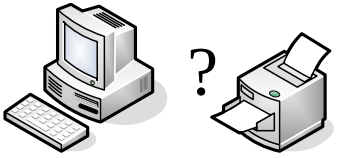
ГОУВПО
«Воронежский государственный технический университет»

Кафедра автоматизированных и вычислительных систем

32-2012

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
«Написание драйверов»
по выполнению лабораторных работ № 3 по дисциплине "Периферийные устройства" для студентов спец. 230101 всей очной формы обучения



Воронеж 2012

Составители: канд. техн. наук А.М. Нужный,
канд. техн. наук Н.И. Гребенникова

УДК 681.3.06

Методические указания по выполнению лабораторных работ № 3 по дисциплине "Периферийные устройства" для студентов специальности 230100 «Вычислительные машины, комплексы, системы и сети» очной и очной сокращенной форм обучения / ГОУВПО «Воронежский государственный технический университет»; сост. А.М. Нужный, Н.И.Гребенникова. Воронеж, 2012. 68 с.

В методических указаниях приводятся задания и теоретические сведения по темам лабораторных работ.

Предназначены для студентов специальности 230100 по дисциплине "Периферийные устройства"

Методические указания подготовлены в электронном виде в текстовом редакторе MS WORD и содержатся в файле PU1.DOC.

Ил. 2. Библиогр.: 4 назв.

Рецензент д-р техн. наук, проф. О.Н.Чопоров

Ответственный за выпуск зав. кафедрой д-р техн. наук, проф. С.Л. Подвальный

Издается по решению редакционно-издательского совета Воронежского государственного технического университета

ВВЕДЕНИЕ

Методические указания посвящены рассмотрению практических приемов по разработке драйверов периферийных устройств для операционной системы Windows.

Ниже описаны основные приемы работы с такими средствами разработки, как Microsoft Windows DDK. Device Driver Kit, — пакет разработки драйверов, включающий компилятор, редактор связей (линкер), заголовочные файлы, библиотеки, большой набор примеров (часть из которых является драйверами, реально работающими в операционной системе) и, разумеется, документацию. В состав пакета входит также отладчик WinDbg, позволяющий проводить интерактивную отладку драйвера на двухкомпьютерной конфигурации и при наличии файлов отладочных идентификаторов операционной системы WinDbg кроме того, позволяет просматривать файлы дампа (образа) памяти, полученного при фатальных сбоях операционной системы (так называемый crash dump file).

Разработка драйверов в рамках лабораторных работ осуществляется для виртуального устройства виртуальной машины Oracle VirtualBox.

ЛАБОРАТОРНАЯ РАБОТА № 3

НАПИСАНИЕ ДРАЙВЕРА ДЛЯ РАБОТЫ С ПРЕРЫВАНИЯМИ.

ПЕРЕДАЧА ДАННЫХ С ИСПОЛЬЗОВАНИЕМ DMA

Цель работы: изучение драйвера, предназначенного для работы с прерываниями и организации передачи данных с использованием прямого доступа устройства к памяти (Direct memory access, DMA).

Необходимое ПО: Для выполнения работы необходимо установить Oracle VirtualBox (инструкции по установке и настройке приведены в отдельном документе) с установленной на неё виртуальной машиной (ВМ) под управлением ОС Windows XP. Следующее ПО должно присутствовать на ВМ: Microsoft Visual Studio C++, DDK и DebugView. Для чтения/записи данных в буфер виртуального устройства применяется программа PuTTY.

Исходные данные для лабораторной работы:

Все необходимые файлы для компиляции и сборки драйвера находятся в папке PIO:

– itit.cpp; - файл содержит код драйвера на языке C++;

– driver.h – заголовочный файл, содержащий объявления, необходимые для компиляции драйвера;

– MAKEFILE - управляет работой программы Build пакета DDK;

– SOURCES - отражает индивидуальные настройки процесса компиляции и сборки;

Тестирующая программы находится в папке ExampleTest.

Порядок выполнения работы:

1. Ознакомиться с механизмами передачи данных посредством прямого доступа к памяти и регистрами устройства.
2. Ознакомиться с описанием виртуального устройства.
3. Выполнить компиляцию и сборку драйвера средствами DDK.
4. Провести тестирование драйвера.

Механизмы передачи данных

При всем многообразии компьютерной техники, существует три основных механизма, используя которые устройство может обмениваться с центральным процессором или, в широком смысле, с компьютером:

– Программируемый ввод/вывод.

– Прямой доступ к памяти (Direct Memory Access, DMA).

– Совместно используемые области памяти.

При выборе разработчиком аппаратуры механизма передачи данных, используемого для связи с устройством, следует исходить из скорости, с которой требуется передавать данные, и средним размером передаваемого непрерывного блока данных.

Прямой доступ к памяти

Прямой доступ к памяти (Direct Memory Access, DMA) использует выгоды от вовлечения в работу вторичного процессора, называемого контроллером DMA (DMA controller, DMAC, контроллер ПДП). Этот контроллер является вспомогательным процессором с ограниченным набором возможностей и обязанностей, но при этом с достаточным интеллектом и статусом, чтобы выполнять передачу данных между устройством и оперативной памятью. Контроллер DMA работает параллельно с основным процессором, и обычно его деятельность почти незаметна в общем функционировании системы.

При инициации операции ввода/вывода с привлечением DMA метода драйвер должен выполнить установку нужного состояния DMA контроллера, определив адрес начала буферной области и количество

передаваемых данных. По поступлению от драйвера указания начать работу DMA контроллер приступает к выполнению передачи данных между устройством и системной оперативной памятью. Когда DMA контроллер завершает передачу данных полностью, генерируется прерывание. Таким образом, драйверный код выполняется только в начале операции передачи данных и затем — лишь по ее завершении, освобождая центральный процессор для выполнения других задач.

Следует обратить особое внимание на то, что в Windows NT при DMA операциях всегда используются логические адреса (трюк с памятью, схожий с виртуальной адресацией, но предназначенный специально для DMA операций), позволяющие "обмануть" DMA контроллер, для которого логические адреса кажутся непрерывными, хотя и могут представлять физически разрывные области памяти.

Высокоскоростные устройства, которые вынуждены регулярно заниматься передачей крупных блоков данных, являются первыми кандидатами на использования метода DMA. По сравнению с операциями программируемого ввода/вывода, интенсивность использования сигналов прерываний и активность драйвера существенно уменьшаются. Жесткие диски, устройства мультимедиа, сетевые карты являются примерами устройств, использующих DMA.

DMA операции с использованием системных контроллеров

Первоначальная спецификация персонального компьютера по IBM (и последовавшие стандарты) включали основную плату (называемую ныне "материнской") с набором общих DMA котроллеров.

Каждый DMA контроллер предоставляет DMA каналы (DMA channel), и данное устройство может быть сконфигурировано так, чтобы использовать один (или более) доступных каналов. Изначально существовало четыре канала, которые были расширены до семи при введении спецификации AT. Системный DMA (system DMA) известен еще как «slave DMA».

Преимущество использования системного DMA состоит в том, что количество аппаратной логики для реализации DMA в устройстве уменьшается. К минусам следует отнести то, что в случае совместного использования канала данным устройством, оно получает возможность участия в DMA передаче данных только один раз за определённый временной интервал. В каждый конкретный момент времени DMA канал находится «в собственности» только одного устройства, остальные попытки использования этого канала со стороны других устройств откладываются до момента, когда первое устройство «откажется» от собственности на канал. Такое совместное использование канала дает плохие результаты при использовании его для двух высокоскоростных устройств. Контроллер флоппи дисководов в большинстве персональных компьютеров как раз является устройством, реализующим операции «slave DMA».

Операции bus master DMA

Более сложные устройства, которые не желают быть зависимыми от системных DMA контроллеров, содержат собственные аппаратные средства обеспечения DMA. Так как эти средства принадлежат собственно устройству, то передача происходит по «воле» устройства — если, разумеется, протокол шины поддерживает такие действия. Для выполнения своей операции устройству необходимо

получить «контроль над шиной» в результате процедуры, которая описывается в протоколах соответствующих шин.

Примечание. Виртуальное устройство, драйверы для которого рассматриваются в данной серии лабораторных работ, использует именно bus master DMA.

Память, отведенная устройству

Третий из механизмов передачи данных состоит в том, что для доступа к устройству используются диапазоны адресов памяти, открытые для совместного использования.

В некоторых случаях эти диапазоны определены жестко, как в случае с диапазоном адресов буфера адаптера VGA.

Память, отведенная устройству, является ресурсом, предоставляемым драйверу устройства операционной системой, и обычно с ним можно ознакомиться в апплете свойств драйвера в настройках системы (Пуск - Настройка - Панель Управления - Система - Устройства - Диспетчер Устройств - устройство - ресурсы).

Регистры устройств

Драйверы взаимодействуют с подключаемыми устройствами путем чтения из регистров или записи в их внутренние регистры. Каждый внутренний регистр устройства обычно реализует одну из функций, перечисленных ниже:

– Регистр состояния. Обычно *считывается* драйвером, когда тому необходимо получить информацию о текущем состоянии устройства.

– Регистр команд. Биты этого регистра управляют устройством некоторым образом, например, начиная или

прекращая передачу данных. Драйвер обычно производит запись в такие регистры.

– Регистры данных. Обычно такие регистры используются для передачи данных между устройством и драйвером. В выходные (output) регистры, регистры вывода, драйвер производит запись, в то время как информация входных (input) регистров, регистров ввода, считывается драйвером.

Доступ к регистрам устройства достигается в результате выполнения инструкций доступа к портам ввода/вывода (port address) или обращения к определенным адресам в адресном пространстве оперативной памяти (memory-mapped address), что и интерпретируются системой как доступ к аппаратным регистрам.

Доступ к регистрам устройств. Пространство ввода/вывода

В некоторых реализациях процессорных архитектур доступ к регистрам устройств осуществляется при помощи специальных команд процессора – инструкций ввода/вывода. Они ссылаются на специальные наборы выводов процессора и определяют отдельное шинно-адресное пространство для устройств ввода/вывода. Адреса на этих шинах широко известны как порты (ports) и не имеют никакого отношения к адресации памяти. В архитектуре Intel x86 адресное пространство ввода/вывода имеет размер 64 КБ (16 разрядов), а в языке ассемблера определено две инструкции для чтения и записи в этом пространстве: 'IN' и 'OUT' (точнее, две группы инструкций, внутри которых различие имеет место по разрядности считываемых/записываемых данных).

Поскольку при создании драйвера следует избегать привязки к аппаратной платформе, Microsoft рекомендует избегать и использования реальных инструкций IN/OUT. Вместо этого следует использовать макроопределения HAL. Соответствие между традиционными инструкциям DOS/Windows ассемблера и макроопределениями HAL приводится в таблице 1.

