SUCCESS;

    // Inform the caller of the result and completion
    Irp->IoStatus.Information = bytesToCopy;
    Irp->IoStatus.Status = status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT); // Complete the IRP request

    return status;
}


// Create or close the device (dummy handler)
NTSTATUS CreateCloseHandler(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

// DriverEntry function (entry point)
NTSTATUS DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
)
{
    NTSTATUS status;
    UNICODE_STRING deviceName;
    UNICODE_STRING symbolicLinkName;
    WDFDEVICE device;
    PDEVICE_EXTENSION deviceExtension;

    // Initialize the device and symbolic link
    RtlInitUnicodeString(&deviceName, L"\\Device\\IoctlDevice");
    RtlInitUnicodeString(&symbolicLinkName, L"\\DosDevices\\IoctlDevice");

    // Create the device object
    status = IoCreateDevice(
        DriverObject,
        sizeof(DEVICE_EXTENSION),
        &deviceName,
        FILE_DEVICE_UNKNOWN,
        0,
        FALSE,
        &device
    );

    if (!NT_SUCCESS(status)) {
        return status;
    }

    // Configure the serial port (COM2)
    status = ConfigureSerialPort(COM2_PORT_BASE_ADDRESS);
    if (!NT_SUCCESS(status)) {
        IoDeleteDevice(device);
        return status;
    }

    // Initialize device extension
```

```
    deviceExtension = (PDEVICE_EXTENSION)DriverObject->DeviceObject->DeviceExtension;
    deviceExtension->DataLength = 0;

    // Set up major function handlers
    DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateCloseHandler;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = CreateCloseHandler;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DeviceIoControlHandler;

    // Create the symbolic link for user-mode access
    status = IoCreateSymbolicLink(&symbolicLinkName, &deviceName);
    if (!NT_SUCCESS(status)) {
        IoDeleteDevice(device);
        return status;
    }

    return STATUS_SUCCESS;
}
```

Main.c

```
typedef struct IUnknown IUnknown;


#include <windows.h>
#include <stdio.h>

#define IOCTL_SEND_DATA_TO_PORT CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED,
FILE_WRITE_ACCESS)

int main() {
    HANDLE hDevice;
    DWORD bytesReturned;
    char buffer[256]; // Buffer for data to send to the driver

    // Open the device (this will communicate with the driver)
    hDevice = CreateFile(
        L"\\\\.\\IoctlDevice",       // The symbolic link to the driver
        GENERIC_READ | GENERIC_WRITE, // We need both read and write access
        0,                          // No sharing
        NULL,                       // Default security attributes
        OPEN_EXISTING,              // Open the existing device
        FILE_ATTRIBUTE_NORMAL,      // Normal file attributes
        NULL                        // No template file
    );

    // Check if the device was opened successfully
    if (hDevice == INVALID_HANDLE_VALUE) {
        printf("Failed to open device. Error: %lu\n", GetLastError());
        return 1;
    }
```

```
    // Prepare data to send to the serial port
    snprintf(buffer, sizeof(buffer), "Hello, Serial Port!\n");

    // Send data via IOCTL to the driver
    if (!DeviceIoControl(
        hDevice,                // Device handle
        IOCTL_SEND_DATA_TO_PORT,    // IOCTL code
        buffer,                 // Input buffer (data to send)
        strlen(buffer) + 1,       // Size of the data (including null terminator)
        NULL,                   // No output buffer
        0,                      // No output
        &bytesReturned,           // Bytes returned
        NULL                    // No overlapped structure
    )) {
        printf("DeviceIoControl failed. Error: %lu\n", GetLastError());
        CloseHandle(hDevice);
        return 1;
    }

    printf("Data sent to driver: %s\n", buffer);

    // Close the device handle
    CloseHandle(hDevice);

    return 0;
}
```

Read_port.py

```
import win32pipe
import win32file
import time

# Define the pipe name
pipe_name = r'\\.\pipe\emulated_com2'

def read_from_pipe():
    # Open the pipe
    try:
        pipe = win32file.CreateFile(
            pipe_name,              # Pipe name
            win32file.GENERIC_READ,    # Read access
            0,                      # No sharing
            None,                   # Default security attributes
```

```
            win32file.OPEN_EXISTING,   # Open existing pipe
            0,                         # No flags
            None                       # No template file
        )
    except Exception as e:
        print(f"Failed to open pipe: {e}")
        return

    print(f"Connected to pipe: {pipe_name}")

    # Continuously read from the pipe in a while loop
    while True:
        try:
            # Read data from the pipe (set buffer size)
            _, data = win32file.ReadFile(pipe, 4096)

            # if hr == 0:
            #     break

            print(f"Received data: {data.decode('utf-8', errors='ignore')}")

        except Exception as e:
            print(f"Error reading from pipe: {e}")
            break

        time.sleep(1)  # Sleep for a short period to prevent high CPU usage

if __name__ == "__main__":
    read_from_pipe()
```

Источники информации

1. Microsoft [Электронный ресурс]/ Функция ZwCreateFile function (wdm.h) https://learn.microsoft.com/ru-ru/windows-hardware/drivers/ddi/wdm/nf-wdmzwcreatefile (дата обращения 17.12.2023)

2. Microsoft [Электронный ресурс]/ Функция ZwOpenFile (wdm.h) https://learn.microsoft.com/ru-ru/windows-hardware/drivers/ddi/wdm/nf-wdmzwopenfile (дата обращения 17.12.2023)

3. Microsoft [Электронный ресурс]/ Функция ZwReadFile (wdm.h) https://learn.microsoft.com/ru-ru/windows-hardware/drivers/ddi/wdm/nf-wdm-zwreadfile (дата обращения 17.12.2023)

4. Microsoft [Электронный ресурс]/ Функция DeviceIoControl (wdm.h) https://learn.microsoft.com/ru-ru/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol (дата обращения 17.12.2023)