Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский политехнический университет Петра Великого

Высшая школа программной инженерии

Институт компьютерных наук и кибербезопасности

**КУРСОВОЙ ПРОЕКТ**

**Вариант №5. Драйвер-приемник последовательного порта c возможностью установки скорости передачи, взаимодействие с программой по методу DeviceIoControl и ReadFile.**

по дисциплине «Программирование драйверов периферийных устройств»

Выполнил студент гр.

5130904/10103 Колосовская А.А.

Руководитель

Иночкин Ф. М

**«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.**

Санкт-Петербург

2024

## **Постановка задачи**

На базе примера «portio» WDK (winddk\src\general\portio). Программа в режиме пользователя передает драйверу параметры работы порта при помощи API DeviceIoControl. Драйвер получает запрос (обработчик EvtIoDeviceControl) и устанавливает параметры порта. Программа пользователя считывает байт данных при помощи API ReadFile. Драйвер получает запрос (обработчик EvtIoRead) и считывает регистр приемника, флаг наличия принятых данных. В случае наличия данных запрос завершается с количеством считанных байт = 1, байт данных при этом записывается в поле дынных запроса. В случае отсутствия принятых данных, запрос завершается с

количеством считанных байт = 0 Запись/чтение из портов ввода-вывода можно

реализовать при помощи ассемблерных вставок и команд in/out.

Чтобы организовать взаимодействие с пользовательской программой, понадобится выделить IOCTL-код (см. GpIoctl.h) и добавить соответствующую обработку IOCTL-кода в обработчик EvtIoDeviceControl. Из пользовательской программы передача запроса осуществляется при помощи API DeviceIoControl с тем же IOCTL-кодом.

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**

Portio — это пример драйвера Windows, который демонстрирует низкоуровневое взаимодействие с портами ввода-вывода (I/O). Этот драйвер позволяет выполнять базовые операции с I/O портами, такие как чтение и запись данных, предоставляя разработчикам возможность работать напрямую с аппаратным обеспечением через режим ядра. Это особенно полезно для взаимодействия с устаревшими устройствами, использующими такие интерфейсы, как параллельные или серийные порты. Доступ к I/O портам ограничен в пользовательском режиме операционной системы из соображений безопасности и стабильности. Драйверы, работающие в режиме ядра, получают привилегии, необходимые для прямого обращения к аппаратным портам и выполнения связанных операций.

**Основные аспекты работы драйвера Portio**

1. Обзор драйвера Portio

* Portio — это пример драйвера, демонстрирующий выполнение операций с I/O портами: чтение и запись данных.
* Драйвер работает в режиме ядра (kernel-mode) и использует функции ядра Windows, такие как IoWritePortUchar и IoReadPortUchar, для взаимодействия с аппаратным обеспечением.
* Основные сценарии использования включают работу с устройствами, которые не поддерживают современные протоколы общения, такими как USB, и зависят от прямого доступа к I/O портам.

2. Точка входа в драйвер

* Точка входа в драйвер — функция DriverEntry, которая вызывается при загрузке драйвера в операционную систему.
* В этой функции выполняются:
  + Инициализация драйвера, включая настройку устройства.
  + Регистрация объектов, таких как обработчики запросов ввода-вывода.
  + Выделение необходимых ресурсов и проверка их доступности.

3. Обработка запросов ввода-вывода

* Приложения пользовательского режима взаимодействуют с драйвером через стандартные механизмы Windows, такие как функции CreateFile, DeviceIoControl, или WriteFile.
* Когда приложение отправляет запрос в драйвер, он обрабатывает его через зарегистрированные функции. Например:
  + Запись в порт может быть выполнена с помощью IoWritePortUchar.
  + Чтение данных из порта — через IoReadPortUchar.
* Результат выполнения запроса передается обратно в приложение.

4. Обработка доступа к I/O портам

* Драйвер Portio предоставляет доступ к I/O портам, представляющим собой специальные адреса памяти, которые напрямую связаны с аппаратными устройствами.
* Основные команды для работы с портами:
  + Чтение данных — драйвер считывает информацию с устройства, используя I/O команды.
  + Запись данных — драйвер отправляет команды или данные устройству.
* Для взаимодействия с аппаратным обеспечением драйвер использует шины данных (например, PCI, ISA) или адресное пространство памяти.

**Пример работы драйвера Portio**

Сценарий: Приложение хочет получить данные с устройства через параллельный порт.