Таблица 1 – Макроопределения HAL для доступа к портам ввода/вывода

Макроопределение HAL	Описание
READ_PORT_UCHAR	Чтение 1 байта из порта ввода/вывода
READ_PORT_USHORT	Чтение 16-ти разрядного слова из порта ввода/вывода
READ_PORT_ULONG	Чтение 32-х разрядного слова из порта ввода/вывода
READ_PORT_BUFFER_UCHAR	Чтение массива байт из порта ввода/вывода
READ_PORT_BUFFER_USHORT	Чтение массива 16-ти разрядных слов из порта ввода/вывода
READ_PORT_BUFFER_ULONG	Чтение массива 32-х разрядных слов из порта ввода/вывода
WRITE_PORT_UCHAR	Запись 1 байта в порт ввода/вывода
WRITE_PORT_USHORT	Запись 16-ти разрядного слова в порт ввода/вывода

WRITE_PORT_ULONG	Запись 32-х разрядного слова в порт ввода/вывода
WRITE_PORT_BUFFER_UCHAR	Запись массива байт в порт ввода/вывода
WRITE_PORT_BUFFER_USHORT	Запись массива 16-ти разрядных слов в порт ввода/вывода
WRITE_PORT_BUFFER_ULONG	Запись массива 32-х разрядных слов в порт ввода/вывода

Доступ через адресацию в памяти

Далеко не все создатели процессоров находили целесообразным организацию «портового» доступа к регистрам устройств через адресное пространство ввода/вывода. В альтернативном подходе доступ к регистрам устройств осуществлялся путем обращения к определенным адресам в пространстве памяти (memory address space). В пример можно привести архитектуру PDP-11 Unibus, где вовсе не было портов ввода/вывода и инструкций процессора для работы с ними: все регистры всех устройств имели свое место в общем пространстве адресации памяти и реагировали на обычную инструкцию доступа к памяти. В некоторых случаях (например, для видеоадаптеров в архитектуре Intel x86) допускаются оба способа доступа сразу: и через пространство ввода/вывода, и через адресное пространство.

Таблица 2 – Макроопределения HAL для доступа к регистрам устройств через адресацию в памяти.

Макроопределение HAL	Описание

READ_REGISTER_XXX	Чтение одного значения из регистра ввода/вывода
WRITE_REGISTER_XXX	Запись одного значения в регистр ввода/вывода
READ_REGISTER_BUFFER_XXX	Чтение массива значений из последовательного набора регистров ввода/вывода
WRITE_REGISTER_BUFFER_XXX	Запись массива значений в набор из следующих друг за другом регистров ввода/вывода

Как и в предыдущем случае, определены макросы HAL для доступа к таким memory mapped (то есть с доступом посредством адресации в памяти) регистрам, что описывается в таблице 2 (XXX принимает значения UCHAR, USHORT или ULONG). Так как эти макроопределения по содержанию отличаются от HAL макроопределений для операций с портами ввода/вывода, драйверный код должен быть разработан так, чтобы компиляция для разных платформ (с разными методами доступа к регистрам) проходила корректно. Хорошим приемом является составление таких макроопределений условной компиляции, которые указывали бы на один из нужных HAL макросов в зависимости от ключей компиляции.

Структура драйвера

Алгоритм передачи данных с помощью DMA-контроллера

Схематичное изображение возможного сценария передачи данных с помощью DMA-контроллера приведено на рисунке 1.

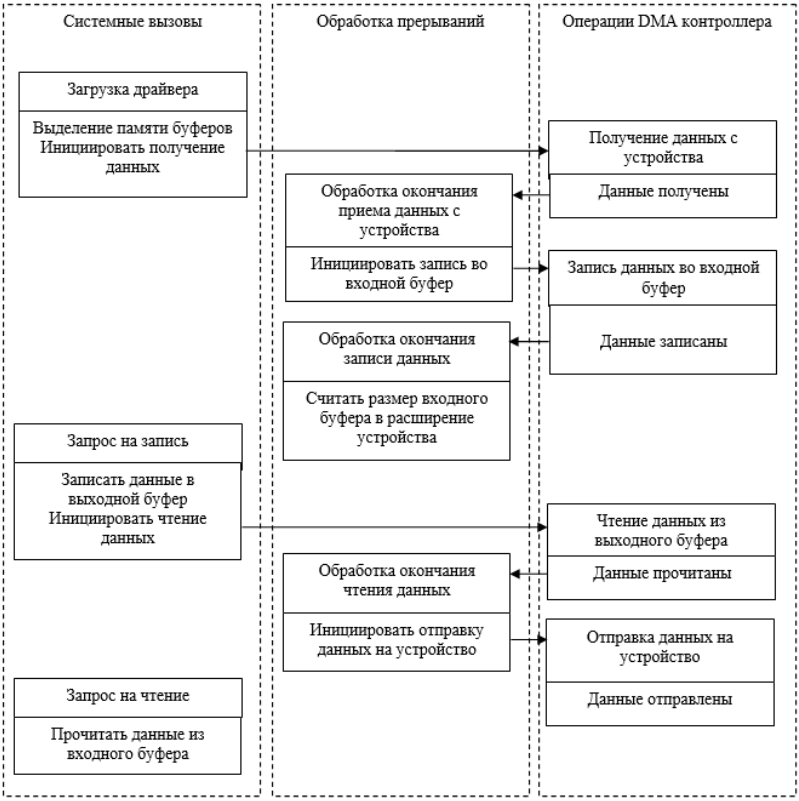


Рисунок 1 – Сценарий передачи данных.

Необходимо отметить, что приведенный сценарий передачи данных не является единственным: порядок

выполняемых цепочек операций зависит от порядка инициации системных вызовов.

Заголовочный файл Driver.h

Замечание: инструкции, заключаемые в блок `#if DBG … #endif` являются отладочными и подробно рассматриваться не будут.

Ниже приводится полный текст файла Driver.h, содержащий необходимые объявления.

```
//==================================================
==========================
// Driver.h - заголовочный файл для драйвера
//==================================================
==========================
#pragma once

extern "C" {
#include <NTDDK.h>
}

#define MAX_BUFFER_SIZE (4096)

/* Регистры PCI устройства */
#define STATUS_REG      0x0
#define COMAND_REG      0x0
#define START_ADR_REG   0x1
#define SIZE_REG        0x2
#define DATA_REG        0x3
/* Комманды PCI */
#define WRONG_CMD       0x00000000
#define WRITE_CMD       0x00000001 //записать буфер
приёма в память
#define READ_CMD        0x00000002 //считать в
буфер отправки из памяти
#define SEND_CMD        0x00000003 //отправить
содержимое буфера отправки
```

```
#define START_READ_CMD  0x00000004 //начать приём
пакетов => вкл. прерывания
/* Статус устройства PCI */
#define DMA_READ_DONE   0x00000001
#define DMA_WRITE_DONE  0x00000002
#define DATA_RECVD      0x00000003
#define BUFF_IS_EMPTY   0x00000004

typedef struct _DEVICE_EXTENSION
{
      PDEVICE_OBJECT    pDevice;
      UNICODE_STRING    ustrDeviceName;   //
внутреннее имя устройства
      UNICODE_STRING    ustrSymLinkName;  //
внешнее имя (символьная ссылка)
      PVOID             deviceOutBuffer,
      //указатель на буфер отправки
                        deviceInBuffer;
      //указатель на буфер приема
      ULONG             OutBufferLength,  // объем
буфера отправки
                        InBufferLength;   // объем
буфера приема
      PUCHAR            portBase;         // адрес
порта ввода/вывода
      ULONG             Irq;              // Irq в
терминах шины PCIBus
      PKINTERRUPT       pIntObj;          //
interrupt object
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Заголовочный файл драйвера содержит включение библиотеки ntddk.h как внешней на языке C, объявления необходимых для работы драйвера констант и объявление структуры расширения устройства DEVICE_EXTENSION и указателя на нее – PDEVICE_EXTENSION.

Первая константа MAX_BUFFER_SIZE определяет размер областей памяти, которые будут совместно использоваться драйвером и DMA-контроллером.

Следующий блок констант содержит адреса регистров PCI-устройства. Все константы имеют тип PUCHAR – указатель на unsigned char (имеет размер 1 байт). В данных константах (а также в нижеследующих) хранятся не реальные адреса в виртуальном адресном пространстве, а смещения относительно базового адреса пространства устройства. Рассматриваемый драйвер использует следующие регистры: регистр команд с адресом COMAND_REG, регистр состояния с адресом STATUS_REG, регистр начального адреса области памяти, с которой будет работать устройство с адресом START_ADR_REG, регистр фактического размера переданных или полученных данных с адресом SIZE_REG.

Дальнейший блок констант содержит коды команд устройства. Используются следующие команды: команда записи устройством данных из буфера приема в заданную область памяти WRITE_CMD, команда чтения устройством данных из заданной области памяти в буфер отправки READ_CMD, команда отправки данных, находящихся в буфере отправки SEND_CMD, команда старта приема контроллером пакетов от устройства и генерации контроллером прерываний START_READ_CMD.

Третий блок констант содержит коды состояний (статусов) устройства. Рассматриваемый драйвер использует следующиие состояния устройства: состояние «чтение контроллером данных из памяти в буфер отправки завершено» DMA_READ_DONE, состояние «запись контроллером данных из буфера приема в память завершена» DMA_WRITE_DONE, состояние «пакет, полученный от устройства записан контроллером в буфер приема» DATA_RECVD.

Структуру расширения объекта устройства разработчик драйвера определяет самостоятельно. В этой

структуре сохранен указатель на структуру объекта устройства, имена устройства (внутреннее и внешняя символьная ссылка, поля ustrDeviceName, ustrSymLinkName в формате Unicode-строки), указатель на область памяти, предназначенную для вывода deviceOutBuffer, указатель на область памяти, предназначенную для ввода deviceInBuffer, переменные длин переданных и полученных данных (OutBufferLength, InBufferLength) базовый адрес устройства portBase, уровень прерываний, генерируемых контроллером Irq и указатель на обработчик прерываний, генерируемых контроллером pIntObj.

Уровень прерываний Irq имеет тип ULONG (беззнаковое длинное целое) и также является номером вектора прерываний.

Хранение указателя на объект устройства в структуре расширения является общепринятой традицией «правописания» драйверов, поскольку достаточно часто указатель на расширение передается в качестве контекстных указателей разным процедурам, которые, в конечном счете, нуждаются и в получении ссылки на сам объект устройства.

Предварительные объявления, процедура DriverEntry и вспомогательная функция CreateDevice

Программная часть драйвера начинается с обязательной функции с именем DriverEntry(), которая автоматически вызывается системой на этапе загрузки драйвера. Эта функция должна содержать все действия по его инициализации. Далее необходимо определить используемые в ней данные, в т.ч. указатель на DEVICE_OBJECT и две символьные строки UNICODE_STRING с именами устройств. Системные

программы взаимодействуют с объектом устройства, созданным драйвером, посредством указателя на него. Необходимо иметь в виду, что объект устройства должен иметь два имени, одно - в пространстве имен NT, другое - в пространстве имен Win32. Эти имена должны представлять собой структуры UNICODE_STRING. Имена объектов устройств составляются по определенным правилам. NT-имя предваряется префиксом Device, а Win32-имя – префиксом DosDevice.

```
//=================================================
========================
// Файл init.cpp
//=================================================
========================
#include "driver.h"
// Предварительные объявления функций
static NTSTATUS CreateDevice(IN PDRIVER_OBJECT
      pDriverObject,
                              IN ULONG
      portBase,
                              IN ULONG
      Irq );
static NTSTATUS Create        (IN PDEVICE_OBJECT
      pDevObj,
                               IN PIRP
      pIrp);
static NTSTATUS Close         (IN PDEVICE_OBJECT
      pDevObj,
                               IN PIRP
      pIrp);
static VOID     DriverUnload(IN PDRIVER_OBJECT
      pDriverObject);
static NTSTATUS Write         (IN PDEVICE_OBJECT
      pDevObj,
                               IN PIRP
      pIrp);
static NTSTATUS Read          (IN PDEVICE_OBJECT
      pDevObj,
```

```
                                    IN PIRP
      pIrp);
BOOLEAN         Isr             (IN PKINTERRUPT
      pInterruptObject,
                                    IN PVOID
      pServiceContext);
//=================================================
=======================
// Функция:     DriverEntry
// Назначение:  Инициализирует драйвер, подключает
объект устройства для
//              получения прерываний.
// Аргументы:   pDriverObject - поступает от
Диспетчера ввода/вывода
//              pRegistryPath - указатель на
Юникод-строку,
//              обозначающую раздел Системного
Реестра, созданный
//              для данного драйвера.
// Возвращаемое значение:
//              NTSTATUS - в случае нормального
завершения STATUS_SUCCESS
//                         или код ошибки
STATUS_Xxx
extern "C" NTSTATUS DriverEntry (IN PDRIVER_OBJECT
pDriverObject,
                                    IN PUNICODE_STRING
pRegistryPath)
{
      NTSTATUS status;
      #if DBG
      DbgPrint("IRQDMA: in DriverEntry,
RegistryPath is:\n     %ws. \n",
                  pRegistryPath->Buffer);
      #endif
      // Регистрируем рабочие процедуры драйвера:
      pDriverObject->DriverUnload = DriverUnload;
      pDriverObject->MajorFunction[IRP_MJ_CREATE] =
Create;
      pDriverObject->MajorFunction[IRP_MJ_CLOSE]  =
Close;
```

```
	pDriverObject->MajorFunction[IRP_MJ_WRITE]  =
Write;
	pDriverObject->MajorFunction[IRP_MJ_READ]   =
Read;
	// Работа по созданию объекта устройства,
подключению
	// ресурсов, прерывания, созданию символьной
ссылки:
	status = CreateDevice(pDriverObject, 0xD040,
23);
return status;
}

//=================================================
=======================
// Функция:     CreateDevice
// Назначение:  Создание устройства с точки зрения
системы
// Аргументы: pDriverObject - поступает от
Диспетчера ввода/вывода
//            portBase - адрес базового регистра
параллельного порта (378h)
//            Irq - прерывание (в терминах шины
ISA) для обслуживания порта
// Возвращаемое значение:
//            NTSTATUS - в случае нормального
завершения STATUS_SUCCESS
//                       или код ошибки
STATUS_Xxx
NTSTATUS CreateDevice (IN PDRIVER_OBJECT
pDriverObject,
                       IN ULONG
portBase,
                       IN ULONG            Irq)
{
	NTSTATUS                status;
	PDEVICE_OBJECT          pDevObj;
	PDEVICE_EXTENSION pDevExt;
	// Создаем внутреннее имя устройства
	UNICODE_STRING          devName;
	RtlInitUnicodeString(&devName,
L"\\Device\\IRQDMA");
```

```
	// Создаем объект устройства
	status= IoCreateDevice(pDriverObject,

sizeof(DEVICE_EXTENSION),
				&devName,
				FILE_DEVICE_UNKNOWN,
				0,
				TRUE,
				&pDevObj);
	if (!NT_SUCCESS(status))  return status;
	// Будем использовать метод буферизации
BUFFERED_IO
	pDevObj->Flags |= DO_BUFFERED_IO;
	// Заполняем данными структуру Device
Extension
	pDevExt = (PDEVICE_EXTENSION)pDevObj-
>DeviceExtension;
	pDevExt->pDevice        = pDevObj;   //
сохраняем - это пригодится
	pDevExt->ustrDeviceName = devName;
	pDevExt->Irq            = Irq;
	pDevExt->portBase       = (PUCHAR)portBase;
	//выполняем выделение физически непрерывной
области памятипод буферы приема/отправки
	PVOID VirtualAddress;
	PHYSICAL_ADDRESS maxAcceptableAddress;
	maxAcceptableAddress.QuadPart =
0x000FFFFFULL;
	pDevExt->deviceOutBuffer =
MmAllocateContiguousMemory(MAX_BUFFER_SIZE,
maxAcceptableAddress);
	if (pDevExt->deviceOutBuffer == NULL)
	{     // При неудаче - удаляем объект
устройства:
		IoDeleteDevice(pDevObj);
		#if DBG
		DbgPrint("Can't get memory");
		#endif
		return status;
	}
```

```
      pDevExt->deviceInBuffer =
MmAllocateContiguousMemory(MAX_BUFFER_SIZE,
maxAcceptableAddress);
      if (pDevExt->deviceInBuffer == NULL)
      {     // При неудаче - удаляем объект
устройства:
            IoDeleteDevice(pDevObj);
            #if DBG
            DbgPrint("Can't get memory");
            #endif
            return status;
      }
      pDevExt->OutBufferLength = 0; // сейчас нет
данных в буфере отправки
      pDevExt->InBufferLength  = 0; // сейчас нет
данных в буфере приема
      // Создаем и подключаем объект прерываний:
      KIRQL     kIrql;
      KAFFINITY kAffinity;
      ULONG kVector =
            HalGetInterruptVector(PCIBus, 0,
pDevExt->Irq, pDevExt->Irq,
                                    &kIrql,
&kAffinity);
      // Замечание. Для PCI шины второй параметр
(номер шины) обычно
      // равен 0, а третий и четвертый параметры
равны.
      #if DBG
      DbgPrint("IRQDMA: Interrupt %d converted to
kIrql = %d, "
            "kAffinity = %d, kVector = %X(hex)\n",
            pDevExt->Irq, kIrql, kAffinity,
kVector);
      #endif
      status =
IoConnectInterrupt (&pDevExt->pIntObj, // Здесь
будет создан Interrupt Object
                  Isr,       // Наша функция ISR
                  pDevExt,   // Этот указатель
ISR функция будет
```

```
                    // получать при вызове
(контекстный указатель)
                    NULL,       // Не будем
использовать spin-блокировку для
                    // безопасного доступа к
совместно используемым данным
                    kVector,    // транслированное
значение прерывания
                    kIrql,      // DIRQL
                    kIrql,      // DIRQL
                    Latched,    // Прерывание по
перепаду
                    TRUE,       // Совместно
используемое (Shared) прерывание
                    kAffinity, // Поцессоров в
мультипроцессорной системе
                    FALSE);     // Не сохранять
значения регистров сопроцессора
      if (!NT_SUCCESS(status))
      {
            // В случае неудачи удаляем объект
устройства
            IoDeleteDevice(pDevObj);
            return status;
      }
      #if DBG
      DbgPrint("IRQDMA: Interrupt successfully
connected.\n");
      #endif
      // Создаем символьную ссылку:
      UNICODE_STRING symLinkName;
      // Сформировать символьное имя:
      // Для того, чтобы работало в Windows 98 & XP
:
      #define SYM_LINK_NAME L"\\DosDevices\\IRQDMA"
      RtlInitUnicodeString(&symLinkName,
SYM_LINK_NAME);
      // Создать символьную ссылку:
      status = IoCreateSymbolicLink(&symLinkName,
&devName);
      if (!NT_SUCCESS(status))
```

```
        {       // При неудаче - отключаемся от
прерывания и
                // удаляем объект устройства:
                IoDisconnectInterrupt(pDevExt-
>pIntObj);
                IoDeleteDevice(pDevObj);
                return status;
        }
        pDevExt->ustrSymLinkName = symLinkName;
        #if DBG
        DbgPrint("IRQDMA: Symbolic Link is created:
%ws. \n",
                                        pDevExt-
>ustrSymLinkName.Buffer);
        #endif
        WRITE_PORT_ULONG((PULONG)(pDevExt->portBase +
COMAND_REG), START_READ_CMD);
        return STATUS_SUCCESS;
}
```

Рассмотрим приведенный код. Первый блок:

```
        static NTSTATUS CreateDevice(IN
        PDRIVER_OBJECT pDriverObject, IN ULONG
        portBase, IN ULONG Irq);
        static NTSTATUS Create(IN PDEVICE_OBJECT
              pDevObj, IN PIRP pIrp);
        static NTSTATUS Close(IN PDEVICE_OBJECT
        pDevObj, IN PIRP pIrp);
        static VOID DriverUnload(IN PDRIVER_OBJECT
        pDriverObject);
        static NTSTATUS Write(IN PDEVICE_OBJECT
        pDevObj, IN PIRP pIrp);
        static NTSTATUS Read(IN PDEVICE_OBJECT
        pDevObj, IN PIRP pIrp);
        BOOLEAN Isr(IN PKINTERRUPT pInterruptObject,
        IN PVOID     pServiceContext);
```

содержит объявления функций, которые будут использоваться до определения этих функций. Каждая из этих функций будет описана отдельно.

Следующий блок:

```
extern "C" NTSTATUS DriverEntry (IN
PDRIVER_OBJECT  pDriverObject,
                                    IN
PUNICODE_STRING pRegistryPath)
{
      NTSTATUS status;
      pDriverObject->DriverUnload =
DriverUnload;
      pDriverObject-
>MajorFunction[IRP_MJ_CREATE] = Create;
      pDriverObject-
>MajorFunction[IRP_MJ_CLOSE]  = Close;
      pDriverObject-
>MajorFunction[IRP_MJ_WRITE]  = Write;
      pDriverObject-
>MajorFunction[IRP_MJ_READ]   = Read;
      status = CreateDevice(pDriverObject,
0xD040, 23);
      return status;
}
```

является определением функции DriverEntiry(). В качестве первого параметра наша функция получает указатель DriverObject типа PDRIVER_OBJECT на объект драйвера. При загрузке драйвера система создает объект драйвера (driver object) – структуру типа DRIVER_OBJECT, олицетворяющую образ драйвера в памяти и содержащую необходимые для функционирования драйвера данные и адреса функций. Второй параметр RegistryPath типа PUNICODE_STRING – указатель на Unicode-строку UNICODE_STRING с разделом реестра (driver service key). Возвращаемым значением является переменная типа NTSTATUS, которая должна содержать статус

осуществленной операции. В поле DriverUnload объекта драйвера адрес функции DriverUnload, которая ответственна за выгрузку драйвера. Затем в массив MajorFunction, являющийся полем объекта драйвера записываются адреса функций создания, закрытия, чтения и записи. Системные константы IRP_MJ_CREATE, IRP_MJ_CLOSE, IRP_MJ_READ, IRP_MJ_WRITE – индексы элементов массива MajorFunction, ответственных за хранение адресов функций создания, закрытия, чтения и записи соответственно. Завершающими инструкциями являются вызов определенной разработчиком драйвера вспомогательной функции CreateDevice(), которой в качестве агрументов передается указатель на объект драйвера, числовая константа – базовый адрес устройства и числовая константа – уровень прерывания, генерируемого контроллером, и возврат из функции статуса осуществленной операции.

Блок:

```
      NTSTATUS CreateDevice (IN PDRIVER_OBJECT
pDriverObject,
                              IN ULONG
portBase,
                             IN ULONG
      Irq)
      {…}
```

является определением вспомогательной функции CreateDevice, производящей операции, необходимые для регистрации объекта устройства. В качестве агрументов функция принимает указатель на объект драйвера, базовый адрес устройства и уровень прерываний, генерируемых контроллером.

Первый блок функции:

```
	NTSTATUS status;
	PDEVICE_OBJECT pDevObj;
	PDEVICE_EXTENSION pDevExt;
	UNICODE_STRING devName;
```

объявляет необходимые переменные: переменную статуса осуществленной операции, указатель на объект устройства, указатель на структуру расширения устройства (вид которой определен в заголовочном файле driver.h) и Unicode-строку с именем устройства.

Рассмотрим следующий блок:

```
	RtlInitUnicodeString(&devName,
L"\\Device\\IRQDMA");
	status = IoCreateDevice(pDriverObject,

sizeof(DEVICE_EXTENSION),
                            &devName,
                            FILE_DEVICE_UNKNOWN,
                            0,
                            TRUE,
                            &pDevObj);
	if(!NT_SUCCESS(status)) return status;
	pDevObj->Flags |= DO_BUFFERED_IO;
```

Первая иснтрукция преобразует строку "\\Device\\IRQDMA" (префикс L обеспечивает хранение строки не в виде массива char, а в виде массива wchar_t) в Unicode-кодировку и записывает результат преобразования в переменную devName.

Вторая инструкция создает Functional Device Object – структуру объекта устройства. Параметры системной функции IoCreateDevice: указатель на объект драйвера; размер структуры расширения устройства, определенной нами в заголовочном файле (размер необходим для выделения памяти под структуру расширения); адрес переменной с именем устройства; константа типа

устройства FILE_DEVICE_UNKNOWN (неизвестный тип); флаги характеристик устройства (отсутствуют, значение равно нулю); флаг эксклюзивного доступа к устройству со значением TRUE; и адрес структуры объекта устройства.

Третья инструкция прерывает выполнение функции в случае, если статус, который вернула функция IoCreateDevice не соответствует успешному завершению операции (для проверки используется макрос NT_SUCCESS()), возвращая ошибочный статус.

Четвертая инструкция выставляет структуре объекта устройства флаг, сигнализирующий о буферизованном вводе-выводе, и о том, что драйвер должен получать адрес буферной области из поля IRP пакета AssociatedIrp.SystemBuffer.

Далее следует блок заполнения структуры расширения устройства:

```
      pDevExt = (PDEVICE_EXTENSION)pDevObj-
>DeviceExtension;
      pDevExt->pDevice        = pDevObj;
      pDevExt->ustrDeviceName = devName;
      pDevExt->Irq            = Irq;
      pDevExt->portBase       = (PUCHAR)portBase;
```

Первая инструкция извлекает из созданного объекта устройства указатель на структуру расширения устройства и приводит к определенному нам типу указателя PEXAMPLE_DEVICE_EXTENSION. Последующие инструкции заполняют поля расширения устройства нужными значениями: ссылкой на объект устройства, внутренним именем устройства, уровнем прерывания, полученным в качестве аргумента, базовым адресом устройства (преобразуемым из беззнакового длинного целого ULONG в указатель на беззнаковый символ PUCHAR).

Следующий блок выделяет требуемые области памяти:

```
	PHYSICAL_ADDRESS maxAcceptableAddress;
	maxAcceptableAddress.QuadPart =
0x000FFFFFULL;
	pDevExt->deviceOutBuffer =
MmAllocateContiguousMemory(MAX_BUFFER_SIZE,
maxAcceptableAddress);
	if (pDevExt->deviceOutBuffer == NULL)
		IoDeleteDevice(pDevObj);
		return status;
	}
	pDevExt->deviceInBuffer =
MmAllocateContiguousMemory(MAX_BUFFER_SIZE,
maxAcceptableAddress);
	if (pDevExt->deviceInBuffer == NULL)
	{
		IoDeleteDevice(pDevObj);
		return status;
	}
	pDevExt->OutBufferLength = 0;
	pDevExt->InBufferLength  = 0;
```

Первыми двумя инструкциями идет объявление структуры maxAcceptableAddress, хранящей максимально возможный физический адрес в памяти, используемый драйвером и запись в нее численного значения 0x000FFFFF (суффикс ULL говорит о том, что константа имеет тип unsigned long long – особо длинное беззнаковое целое).

Третьей инструкцией осуществляется вызов системной функции MmAllocateContigiousMemory, которая выделяет физически непрерывную область памяти размером MAX_BUFFER_SIZE так, что ее адреса не выходят за заданный предел maxAcceptableAddress). Виртуальный адрес выделенной области записывается в структуру расширения устройства в поле deviceOutBuffer –

данная область будет использоваться драйвером и контроллером DMA как буфер обмена данными от системы к устройству.

Блок if проверяет, была ли действительно выделена память, и в случае, если выделение закончилось неудачей, удаляет объект устройства и возвращает ошибочный статус.

Следующей инструкцией производится выделение физически непрерывной области памяти для буфера обмена данными от устройства к системе. Виртуальный адрес области записывается в структуру расширения устройства в поле deviceInBuffer.

Последующий блок if аналогичен предыдущему – проверка выделения памяти и возврат ошибочного статуса.

Последние две инструкции обнуляют поля с размерами буферов обмена в структуре расширения устройства (так как ни чтение, ни запись пока не производились).

Далее следует блок создания и подключения объекта прерывания:

```
      KIRQL     kIrql;
      KAFFINITY kAffinity;
      ULONG kVector =
            HalGetInterruptVector(PCIBus, 0,
pDevExt->Irq, pDevExt->Irq,
                                  &kIrql,
&kAffinity);
      status = IoConnectInterrupt (&pDevExt-
>pIntObj,
                  Isr,
                  pDevExt,
                  NULL,
                  kVector,
                  kIrql,
                  kIrql,
                  Latched,
```

```
                    TRUE,
                    kAffinity,
                    FALSE);
    if (!NT_SUCCESS(status))
    {
          IoDeleteDevice(pDevObj);
          return status;
    }
```

В начале объявляются две переменные: kIrql – для хранения транслированного системой уровня прерывания, kAffinity – для хранения специального значения, отражающего количество и параметры ядер процессора, используемого системой.

Следующей инструкцией производится трансляция вектора прерываний для шины PCI с помощью вызова HalGetInterruptVector. Параметры вызова: тип интерфейса (системная константа PCIBus), номер шины (в случае PCI – всегда 0), уровень прерывания (задан аргументом функции CreateDevice), номер вектора прерываний для уровня (значение равно самому уровню), адрес переменной для хранения траслированного уровня прерывания, адрес переменной kAffinity. После вызова данной функции в объявленную переменную kVector записывается транслированный номер вектора прерываний, в kIrql – транслированный уровень прерывания, в kAffinity – количество и параметры ядер процессора, используемого системой.

Следующая инструкция включает обработку прерываний по заданному транслированному уровню и транслированному вектору вызовом IoConnectInterrupt. Первый параметр – адрес, по которому будет записан объект прерывания (в нашем случае – в созданное для этого поле расширения устройства pIntObj). Второй параметр – указатель на функцию обработки прерывания

Isr, которая будет вызываться всякий раз, когда прерывание будет сгенерировано контроллером (функция будет описана ниже). Третий параметр – указатель, который всегда будет передаваться функции обработки прерывания при ее вызове (в нашем случае – структура расширения устройства). Четвертый агрумент – так называемый spin-lock, необходимый для защиты совместно используемых различными программными модулями данных. В нашем случае его значение NULL и рассматриваться он не будет. Пятый аргумент – транслированный номер вектора прерываний. Шестой и седьмой – транслированный уровень прерывания (аргументы одинаковы, т.к. прерывание обрабатывается одной функцией-обработчиком). Седьмой аргумент – задает вид сигнала, на который должен реагировать обработчик прерывания, в нашем случае – системная константа Latched – перепад сигнала. Восьмой аргумент – флаг, задающий, является ли прерывание совместно используемым, в нашем случае – TRUE. Девятый аргумент – число процессоров в системе, kAffinity. Десятый аргумент определяет, будут ли сохраняться значения арифметического сопроцессора при вызове обработчика прерывания (в нашем случае FALSE). Возвращает функция статус операции подключения прерывания.

Блок if проверяет, успешно ли подключилось прерывание, и в противном случае удаляет объект устройства и возвращает ошибочный статус.

Следующий блок:

```
	UNICODE_STRING symLinkName;
	#define SYM_LINK_NAME L"\\DosDevices\\
IRQDMA"
	RtlInitUnicodeString(&symLinkName,
SYM_LINK_NAME);
```

```
	status = IoCreateSymbolicLink(&symLinkName,
&devName);
	if(!NT_SUCCESS(status))
	{
		IoDisconnectInterrupt( pDevExt->pIntObj
	);
		IoDeleteDevice(pDevObj);
		return status;
	}
	pDevExt->ustrSymLinkName = symLinkName;
```

содержит операции по созданию символьной ссылки. Первая строка объявляет переменную Unicode-строки для записи символьной ссылки, вторая строка – константу wchar_t-строки со ссылкой. Третья инструкция преобразует wchar_t-строку в Unicode-строку, четвертая – записывает адрес unicode-строки со ссылкой в поле структуры расширения устройства. Пятая инструкция вызывает системную функцию IoCreateSymbolicLink(), которая устанавливает соответствие между именем устройства и символьной ссылкой, видимой пользователю. Блок if проверяет статус операции создания ссылки и в случае неудачи отключает прерывание, удаляет объект устройства и возвращает ошибочный статус.

Заключительный блок:

```
	WRITE_PORT_ULONG((PULONG)(pDevExt->portBase +
COMAND_REG), START_READ_CMD);
	return STATUS_SUCCESS;
```

записывает в порт команд DMA-контроллера код команд START_READ_CMD. После этого контроллер начинает прием пакетов от устройства и генерацию прерываний. Финальной инструкцией функция возвращает успешный статус.

Рабочие процедуры обработки запросов read/write

Процедуры Read/Write предназначены для обработки запросов Диспетчера ввода/вывода, которые он формирует в виде IRP пакетов с кодами IRP_MJ_READ/IRP_MJ_WRITE по результатам обращения к драйверу из пользовательских приложений с вызовами read/write или из кода режима ядра с вызовами ZwReadFile или ZwWriteFile.