1. Отправка запроса:
   * Приложение пользовательского режима вызывает системные API Windows, например, CreateFile для открытия соединения с драйвером.
   * Через DeviceIoControl приложение отправляет запрос на чтение данных с устройства.
2. Обработка драйвером:
   * Запрос приложения передается драйверу Portio.
   * Драйвер считывает данные с порта устройства, используя функцию IoReadPortUchar.
3. Возврат результата:
   * Полученные данные возвращаются драйвером в приложение.
   * Приложение обрабатывает данные (например, выводит их пользователю или записывает в файл).

**Почему драйверы I/O портов важны**

* Наследие аппаратного обеспечения: Множество старых, но критически важных устройств (например, промышленные контроллеры) продолжают использовать I/O порты.
* Простота доступа: Работа с I/O портами через драйверы дает разработчику простой интерфейс для взаимодействия с устройствами, абстрагируя сложности.
* Безопасность: Драйверы Portio позволяют обрабатывать команды через интерфейс ядра, предотвращая случайный или злонамеренный доступ к аппаратным ресурсам.

Таким образом, драйвер Portio служит отличным примером для начинающих разработчиков драйверов, помогая изучить: основы работы с I/O портами, взаимодействие между пользовательским и ядром системы, а также реализацию простых сценариев ввода-вывода в Windows.

## **Описание внесенных изменений**

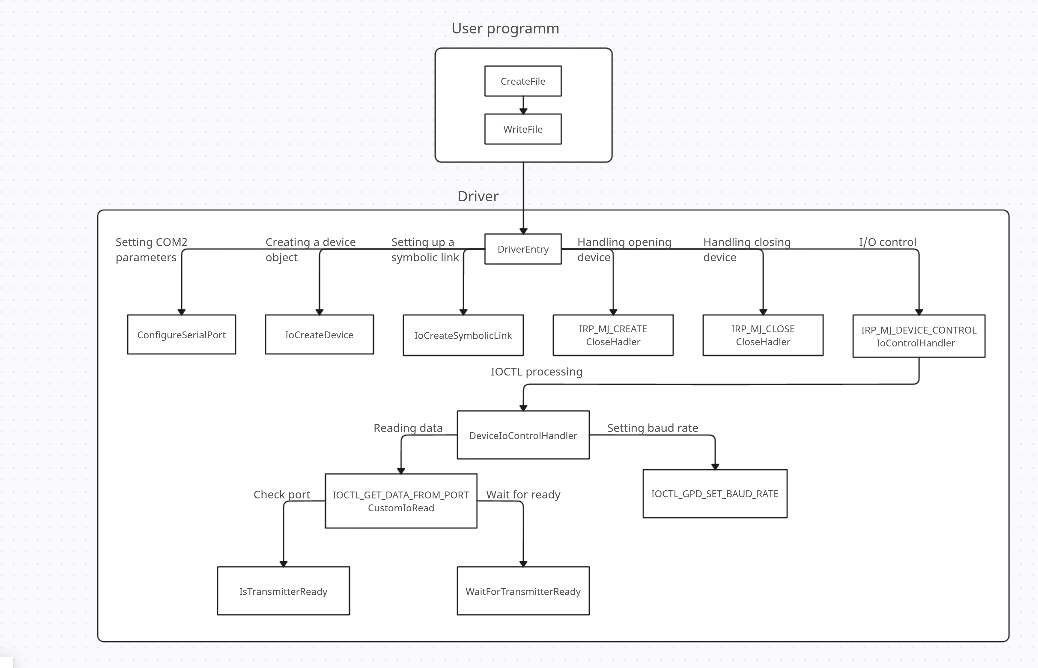


Рисунок 1. Структурная схема

**1. Использование WDF и IOCTL**

Код использует Windows Driver Framework для упрощения разработки драйвера и IOCTL-коды для передачи команд от приложений в пользовательском режиме к драйверу. IOCTL-коды, такие как IOCTL\_GPD\_SET\_BAUD\_RATE и IOCTL\_GET\_DATA\_FROM\_PORT, обеспечивают стандартизированный интерфейс для взаимодействия с драйвером, позволяя настраивать скорость передачи данных и инициировать операции чтения данных с устройства. Это позволяет отделить высокоуровневую логику приложений от низкоуровневой работы с оборудованием, обеспечивая гибкость и совместимость.