```
//================================================
========================
// Функция:     Write
// Назначение:  Обрабатывает запрос по поводу Win32
вызова  WriteFile
// Аргументы:   pDevObj - поступает от Диспетчера
ввода/вывода
//              pIrp - поступает от Диспетчера
ввода/вывода
// Возвращаемое значение:
//              NTSTATUS - в случае нормального
завершения STATUS_SUCCESS
//                         или код ошибки
STATUS_Xxx
NTSTATUS Write (IN PDEVICE_OBJECT pDevObj, IN PIRP
pIrp)
{
      PDEVICE_EXTENSION  pDevExt =
      (PDEVICE_EXTENSION) pDevObj->DeviceExtension;
      PIO_STACK_LOCATION pIrpStack =
IoGetCurrentIrpStackLocation(pIrp);
      // Размер буфера для данных, отправленных
пользователем
      ULONG InputLength = pIrpStack-
>Parameters.Write.Length;
      // Задаем печать отладочных сообщений
      #if DBG
      DbgPrint("IRQDMA: in Write now\n");
```

```
	#endif
	ULONG BytesTxd = 0;// Число
переданных/полученных байт (пока 0)
	NTSTATUS status = STATUS_SUCCESS;
//Завершение с кодом status
	#if DBG
	DbgPrint("InputLength: %d", InputLength);
	#endif
	UCHAR *buff = (PUCHAR)pIrp-
>AssociatedIrp.SystemBuffer;
	pDevExt->OutBufferLength = InputLength;
	while(InputLength > 0)
	{
		RtlFillMemory((PVOID)((ULONG)pDevExt-
>deviceOutBuffer + BytesTxd), 1, *buff);
		InputLength--;
		BytesTxd++;
		buff++;
	}
	WRITE_PORT_ULONG((PULONG)(pDevExt->portBase +
START_ADR_REG),
(ULONG)MmGetPhysicalAddress(pDevExt-
>deviceOutBuffer).QuadPart);
	WRITE_PORT_ULONG((PULONG)(pDevExt->portBase +
SIZE_REG), pDevExt->OutBufferLength);
	WRITE_PORT_ULONG((PULONG)(pDevExt->portBase +
COMAND_REG), READ_CMD);
	pDevExt->OutBufferLength = 0;
	pIrp->IoStatus.Status = status;
	pIrp->IoStatus.Information = BytesTxd;
	IoCompleteRequest(pIrp, IO_NO_INCREMENT);
	return STATUS_SUCCESS;
}

//=================================================
=======================
// Функция:     Read
// Назначение:  Обрабатывает запрос по поводу Win32
вызова ReadFile
// Аргументы:   pDevObj - поступает от Диспетчера
ввода/вывода
```

```
//              pIrp - поступает от Диспетчера
ввода/вывода
// Возвращаемое значение:
//              NTSTATUS - в случае нормального
завершения STATUS_SUCCESS
//                         или код ошибки
STATUS_Xxx
NTSTATUS Read (IN PDEVICE_OBJECT pDevObj, IN PIRP
pIrp)
{
      PDEVICE_EXTENSION pDevExt =
      (PDEVICE_EXTENSION) pDevObj->DeviceExtension;
      PIO_STACK_LOCATION pIrpStack =
IoGetCurrentIrpStackLocation(pIrp);
      UCHAR *buff = (PUCHAR)pIrp-
>AssociatedIrp.SystemBuffer;
      // Задаем печать отладочных сообщений
      #if DBG
      DbgPrint("IRQDMA: in Read now\n");
      #endif
      ULONG BytesTxd = 0;// Число
переданных/полученных байт (пока 0)
      NTSTATUS status = STATUS_SUCCESS;
//Завершение с кодом status
      if(READ_PORT_ULONG((PULONG)(pDevExt->portBase
+ STATUS_REG)) == (ULONG)DMA_WRITE_DONE)
      {
            while(pDevExt->InBufferLength > 0)
            {
                  *(buff + BytesTxd) =
*(PUCHAR)((ULONG)pDevExt->deviceInBuffer +
BytesTxd);
                  DbgPrint("data: %c", *(buff +
BytesTxd));
                  pDevExt->InBufferLength--;
                  BytesTxd++;
            }
      }
      pIrp->IoStatus.Status = status;
      pIrp->IoStatus.Information = BytesTxd;
      IoCompleteRequest(pIrp, IO_NO_INCREMENT);
return STATUS_SUCCESS;
```

```
}
```

Рассмотрим код функций подробнее. Функция:

```
      NTSTATUS Write(IN PDEVICE_OBJECT pDevObj, IN
PIRP pIrp)
```

отвечает за операцию записи. Адрес данной функции при загрузке драйвера (DriverEntry) был записан в объект драйвера в массив MajorFunction по индексу IRP_MJ_WRITE, и именно поэтому функция будет вызываться для обработки пакетов IRP с кодом IRP_MJ_WRITE, создаваемых диспетчером при вызове клиентским кодом операции write. На вход функции поступает указатель на объект устройства и указатель на IRP пакет.

Первый блок функции:

```
      PDEVICE_EXTENSION  pDevExt =
      (PDEVICE_EXTENSION) pDevObj->DeviceExtension;
      PIO_STACK_LOCATION pIrpStack =
IoGetCurrentIrpStackLocation(pIrp);
      ULONG InputLength = pIrpStack-
>Parameters.Write.Length;
      ULONG BytesTxd = 0;
      NTSTATUS status = STATUS_SUCCESS;
      UCHAR *buff = (PUCHAR)pIrp-
>AssociatedIrp.SystemBuffer;
      pDevExt->OutBufferLength = InputLength;
```

проводит подготовительные операции.

Первая инструкция извлекает адрес структуры расширения устройства из структуры объекта устройства.

Вторая инструкция вызовом функции IoGetCurrentIrpStackLocation получает адрес вершины IRP-стека по переданному IRP-пакету.

Третья инструкция извлекает длину переданных на запись данных. Поле Parameters верхнего элемента стека с адресом pIrpStack содержит в себе поле Parameters (параметры запроса), которое, в свою очередь содержит поле Write с параметрами, относящимися к процедуре записи. Из поля Write извлекается длина блока данных переданных на запись. Параметры запроса заполняются диспетчером ввода-вывода без участия программиста.

Четвертая и пятая инструкции объявляют переменную для хранения числа переданных байт (начальное значение – ноль) и переменную статуса операции (начальное значение – STATUS_SUCCESS, успешная операция.

Шестая инструкция объявляет переменную buff, в которую записывается адрес начала системного буфера, извлекаемый поля структуры IRP AccociatedIrp.SystemBuffer.

Седьмая инструкция записывает в структуру расширения устройства размер выходного буфера – длина блока переданных на запись данных.

Цикл:

```
	while(InputLength > 0)
	{
		RtlFillMemory((PVOID)((ULONG)pDevExt-
>deviceOutBuffer + BytesTxd), 1, *buff);
		InputLength--;
		BytesTxd++;
		buff++;
	}
```

производит побайтную запись данных из системного буфера в область памяти, выделенную передачи данных от системы к устройству.

Первая инструкция цикла – вызов системной функции RtlFillMemory, которая копирует данные из одной области памяти в другую. Первый агрумент – конструкция:

```
(PVOID)((ULONG)pDevExt->deviceOutBuffer + BytesTxd)
```

К указателю на выходной буфер (выделенный в DriverEntry) deviceOutBuffer, хранящемуся в расширении устройства прибавляется текущий счетчик байт, результат сложения приводится к нетипизированному адресу (тип PVOID). Таким образом, первый аргумент – адрес в выходном буфере, в который необходимо скопировать текущий байт.

Второй аргумент – число копируемых байт, равное единице.

Третий аргумент – значение, хранящееся по адресу buff – текущий элемент системного буфера.

Три последующих инструкции смещают указатель буфера на одну позицию (переходя к следующему байту массива пересылаемых данных), уменьшают число еще не переданных байт на единицу и увеличивают число переданных байт на единицу.

Цикл выполняется до тех пор, пока число непереданных байт не станет равным нулю. Таким образом, данная конструкция обеспечивает проход по системному буферу с побайтным копированием в выходную область памяти хранящихся в системном буфере данных в заданных переменной InputLength пределах.

Далее следует блок задания команды DMA-контроллеру:

```
WRITE_PORT_ULONG((PULONG)(pDevExt->portBase + START_ADR_REG),
```

```
(ULONG)MmGetPhysicalAddress(pDevExt-
>deviceOutBuffer).QuadPart);
      WRITE_PORT_ULONG((PULONG)(pDevExt->portBase +
SIZE_REG), pDevExt->OutBufferLength);
      WRITE_PORT_ULONG((PULONG)(pDevExt->portBase +
COMAND_REG), READ_CMD);
```

Первой инструкцией в порт стартового адреса контроллера (адрес порта получается в результате сложения базового адреса устройства, извлеченного из структуры расширения и смещения START_ADR_REG) пишется значение физического адреса выходного буфера (получается вызовом системной функции MmGetPhysicalAddress с аргументом, равным виртуальному адресу выходной области памяти, извлеченному из расширения устройства). Таким образом DMA-контроллер «узнает», откуда начинать чтение данных для пересылки устройству.

Второй инструкцией в порт фактического размера данных контроллера (складывается из базового адреса устройства и смещения SIZE_REG) записывается размер выходного буфера (извлечен из расширения устройства). Так DMA-контроллер получает информацию о том, сколько байт необходимо прочесть для пересылки устройству.

Третья инструкция в порт команд контроллера (базовый адрес + смещение COMAND_REG) пишет код команды чтения READ_CMD, чем инициирует чтение контроллером в буфер отправки числа байт, записанного в регистр размера данных из области памяти с начальным адресом, записанным в регистр стартового адреса (то есть всех пришедших на обработку с IRP пакетом данных).

Важно понимать, что как правило процедура чтение занимает некоторое время, и отправка данных устройству из буфера отправки произведется только по ее

завершению. Отсылка инициируется в обработчике прерывания (рассматривается ниже).

Завершающий блок:

```
pDevExt->OutBufferLength = 0;
pIrp->IoStatus.Status = status;
pIrp->IoStatus.Information = BytesTxd;
IoCompleteRequest(pIrp, IO_NO_INCREMENT);
return STATUS_SUCCESS;
```

обнуляет фактический размер выходного буфера (так как данные переданы контроллеру), информирует диспетчер ввода-вывода об успешном завершении операции записи, записывая в IRP-структуру успешный статус, число переданных байт и вызывая системную функцию IoCompleteRequest для пакета IRP с параметром IO_NO_INCREMENT – константой, сигнализирующей о том, что у вызвавшего операцию ввода-вывода потока не поменяется приоритет в связи с ожиданием ввода-вывода. Завершающая инструкция возвращает успешный статус операции.

Функция:

```
NTSTATUS Read(IN PDEVICE_OBJECT pDevObj, IN PIRP pIrp)
```

отвечает за чтение и во многом походит на функцию Write. Адрес данной функции при загрузке драйвера был записан в объект драйвера в массив MajorFunction по индексу IRP_MJ_READ, и именно поэтому функция будет вызываться для обработки пакетов IRP с кодом IRP_MJ_READ, создаваемых диспетчером при вызове клиентским кодом операции read. На вход функции (как и на вход функции Write) поступает указатель на объект

устройства и указатель на IRP пакет, а возвращает она статус совершенной операции.

Начальный блок функции:

```
PDEVICE_EXTENSION  pDevExt =
(PDEVICE_EXTENSION)pDevObj->DeviceExtension;
PIO_STACK_LOCATION pIrpStack =
IoGetCurrentIrpStackLocation(pIrp);
UCHAR *buff =(PUCHAR)pIrp-
>AssociatedIrp.SystemBuffer;
ULONG BytesTxd = 0;
NTSTATUS status = STATUS_SUCCESS;
```

практически совпадает с соответствующим блоком Write (извлечение расширения устройства, получение стека, получение адреса системного буфера, задание счетчика полученных байт, объявление переменной успешного статуса).

Далее следует блок if:

```
if(READ_PORT_ULONG((PULONG)(pDevExt->portBase
+ STATUS_REG)) == (ULONG)DMA_WRITE_DONE)
{…}
```

Проверка имеет следующий смысл: чтение данных может произвестись только тогда, когда DMA-контроллер закончил запись содержимого буфера приема в область памяти, выделенную для обмена данными от устройства к системе. Драйвер считывает код статуса контроллера из порта с базовым адресом устройства и смещением STATUS_REG и сравнивает его с кодом состояния DMA_WRITE_DONE. При совпадении кодов запускается цикл побайтного копирования данных из входной области памяти в системный буфер:

```
while(pDevExt->InBufferLength > 0)
```

```
        {
                *(buff + BytesTxd) =
*(PUCHAR)((ULONG)pDevExt->deviceInBuffer +
BytesTxd);
                pDevExt->InBufferLength--;
                BytesTxd++;
        }
```

Поле InBufferLength содержит значение, полученное при обработке прерывания от контроллера (описывается ниже), равное числу записанных контроллером байт. Первая инструкция цикла записывает в системный буфер по индексу, равному текущему счетчику байт значение, содержащееся во входной области памяти по тому же индексу (начальный адрес области памяти записан в поле расширения устройства deviceInBuffer при выполнении DriverEntry). Две последующие инструкции уменьшают на единицу счетчик еще не прочитанных байт и увеличивают счетчик уже прочитанных байт.

Таким образом, данная конструкция обеспечивает проход по входной области памяти с побайтным копированием в системный буфер хранящихся во входной области данных в заданных полем InBufferLength пределах.

Завершающий блок:

```
pIrp->IoStatus.Status = status;
pIrp->IoStatus.Information = BytesTxd;
IoCompleteRequest(pIrp, IO_NO_INCREMENT);
return STATUS_SUCCESS;
```

информирует диспетчер ввода-вывода об успешном завершении операции записи, записывая в IRP-структуру успешный статус, число переданных байт и вызывая системную функцию IoCompleteRequest для пакета IRP с

параметром IO_NO_INCREMENT – константой, сигнализирующей о том, что у вызвавшего операцию ввода-вывода потока не поменяется приоритет в связи с ожиданием ввода-вывода. Завершающая инструкция возвращает успешный статус операции.

Функция обработки прерываний контроллера

Функция Isr предназначена для обработки прерываний, генерируемых DMA-контроллером.

Прерывания возникают по следующим событиям контроллера:

- данные от устройства получены и записаны в буфер приема;
- завершено чтение данных из выходной области памяти в буфер отправки;
- завершена запись данных из буфера приема во входную область памяти.

```
//=================================================
====================
// Процедура обслуживания прерывания:
//
BOOLEAN Isr (IN PKINTERRUPT pInterruptObject, IN
PVOID pServiceContext )
{
      PDEVICE_EXTENSION pDevExt =
(PDEVICE_EXTENSION) pServiceContext;
      PDEVICE_OBJECT    pDevObj = pDevExt->pDevice;

      KIRQL             currentIrql =
KeGetCurrentIrql();

      #if DBG
      DbgPrint("PciIrq: In Isr procedure,
ISR_Irql=%d\n", currentIrql);
      #endif
```

```
      if(READ_PORT_ULONG((PULONG)(pDevExt->portBase
+ STATUS_REG)) == (ULONG)DATA_RECVD)
      {
            WRITE_PORT_ULONG((PULONG)(pDevExt-
>portBase + START_ADR_REG),
(ULONG)MmGetPhysicalAddress(pDevExt-
>deviceInBuffer).QuadPart);
            WRITE_PORT_ULONG((PULONG)(pDevExt-
>portBase + COMAND_REG), WRITE_CMD);
      }

      if(READ_PORT_ULONG((PULONG)(pDevExt->portBase
+ STATUS_REG)) == (ULONG)DMA_READ_DONE)
      {
            WRITE_PORT_ULONG((PULONG)(pDevExt-
>portBase + COMAND_REG), SEND_CMD);
      }

      if(READ_PORT_ULONG((PULONG)(pDevExt->portBase
+ STATUS_REG)) == (ULONG)DMA_WRITE_DONE)
      {
            pDevExt->InBufferLength =
READ_PORT_ULONG((PULONG)(pDevExt->portBase +
SIZE_REG));
            DbgPrint("InBufferLength: %d\n",
pDevExt->InBufferLength);
      }

      return TRUE;     // нормальное завершение
обработки прерывания
}
```

Функция принимает на вход объект прерывания и так называемый сервисный контекст (процедура DriverEntry делает таким контекстом указатель на структуру расширения устройства).

Начальный блок функции:

```
      PDEVICE_EXTENSION pDevExt =
(PDEVICE_EXTENSION) pServiceContext;
      PDEVICE_OBJECT    pDevObj = pDevExt->pDevice;
```

приводит указатель на контекст к типу расширения устройства и извлекает указатель на объект устройства из расширения.

Блок:

```
      if(READ_PORT_ULONG((PULONG)(pDevExt->portBase
      + STATUS_REG)) == (ULONG)DATA_RECVD)
      {
            WRITE_PORT_ULONG((PULONG)(pDevExt-
            >portBase + START_ADR_REG),
            (ULONG)MmGetPhysicalAddress(pDevExt-
            >deviceInBuffer).QuadPart);
            WRITE_PORT_ULONG((PULONG)(pDevExt-
>portBase + COMAND_REG),
            WRITE_CMD);
      }
```

обрабатывает прерывание, сгенерированное в момент, когда данные от устройства получены контроллером и записаны в буфер приема – в этом случае с порта статуса контроллера считывается код DATA_RECVD.

Внутри блока if инициируется запись данных из буфера приема во входную разделяемую область памяти: в порт стартового адреса записывается физический начальный адрес входной области (получается из виртуального адреса, хранимого в расширении устройства, вызовом функции MmGetPhysicalAddress), а в порт команд контроллера – код команды записи WRITE_CMD.

Блок:

```
      if(READ_PORT_ULONG((PULONG)(pDevExt->portBase
      + STATUS_REG)) == (ULONG)DMA_READ_DONE)
      {
```

```
        WRITE_PORT_ULONG((PULONG)(pDevExt-
        >portBase + COMAND_REG), SEND_CMD);
}
```

обрабатывает прерывание, сгенерированное в момент, когда завершено чтение контроллером данных из разделяемой выходной области памяти в буфер отправки – в этом случае с порта статуса контроллера считывается код DMA_READ_DONE.

Внутри блока if инициируется отсылка данных из буфера отправки на устройство: в порт команд контроллера записывается код команды отправки SEND_CMD.

Блок:

```
if(READ_PORT_ULONG((PULONG)(pDevExt->portBase
+ STATUS_REG)) == (ULONG)DMA_WRITE_DONE)
{
      pDevExt->InBufferLength =
      READ_PORT_ULONG((PULONG)(pDevExt-
      >portBase + SIZE_REG));
}
```

обрабатывает прерывание, сгенерированное в момент, когда завершена запись контроллером данных из буфера приема в разделяемую входную область памяти – в этом случае с порта статуса контроллера считывается код DMA_WRITE_DONE.

Внутри блока if в поле расширеня устройства, хранящее фактическую длину входного буфера записывается значение, считанное с порта фактической длины данных контроллера SIZE_REG. Это значение будет использовано при вызове системой операции чтения.

Рабочая процедура обработки запросов открытия драйвера

Процедура Create предназначена для обработки запросов Диспетчера ввода/вывода, которые он формирует в виде IRP пакетов с кодами IRP_MJ_CREATE по результатам обращения к драйверу из пользовательских приложений с вызовами CreateFile или из кода режима ядра с вызовами ZwCreateFile. В нашем примере эта функция не выполняет никаких особых действий (хотя можно было бы завести счетчик открытых дескрипторов и т.п.), однако без регистрации данной процедуры система просто не позволила бы клиенту «открыть» драйвер для работы с ним (хотя сам драйвер мог бы успешно загружаться и стартовать).

```
//=================================================
========================
// Функция:     Create
// Назначение:  Обрабатывает запрос по поводу Win32
вызова CreateFile
// Аргументы: pDevObj - поступает от Диспетчера
ввода/вывода
//            pIrp - поступает от Диспетчера
ввода/вывода
// Возвращаемое значение: STATUS_SUCCESS
NTSTATUS Create (IN PDEVICE_OBJECT pDevObj, IN PIRP
pIrp)
{
      #if DBG
      DbgPrint("IRQDMA: in Create now\n");
      #endif
      pIrp->IoStatus.Status = STATUS_SUCCESS;
      pIrp->IoStatus.Information = 0; // ни одного
байта не передано
      IoCompleteRequest(pIrp, IO_NO_INCREMENT);
      return STATUS_SUCCESS;
}
```

Рабочая процедура обработки запросов закрытия драйвера

Процедура Close предназначена для обработки запросов Диспетчера ввода/вывода, которые он формирует в виде IRP пакетов с кодом IRP_MJ_CLOSE по результатам обращения к драйверу из пользовательских приложений с вызовами CloseHandle или из кода режима ядра с вызовами ZwClose. В нашем примере эта функция не выполняет никаких особых действий, однако, выполнив регистрацию процедуры открытия файла, мы теперь просто обязаны зарегистрировать процедуру завершения работы клиента с открытым дескриптором. Заметим, что если клиент пользовательского режима забывает закрыть полученный при открытии доступа к драйверу дескриптор, то за него эти запросы выполняет операционная система (впрочем, как и в отношении всех открытых приложениями файлов, когда приложения завершаются без явного закрытия открытых файлов).

```
//=================================================
=======================
// Функция:       Close
// Назначение:    Обрабатывает запрос по поводу
Win32 вызова CloseHandle
// Аргументы:     pDevObj - поступает от Диспетчера
ввода/вывода
//                pIrp - поступает от Диспетчера
ввода/вывода
// Возвращаемое значение: STATUS_SUCCESS
//
NTSTATUS Close (IN PDEVICE_OBJECT pDevObj, IN PIRP
pIrp)
{
      #if DBG
      DbgPrint("IRQDMA: in Close now\n");
      #endif
      pIrp->IoStatus.Status = STATUS_SUCCESS;
```

```
	pIrp->IoStatus.Information = 0; // ни одного
байта не передано
	IoCompleteRequest(pIrp, IO_NO_INCREMENT);
	return STATUS_SUCCESS;
}
```

Рабочая процедура выгрузки драйвера

Процедура DriverUnload выполняет завершающую работу.

```
//================================================
========================
// Функция:     DriverUnload
// Назначение:  Останавливает и удаляет объекты
устройств, отключает
//              прерывания, подготавливает драйвер
к выгрузке.
// Аргументы:   pDriverObject - поступает от
Диспетчера ввода/вывода
// Возвращаемое значение: нет
VOID DriverUnload (IN PDRIVER_OBJECT pDriverObject)
{
	#if DBG
	DbgPrint("IRQDMA: in DriverUnload now\n");
	#endif
	PDEVICE_OBJECT    pNextObj = pDriverObject-
>DeviceObject;
	// Проход по всем устройствам, контролирумым
драйвером
	for(; pNextObj!=NULL;)
	{
		PDEVICE_EXTENSION pDevExt =
			(PDEVICE_EXTENSION)pNextObj-
>DeviceExtension;
		// Удаляем объект прерываний:
		if (pDevExt->pIntObj)
		{
			IoDisconnectInterrupt(pDevExt-
>pIntObj);
```

```
            }
            // Удаляем символьную ссылку:
            IoDeleteSymbolicLink(&pDevExt-
>ustrSymLinkName);
            #if DBG
            DbgPrint("IRQDMA: SymLink %ws
deleted\n",
                        pDevExt-
>ustrSymLinkName.Buffer);
            #endif
            // Сохраняем ссылку на следующее
устройство и удаляем
            // текущий объект устройства:
            pNextObj = pNextObj->NextDevice;
            IoDeleteDevice(pDevExt->pDevice);
      }
}
```

Функция DriverUnload принимает в качестве параметра на вход указатель на структуру объекта драйвера. Начальная инструкция:

```
      PDEVICE_OBJECT    pNextObj = pDriverObject-
>DeviceObject;
```

объявляет указатель на объект устройства, необходимый для обхода всех объектов устройств, ассоциированных с драйвером (в нашем случае такое устройство всего одно). Текущим объектом устройством назначается устройство, извлеченное из объекта драйвера.

Цикл:

```
      for(; pNextObj!=NULL;)
      {
            PDEVICE_EXTENSION pDevExt =
                  (PDEVICE_EXTENSION)pNextObj-
      >DeviceExtension;
            if (pDevExt->pIntObj)
            {
```

```
                  IoDisconnectInterrupt(pDevExt-
>pIntObj);
            }

            IoDeleteSymbolicLink(&pDevExt-
      >ustrSymLinkName);
            pNextObj = pNextObj->NextDevice;
            IoDeleteDevice(pDevExt->pDevice);
      }
```

предназначен для обхода всех ассоциированных с драйвером устройств. Тело цикла выполняется до тех пор, пока указатель на следующее устройство не станет равным NULL (т.е. устройств не останется). Первой инструкцией из текущего устройства извлекается структура расширения устройства. Блок if проверяет, имеется ли в расширении устройства объект прерывания, и отключает прерывание вызовом IoDisconnectInterrupt на его объекте. Следующая инструкция извлекает из текущего объекта устройства ссылку на следующее (в нашем примере при первом же выполнении данной инструкции вернется NULL). Далее вызывается системная функция IoDeleteSymbolicLink, удаляющая символьную ссылку, извлеченную из расширения устройства.Последняя инструкция удаляет из системы объект устройства вызовом функции IoDeleteDevice. Ссылка на структуру объекта устройства извлекается из структуры расширения.

Компиляция и сборка драйвера IRQDMA.sys

Для компиляции и сборки драйвера утилитой Build пакета DDK потребуется создать два файла описания проекта — Makefile и Sources.

Файл Makefile. Этот файл управляет работой программы Build и в нашем случае имеет стандартный вид (его можно найти практически в любой директории примеров DDK), а именно:

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Файл Sources. Файл sources отражает индивидуальные настройки процесса компиляции и сборки. В нашем случае файл Sources чрезвычайно прост и имеет вид:

```
TARGETNAME=IRQDMA
TARGETTYPE=DRIVER
TARGETPATH=obj
SOURCES=init.cpp
```

Данный файл задает имя выходного файла Example, параметр `TARGETNAME`. Поскольку проект (TARGETTYPE) имеет тип DRIVER, то выходной файл будет иметь расширение .sys. Промежуточные файлы будут размещены во вложенной директории .\obj. Строка SOURCES задает единственный файл с исходным текстом — это файл init.cpp.

Для компиляции «чистовой» версии драйвера нужно запустить .exe файл:

Пуск – Все программы – Development Kits – Windows DDK 3790.1830 – Build Environments – Windows XP – Windows XP Checked Build Environment.exe

Для компиляции отладочной версии (данная версия позволяет получать отладочные сообщения от драйвера в программе DebugView) драйвера нужно запустить:

Пуск – Все программы – Development Kits – Windows DDK 3790.1830 – Build Environments – Windows XP – Windows XP Free Build Environment.exe

Когда программы запущена, нужно выполнить консольную команду перехода к директории, в которой находятся файлы с кодом драйвера и файлы описания проекта и вызвать команду build (например):

```
C:\WINDDK\3790.1830>cd C:\Example\
C:\Example>build
```

После выполнения этих действий начнется компиляция и сборка драйвера. В случае ошибок компиляции или сборки вывод будет содержать и их диагностику. Рабочее окно сборки драйвера под Windows XP DDK версии checked показано ниже.

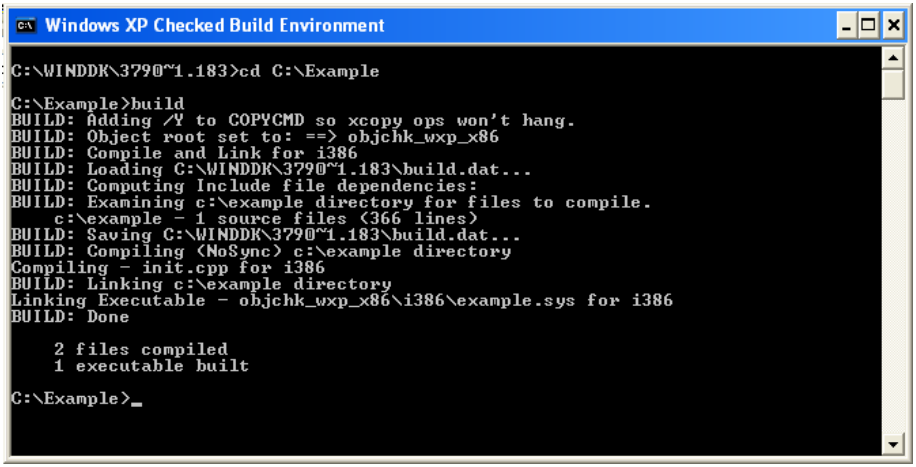


Рисунок 2 – Рабочее окно сборки драйвера под Windows XP DDK версии checked

Тестирование драйвера

Работа с драйвером IRQDMA.sys

Как уже было сказано, из всех возможных способов инсталляции и запуска драйвера IRQDMA.sys, ниже будет использован способ тестирования с применением тестирующего консольного приложения, которое само будет выполнять инсталляцию и удаление драйвера (прибегая к вызовам SCM Менеджера). Для поэтапного ознакомления с процессом взаимодействия драйвера и обращающегося к нему приложения рекомендуется запустить программу IRQDMATest под отладчиком в пошаговом режиме.

Перед запуском тестирующей программы IRQDMATest рекомендуется загрузить программу DebugView, чтобы в ее рабочем окне наблюдать сообщения, поступающие непосредственно из кода драйвера IRQDMA.sys (отладочной сборки).

Протокол полученных программой DebugView отладочных сообщений драйвера можно сохранить в файле для последующего анализа. Ниже приведена информация из такого файла, отражающая события в драйвере IRQDMA.sys с момента его загрузки и вызова процедуры DriverEntry до момента выгрузки и вызова процедуры DriverUnload.

Приложение, работающее с драйвером

Перед тем, как приступить к тестированию драйвера путем вызова его сервисов из приложения, следует это приложение создать, хотя бы в минимальном виде, как это предлагается ниже. И хотя драйвер можно успешно запускать программой Monitor, воспользуемся функциями SCM, поскольку это будет существенно полезнее для будущей практики. Для загрузки и выгрузки драйверов используется диспетчер управления службами

SC Manager (Service Control Manager). Прежде чем начать работау с интерфейсом SC, необходимо получить дескриптор диспетчера служб. Для этого следует обратиться к функции OpenSCManager(). Дескриптор диспетчера служб необходимо использовать при обращении к функциям CreateServise() и OpenService(). Дескрипторы, возвращаемые этими функциями необходимо использовать при обращении к вызовам, имеющим отношение к конкретной службе. К подобным вызовам относятся функции ControlService(), DeleteService() и StartService(). Для освобождения дескрипторов обоих типов используется вызов CloseServiceHandle().

Загрузка и запуск службы подразумевает выполнение следующих действий:

1. Обращение к функции OpenSCManager() для получения дескриптора диспетчера.
2. Обращение к CreateServise() для того, чтобы добавить службу в систему. Если такой сервис уже существует, то CreateServise() выдаст ошибку с кодом 1073 (код ошибки можно прочитать GetLastError()) данная ошибка означает, что сервис уже существует и надо вместо CreateServise() использовать OpenService().
3. Обращение к StartService() для того, чтобы перевести службу в состояние функционирования.
4. Если служба запустилась успешно, то можно вызвать CreateFile(), для получения дескриптора, который мы будем использовать уже непосредственно при обращении к драйверу.
5. По окончании работы необходимо дважды обратиться к CloseServiceHandle() для того, чтобы освободить дескрипторы диспетчера и службы.

Если на каком-то шаге этой последовательности возникла ошибка, нужно выполнить действия обратные тем, которые были выполнены до возникновения ошибки.

Надо помнить о том, что при обращении к функциям подобным CreateServise(), необходимо указывать полное имя исполняемого файла службы (в нашем случае полный путь и имя IRQDMA.sys).