**2. Структура DEVICE\_EXTENSION**

В коде используется структура DEVICE\_EXTENSION, которая служит для хранения данных, специфичных для каждого устройства, созданного драйвером. Она содержит поля BaudRate для сохранения текущей скорости передачи данных, DataLength для отслеживания объема данных, доступных для передачи, и DataBuffer для временного хранения данных, передаваемых между приложением и устройством. Такая структура обеспечивает сохранение состояния устройства между вызовами функций и упрощает управление данными, гарантируя изоляцию устройства от других процессов.

**3. Работа с IRP (I/O Request Packet)**

IRP используется как стандартный механизм передачи запросов на ввод-вывод между операционной системой, драйвером и приложениями. Обработка запросов осуществляется через функции IRP\_MJ\_CREATE, IRP\_MJ\_CLOSE и IRP\_MJ\_DEVICE\_CONTROL, которые обрабатывают операции открытия, закрытия устройства и выполнения конкретных команд. Это централизует обработку операций, улучшая структуру драйвера и упрощая управление потоками данных.

**4. Создание устройства и символической ссылки**

Через функции IoCreateDevice и IoCreateSymbolicLink создаются устройство и символическая ссылка, такие как \\.\GpdDev, которые предоставляют приложениям удобный интерфейс для взаимодействия с драйвером. Приложения используют символическую ссылку для обращения к устройству через стандартные функции API, такие как CreateFile и DeviceIoControl, что упрощает интеграцию драйвера в пользовательские программы.

**5. Работа с I/O портами**

Драйвер работает с I/O портами, используя функции READ\_PORT\_UCHAR и WRITE\_PORT\_UCHAR, которые позволяют читать и записывать данные в порты. Эти операции используются для взаимодействия с аппаратным обеспечением, например, для проверки состояния устройства через регистры Line Status Register (LSR). Прямое обращение к портам позволяет реализовывать низкоуровневые функции, необходимые для работы с устройством.

**6. Управление скоростью передачи (Baud Rate)**

Функция ConfigureSerialPort выполняет настройку параметров последовательного порта, включая скорость передачи данных (baud rate), число стоп-битов и формат данных. Это достигается записью значений в регистры порта через I/O операции. Настройка скорости передачи данных позволяет адаптировать драйвер к различным устройствам, обеспечивая совместимость и гибкость.

**7. Проверка готовности передатчика**

Перед началом передачи данных драйвер проверяет состояние устройства с помощью функций IsTransmitterReady и WaitForTransmitterReady. Это исключает возможность потери данных из-за того, что устройство занято выполнением другой операции. Проверка осуществляется через анализ битов регистра LSR, что гарантирует надежность операций ввода-вывода.

**8. Обработка запросов ввода-вывода**

Функция CustomIoRead обрабатывает запросы на чтение данных с устройства. Она проверяет готовность устройства к передаче, считывает данные из порта и записывает их в буфер, предоставленный приложением. Это обеспечивает безопасную и эффективную передачу данных, необходимую для корректного функционирования приложения.

**9. Буферизация данных**

Для временного хранения данных используется буфер DataBuffer в структуре DEVICE\_EXTENSION. Этот буфер минимизирует количество операций ввода-вывода, улучшая производительность и обеспечивая целостность данных. Данные копируются в буфер перед отправкой или чтением, что облегчает управление потоками данных.

**10. Логгирование и отладка**

В коде активно используется DbgPrint для логгирования ключевых событий, таких как вызовы функций, обработка запросов и состояние устройства. Это упрощает диагностику, помогая отслеживать выполнение кода и выявлять возможные ошибки. Сообщения выводятся в WinDbg, что делает процесс отладки эффективным.

## **Подробное описание реализации**

Три файла — write\_port.py, main.c и driver.c — составляют систему, демонстрирующую взаимодействие между приложением пользовательского режима, драйвером устройства ядра Windows и виртуальным устройством, эмулирующим последовательный порт (COM-порт). Этот набор кода иллюстрирует полный цикл передачи данных, от отправки через пользовательское приложение до обработки и ответа драйвера.

**1. Python-скрипт write\_port.py**

Этот скрипт представляет собой клиент для передачи данных в драйвер через именованный канал (pipe). Он выполняет следующие шаги:

* Подключение к pipe: Скрипт пытается установить соединение с виртуальным COM-портом (\\.\pipe\emulated\_com2), повторяя попытки до 5 раз. Это необходимо для обеспечения устойчивости при временных сбоях подключения.
* Отправка данных: После подключения скрипт отправляет строку "SimulatedData" через канал, кодируя её в байты. Это демонстрирует отправку данных от приложения к драйверу.
* Обработка ошибок: Если возникают ошибки при открытии канала или записи данных, скрипт выводит диагностические сообщения, упрощая отладку.
* Закрытие канала: Скрипт корректно закрывает канал после завершения передачи, освобождая ресурсы.

**2. Программа на C (main.c)**

main.c представляет собой пользовательское приложение, которое взаимодействует с драйвером устройства через системные вызовы. Программа выполняет следующие функции:

* Открытие устройства: Программа подключается к драйверу, используя символическую ссылку (\\.\GpdDev). Это создаёт интерфейс для взаимодействия с драйвером.
* Настройка скорости передачи: С помощью вызова DeviceIoControl и IOCTL-кода IOCTL\_GPD\_SET\_BAUD\_RATE приложение отправляет запрос на установку скорости передачи данных (9600 бод) в драйвер. Это демонстрирует настройку параметров устройства.
* Чтение данных: Программа инициирует чтение данных с устройства, отправляя запрос через IOCTL-код IOCTL\_GET\_DATA\_FROM\_PORT. Полученные данные записываются в буфер и отображаются в консоли в шестнадцатеричном формате.
* Обработка ошибок: Если происходит сбой при взаимодействии с драйвером, программа регистрирует ошибку и выводит подробное сообщение для отладки.
* Закрытие устройства: После выполнения операций программа закрывает дескриптор устройства, завершая взаимодействие.

**3. Драйвер устройства (driver.c)**

driver.c является ядром системы, реализующим функционал взаимодействия с аппаратным уровнем через драйвер. Драйвер выполняет следующие задачи:

**Инициализация**

* Создание устройства: В функции DriverEntry драйвер создаёт устройство и регистрирует его с операционной системой, устанавливая символическую ссылку для доступа приложений.
* Регистрация обработчиков: Драйвер назначает обработчики для операций, таких как открытие, закрытие и управление устройством (IRP\_MJ\_CREATE, IRP\_MJ\_CLOSE, IRP\_MJ\_DEVICE\_CONTROL).

**Обработка запросов**

* Настройка порта: Драйвер обрабатывает IOCTL-запрос IOCTL\_GPD\_SET\_BAUD\_RATE в функции DeviceIoControlHandler. Для настройки COM-порта используется функция ConfigureSerialPort, которая настраивает регистры порта (скорость передачи, формат данных).
* WRITE\_PORT\_UCHAR((PUCHAR)(COM2\_PORT\_BASE\_ADDRESS + 4), 0x00);
  + Отключение всех прерываний через регистр IER (Interrupt Enable Register). Предотвращает обработку прерываний во время настройки порта, что делает процесс более безопасным.
* WRITE\_PORT\_UCHAR((PUCHAR)(COM2\_PORT\_BASE\_ADDRESS + 3), 0x80);
  + Включение режима DLAB (Divisor Latch Access Bit) через регистр LCR (Line Control Register), установка бита 7. Этот режим необходим для записи делителя частоты в регистры DLL и DLM.
* WRITE\_PORT\_UCHAR((PUCHAR)COM2\_PORT\_BASE\_ADDRESS, (UCHAR)(baudRate & 0xFF));
  + Запись младшего байта делителя частоты в регистр DLL (Divisor Latch Low Byte). Делитель частоты определяет скорость передачи данных, и младший байт должен быть настроен для корректной работы.
* WRITE\_PORT\_UCHAR((PUCHAR)(COM2\_PORT\_BASE\_ADDRESS + 1), (UCHAR)((baudRate >> 8) & 0xFF));
  + Запись старшего байта делителя частоты в регистр DLM (Divisor Latch High Byte). Старший байт делителя частоты дополняет настройку скорости передачи данных.
* WRITE\_PORT\_UCHAR((PUCHAR)(COM2\_PORT\_BASE\_ADDRESS + 3), 0x03);
  + Настройка параметров данных через регистр LCR (Line Control Register): 8 бит данных, 1 стоп-бит, отключение проверки чётности. Задаёт стандартные параметры для передачи данных.
* WRITE\_PORT\_UCHAR((PUCHAR)(COM2\_PORT\_BASE\_ADDRESS + 2), 0xC7);
  + Включение FIFO, очистка буферов, установка порога в 14 байт через регистр FCR (FIFO Control Register). Увеличивает производительность за счёт буферизации данных, снижая вероятность их потери.
* WRITE\_PORT\_UCHAR((PUCHAR)(COM2\_PORT\_BASE\_ADDRESS + 4), 0x0F);
  + Включение передатчика и приёмника через регистр MCR (Modem Control Register). Завершает настройку порта, делая его готовым для отправки и приёма данных.
* Чтение данных: Драйвер обрабатывает IOCTL-запрос IOCTL\_GET\_DATA\_FROM\_PORT с помощью функции CustomIoRead.
  + Проверяется готовность устройства к передаче данных через WaitForTransmitterReady и IsTransmitterReady.
  + Считывание выполняется по одному байту через READ\_PORT\_UCHAR, а данные передаются в буфер пользовательского приложения.