```
//////////////////////////////////////////////////
/////////////////
// (Файл ExampleTest.cpp)
//////////////////////////////////////////////////
/////////////////
// Заголовочные файлы, которые необходимы в данном
приложении:
#include <windows.h>
#include <stdio.h>
#include <winioctl.h>
#include <tchar.h>
#include <stdio.h>
// Внимание! Файл Ioctl.h должен быть получен из
файла Driver.h
// (см. комментрарии к Driver.h) и размещен в одной
директории с
// данным файлом (TestExam.cpp).
#include "Ioctl.h"
// Имя объекта драйвера и местоположение
загружаемого файла
#define DRIVERNAME     _T("IRQDMA")
#define DRIVERBINARY  _T("C:\\IRQDMA.sys")
// Функция установки драйвера на основе SCM вызовов
BOOL InstallDriver( SC_HANDLE  scm, LPCTSTR
DriverName, LPCTSTR driverExec )
{
      SC_HANDLE Service =
      CreateService (scm,    // открытый дескриптор
к SCManager
                     DriverName,      // имя
сервиса - IRQDMA
                     DriverName,               //
для вывода на экран
                     SERVICE_ALL_ACCESS,    //
желаемый доступ
                     SERVICE_KERNEL_DRIVER, // тип
сервиса
                     SERVICE_DEMAND_START,  // тип
запуска
                     SERVICE_ERROR_NORMAL,  // как
обрабатывается ошибка
```

```
                        driverExec,             // путь
к бинарному файлу
                        // Остальные параметры не
используются - укажем NULL
                        NULL,     // Не определяем
группу загрузки
                        NULL, NULL, NULL, NULL);
      if (Service == NULL) // неудача
      {
            DWORD err = GetLastError();
            if (err == ERROR_SERVICE_EXISTS) {/*
уже установлен */}
            // более серьезная ошибка:
            else  printf ("ERR: Can't create
service. Err=%d\n",err);
            // (^^ Ётот код ошибки можно подставить
в ErrLook):
            return FALSE;
      }
      CloseServiceHandle (Service);
return TRUE;
}

// Функция удаления драйвера на основе SCM вызовов
BOOL RemoveDriver(SC_HANDLE scm, LPCTSTR
DriverName)
{
      SC_HANDLE Service = OpenService (scm,
DriverName, SERVICE_ALL_ACCESS);
      if (Service == NULL) return FALSE;
      BOOL ret = DeleteService (Service);
      if (!ret) { /* неудача при удалении драйвера
*/ }

      CloseServiceHandle (Service);
return ret;
}

// Функция запуска драйвера на основе SCM вызовов
BOOL StartDriver(SC_HANDLE  scm, LPCTSTR
DriverName)
{
```

```
      SC_HANDLE Service = OpenService(scm,
DriverName, SERVICE_ALL_ACCESS);
      if (Service == NULL) return FALSE; /* open
failed */
      BOOL ret =
                  StartService( Service, //
дескриптор
                                    0,       //
число аргументов
                                    NULL  ); //
указатель  на аргументы
      if (!ret) // неудача
      {
            DWORD err = GetLastError();
            if (err ==
ERROR_SERVICE_ALREADY_RUNNING)
                  ret = TRUE; // OK, драйвер уже
работает!
            else { /* другие проблемы */}
      }
      CloseServiceHandle (Service);
return ret;
}
// Функция останова драйвера на основе SCM вызовов
BOOL StopDriver(SC_HANDLE  scm, LPCTSTR DriverName)
{
      SC_HANDLE Service =
                  OpenService (scm, DriverName,
SERVICE_ALL_ACCESS );
      if (Service == NULL)  // Невозможно выполнить
останов драйвера
      {
            DWORD err = GetLastError();
            return FALSE;
      }
      SERVICE_STATUS serviceStatus;
      BOOL ret =
      ControlService(Service, SERVICE_CONTROL_STOP,
&serviceStatus);
      if (!ret)
      {
            DWORD err = GetLastError();
```

```
            // дополнительная диагностика
      }
      CloseServiceHandle (Service);
return ret;
}

#define SCM_SERVICE
// ^^^^^^^^^^^^^^^^ вводим элемент условной
компиляции, при помощи
// которого можно отключать использование SCM
установки драйвера
// в тексте данного приложения. (Здесь Ц
использование SCM включено.)
// Основная функция тестирующего приложения.
// Здесь минимум внимания уделен диагностике
ошибочных ситуаций.
// В действительно рабочих приложениях следует
уделить этому больше внимания

int __cdecl main(int argc, char* argv[])
{
      #ifdef SCM_SERVICE
      // Используем сервис SCM для запуска
драйвера.
      BOOL res; // Получаем доступ к SCM :
      SC_HANDLE scm =
OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
      if(scm == NULL) return -1; // неудача
      // Делаем попытку установки драйвера
      res = InstallDriver(scm, DRIVERNAME,
DRIVERBINARY );
      if(!res) // Неудача, но возможно, он уже
инсталлирован
            printf("Cannot install service");
      res = StartDriver  (scm, DRIVERNAME );
      if(!res)
      {
            printf("Cannot start driver!");
            res = RemoveDriver (scm, DRIVERNAME );
            if(!res)
            {
                  printf("Cannot remove driver!");
```

```
            }
            CloseServiceHandle(scm); // Отключаемся
от SCM
            return -1;
      }
      #endif
      HANDLE hHandle =        // Получаем доступ к
драйверу
                  CreateFile( "\\\\.\\IRQDMA ",
                              GENERIC_READ |
GENERIC_WRITE,
                              FILE_SHARE_READ |
FILE_SHARE_WRITE,
                              NULL,
                              OPEN_EXISTING,

      FILE_ATTRIBUTE_NORMAL,
                              NULL );
      if(hHandle==INVALID_HANDLE_VALUE)
      {
            printf("ERR: can not access driver
IRQDMA.sys !\n");
            return (-1);
      }
      DWORD BytesReturned;    // Переменная для
хранения числа переданных байт
      // строка для отправки
      unsigned char *xdata = new unsigned char [5];
      *(xdata) = 'T';
      *(xdata + 1) = 'E';
      *(xdata + 2) = 'S';
      *(xdata + 3) = 'T';
      char gets_line[100];
      gets(gets_line);
      printf( "Write to DATA port\n");
      if( !WriteFile(   hHandle,
                        xdata,
                        4,
                        &BytesReturned,
                        NULL)   )
      {
            printf( "Error with byte receive!" );
```

```
            return(-1);
      }
      // Вывод диагностического сообщения в
консольном окне:
      printf("Byte send: BytesTransferred=%d
xdata=%d\r\n",
            BytesReturned, *xdata);
      printf("Printe some in PuTTY\r\n");
      gets(gets_line);
      printf( "Read from DATA port\n" );
      while(*gets(gets_line) != '1')
      {
            int step = 1;
            // читаем из устройства
            while(step == 1)
            {
                  // читаем данные из устройства
                  if( !ReadFile(    hHandle,
                                    xdata,
                                    1,
                                    &BytesReturned,
                                    NULL)  )
                  {
                        printf( "Error with byte
            receive!" );
                        return(-1);
                  }
                  // Вывод принятой строки в
            консоле:
                  if(BytesReturned>0)
                  {
                        printf("%c", *xdata);
                        step = BytesReturned;
                  }
                  else
                        step = 0;
                  }
      }
      // Закрываем дескриптор доступа к драйверу:
      CloseHandle(hHandle);
      #ifdef SCM_SERVICE
```

```
	// Останавливаем и удаляем драйвер.
Отключаемся от SCM.
	res = StopDriver   (scm, DRIVERNAME );
	if(!res)
	{
		printf("Cannot stop driver!");
		CloseServiceHandle(scm);
		return -1;
	}
	res = RemoveDriver (scm, DRIVERNAME );
	if(!res)
	{
		printf("Cannot remove driver!");
		CloseServiceHandle(scm);
		return -1;
	}
	CloseServiceHandle(scm);
	#endif
	return 0;
}
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Свен Шрайбер «Недокументированные возможности Windows 2000». Издательство «Питер» 2002 год.

2. Солдатов В.П. Программирование драйверов Windows. Изд. 2-е, перераб. и доп. — М.: ООО "Бином-Пресс", 2004 г. — 480 с: ил.

3. П. И. Рудаков, К. Г. Финогенов «Язык ассемблера: уроки программирования» Диалог МИФИ 2001 год.

4. Светлана Сорокина, Андрей Тихонов, Андрей Щербаков «Программирование драйверов и систем безопасности». Издательство «БХВ-Петербург» 2002 год.

СОДЕРЖАНИЕ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
«Написание драйверов»
по выполнению лабораторных работ № 3 по дисциплине
"Периферийные устройства" для студентов спец. 230101
всей очной формы обучения

Составители:
Нужный Александр Михайлович
Гребенникова Наталия Ивановна

В авторской редакции

Подписано к изданию 07.04.12.
Уч.-изд. л. 3,7. "С"

ГОУВПО «Воронежский государственный технический университет»
394026 Воронеж, Московский просп., 14

СПРАВОЧНИК МАГНИТНОГО ДИСКА
(кафедра автоматизированных и вычислительных систем)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
«Написание драйверов»
по выполнению лабораторных работ № 1-2 по дисциплине
"Периферийные устройства" для студентов спец. 230101
всей очной формы обучения

Составители:
А.М.Нужный
Н.И. Гребенникова

PUDRV2часть.doc 516 Кбайт 20.02.2012
3,7 уч.-изд. л.