**Буферизация данных**

Данные, отправленные через pipe, копируются в буфер DataBuffer драйвера. Этот буфер управляется структурой DEVICE\_EXTENSION, хранящей текущую скорость передачи данных (BaudRate) и длину буфера (DataLength).

**Логгирование**

Драйвер активно использует DbgPrint для регистрации событий (например, вызов функций, успешность операций). Это упрощает отладку, позволяя разработчикам отслеживать выполнение кода в режиме ядра.

## **Подготовка системы**

Для того, чтобы всё работало корректно, необходимо установить следующие программы:

* **WinDbg** — инструмент для отладки кода и просмотра сообщений дебага ядра при необходимости.
* **Visual Studio** — необходим для компиляции кода, взаимодействующего с драйвером.
* **VirtualBox** — используется для запуска Windows XP в виртуальной машине.
* **VSCode** — редактор для написания кода.

**Виртуальная машина:**

После установки всего необходимого программного обеспечения была проведена настройка образа виртуальной машины. Для COM портов были заданы следующие параметры:

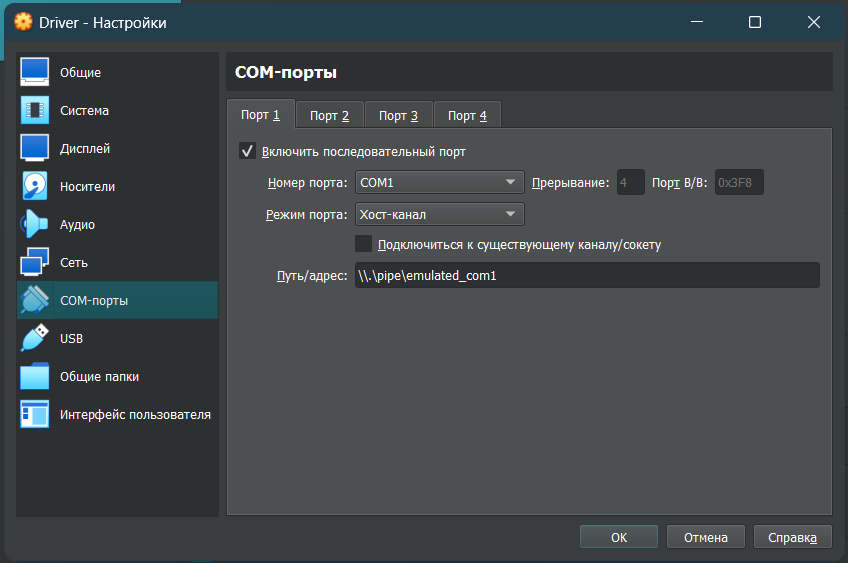


Рисунок 2. Настройка COM1

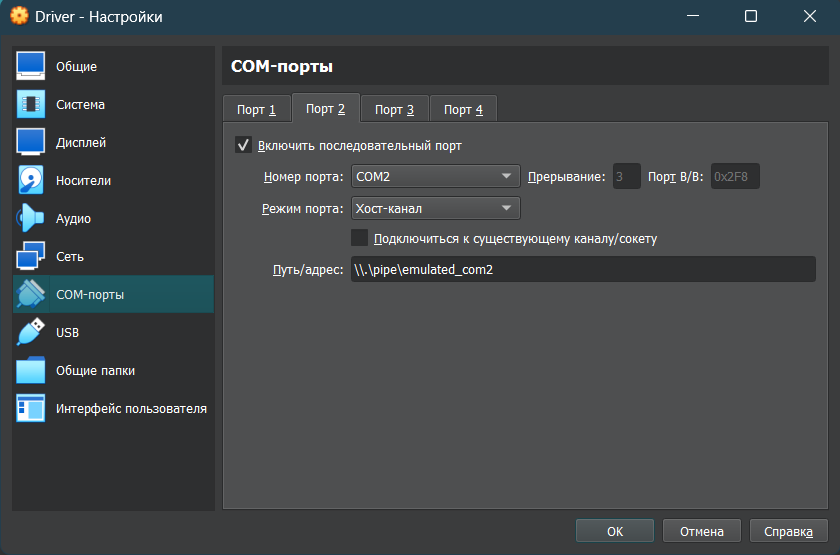


Рисунок 3. Настройка COM2

Затем был настроен вывод отладочной информации системы через COM1 в WinDbg:

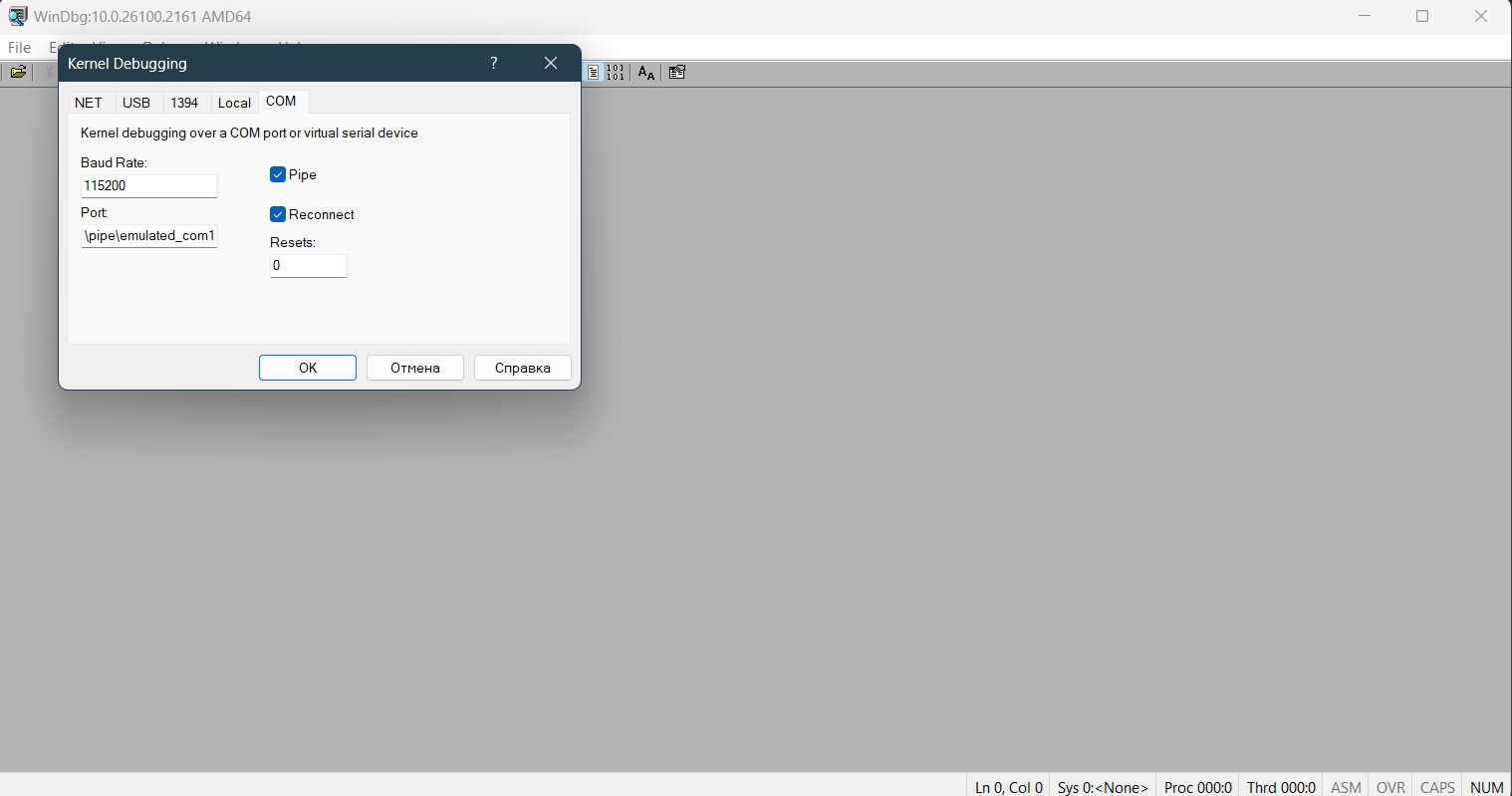


Рисунок 4. Настройка параметров WinDbg

В виртуальной машине в настройках системы нужно выставить значения ниже:

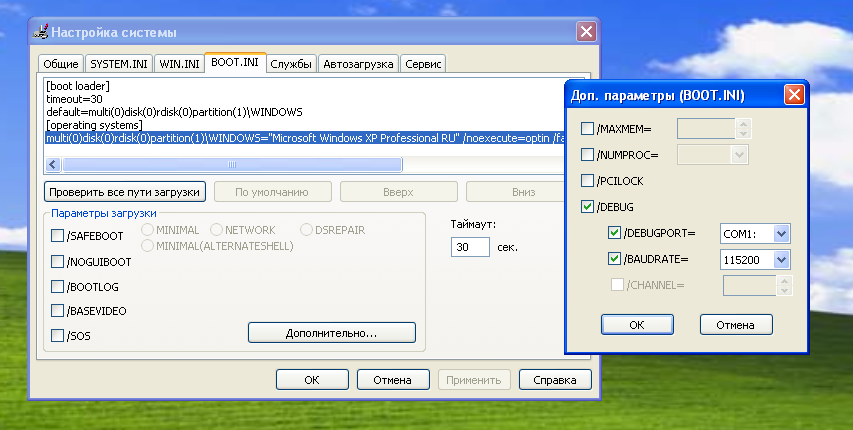


Рисунок 4. Настройка параметров WindowsXP

**Сборка кода**

Для сборки необходимо создать общую папку на виртуальной машине и поместить туда исходный код. После этого перейти в эту папку и выполнить команду build -ceZ для сборки.

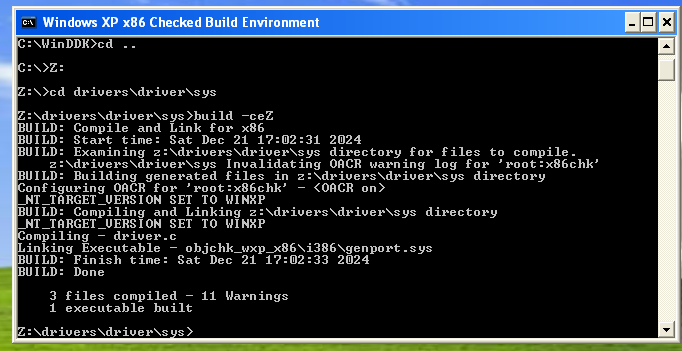


Рисунок 5. Сборка кода драйвера

**Загрузка драйвера**

Для загрузки драйвера переходим в c:\WinDDK\7600.16385.0\tools\devcon\i386 и выполняем команду devcon.exe INSTALL .\genport.inf "root\portio" (указываем путь до нашего созданного genport.inf после сборки кода)

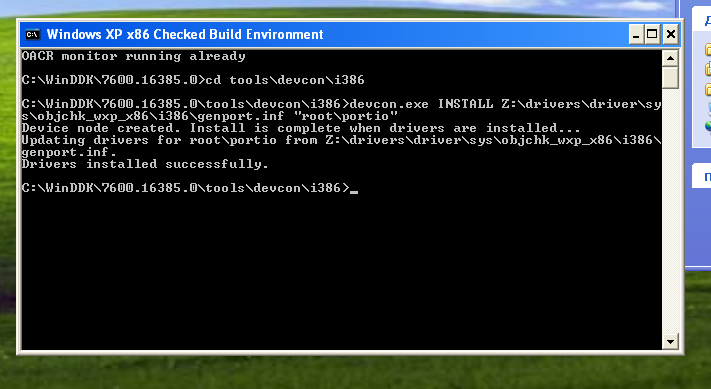


Рисунок 6. Сборка кода драйвера

**Проверка работы драйвера**

Для проверки работы драйвера использовались две программы:

* Скрипт на Python, который будет записывать данные в \\.\pipe\emulated\_com2.
* Скомпилированная программа для Windows XP32, которая будет считывать данные из пользовательского кода (write\_port.py).

Сначала необходимо запустить код write\_port.py, таким образом мы отправляем данные в наш драйвер:

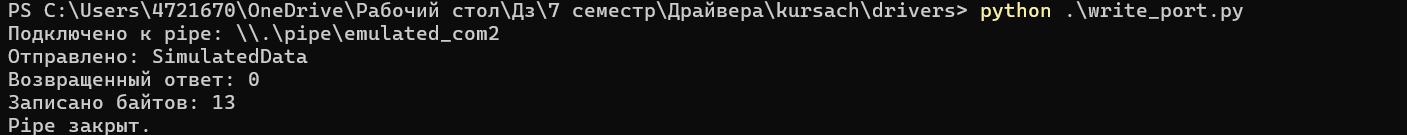


Рисунок 7. Запуск пользовательского кода

После этого запускаем нашу скомпилированную программу для получения данных:

Рисунок 8. Запуск скомпилированной программы

В WinDbg увидим следующее:

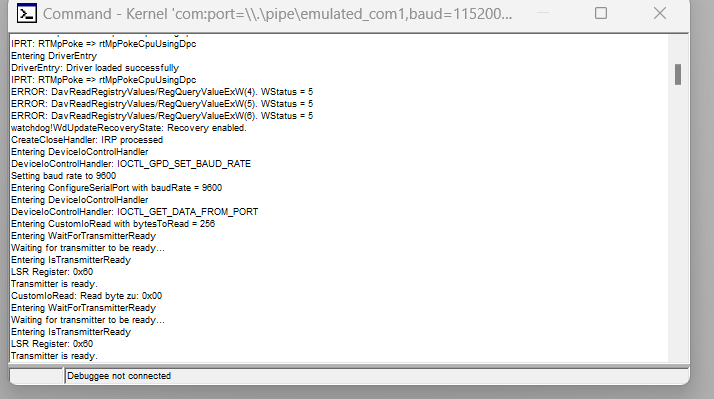
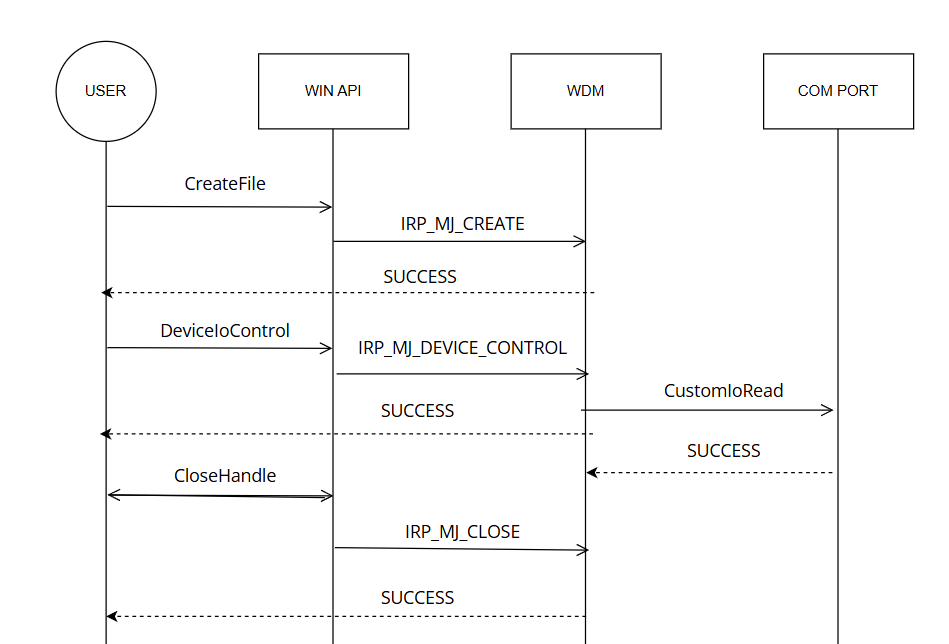


Рисунок 9. Вывод отладочных сообщений в WinDbg

## **Диаграмма взаимодействия**



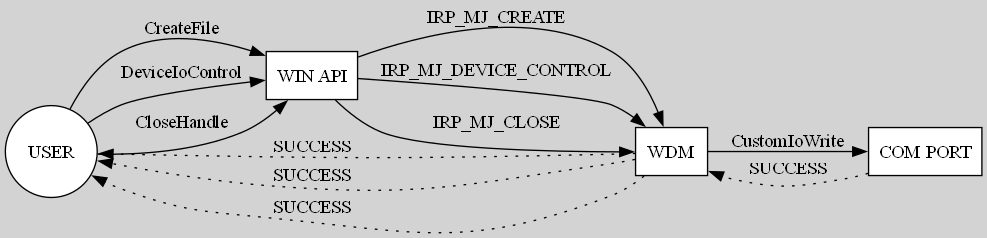


Рисунок 10. Диаграмма взаимодействия