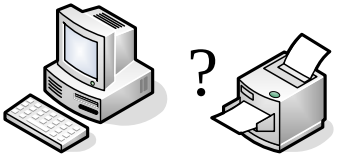
ГОУВПО
«Воронежский государственный технический университет»

Кафедра автоматизированных и вычислительных систем

31-2012

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
«Написание драйверов»
по выполнению лабораторных работ № 1-2 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 № 1-2 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. 84 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 № 1

ИЗУЧЕНИЕ ПРОСТОГО ДРАЙВЕРА «В-СТИЛЕ-NT»

Цель работы: изучение структуры, примеров создания и работы с драйвером в ОС Windows.

Примечание. Приведенный ниже код драйвера Example.sys является завершенным драйвером, который готов к компиляции и использованию в операционной системе в качестве тестового примера. По своей сути, приведенный код не может быть полноценным WDM драйвером (в силу отсутствия в нем некоторых основных рабочих процедур драйверов PnP устройств), хотя и может быть успешно скомпилирован. Драйвер Example.sys более подходит под описание "монолитный драйвер в-стиле-NT", так что он вполне подойдет в качестве заготовки для экспериментов.

Необходимое ПО:

Для выполнения работы необходимо установить Oracle VirtualBox и установить виртуальную машину (ВМ) под управлением ОС Windows XP. Следующее ПО должно присутствовать на ВМ: Microsoft Visual Studio C++, DDK и DebugView, служащую для отображения отладочных сообщений, поступающих от драйвера. Исходный код драйвера можно набирать в любом текстовом редакторе.

Исходные данные для лабораторной работы:

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

– 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 со структурой драйвера и его основными функциями.
2. Выполнить компиляцию и сборку драйвера средствами DDK.
3. Провести тестирование драйвера.

ВЫПОЛНЕНИЕ РАБОТЫ

Структура драйвера

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 объявления, необходимые для компиляции драйвера Example.sys.

```
#ifdef __cplusplus
extern "C"
{
#endif

#include "ntddk.h"

//#include "wdm.h"
//  ^^^^^^^^^^^^^^  если выбрать эту строку и закомментировать
// предыдущую, то компиляция  в среде DDK (при помощи утилиты Build)
// также пройдет успешно, однако драйвер Example не станет от этого
// настоящим WDM драйвером.

#ifdef __cplusplus
}
#endif

typedef struct _EXAMPLE_DEVICE_EXTENSION
{
        PDEVICE_OBJECT fdo;
        UNICODE_STRING ustrSymLinkName;
}                          EXAMPLE_DEVICE_EXTENSION,
*PEXAMPLE_DEVICE_EXTENSION;
```

Заголовочный файл драйвера содержит включение библиотеки ntddk.h (две конструкции вида `#ifdef __cplusplus … #endif` необходимы для того, чтобы компилятор языка C++ включал данную библиотеку как внешнюю на языке C (инструкция заключается в блок `extern "C" {…}`) и объявление структуры расширения устройства EXAMPLE_DEVICE_EXTENSION и указателя на нее – PEXAMPLE_DEVICE_EXTENSION. Данная структура содержит переменную fdo типа PDEVICE_OBJECT (указатель на структуру DEVICE_OBJECT, олицетворяющую собой устройство с которым работает драйвер), которая нужна для удобства работы функции UnloadRoutine, и переменную ustrSymLinkName типа UNICODE_STRING – символьную ссылку в формате строки символов unicode.

Предварительные объявления и процедура DriverEntry

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.

```
//////////////////////////////////////////////////
//////////////////
// init.cpp:            Инициализация драйвера
//   Замечание.  Рабочая  версия  данного  драйвера
должна быть
// скомпилирована как не-WDM версия. В противном
случае - драйвер
// не сможет корректно загружаться и выгружаться с
использованием
// программы monitor (пакет Numega Driver Studio) и
сервисов SCM
// Менеджера.

//////////////////////////////////////////////////
//////////////////
//    DriverEntry           Главная  точка  входа  в
драйвер
//    UnloadRoutine         Процедура       выгрузки
драйвера
//    DeviceControlRoutine                Обработчик
DeviceIoControl IRP пакетов
//////////////////////////////////////////////////
//////////////////
#include "Driver.h"

// Предварительные объявления функций:

VOID     UnloadRoutine  (IN           PDRIVER_OBJECT
DriverObject);
NTSTATUS Read                 (IN     PDEVICE_OBJECT
fdo, IN PIRP Irp);
NTSTATUS Write                (IN     PDEVICE_OBJECT
fdo, IN PIRP Irp);
NTSTATUS Create               (IN     PDEVICE_OBJECT
fdo, IN PIRP Irp);
NTSTATUS Close                (IN     PDEVICE_OBJECT
fdo, IN PIRP Irp);
```

```
// Глобальные переменные в драйвере
unsigned char *buff;
unsigned char data = 0x21;

#pragma code_seg("INIT") // начало секции INIT
//////////////////////////////////////////////////
//////////////////
// (Файл init.cpp)
//    DriverEntry   -   инициализация   драйвера   и
необходимых объектов
//    Аргументы:  указатель на объект драйвера
//                  раздел реестра (driver service
key) в UNICODE
//  Возвращает:   STATUS_Xxx

extern "C"

NTSTATUS        DriverEntry(IN        PDRIVER_OBJECT
DriverObject,
                                IN    PUNICODE_STRING
RegistryPath)
{
      NTSTATUS status = STATUS_SUCCESS;
      PDEVICE_OBJECT  fdo;
      UNICODE_STRING  devName;

      #if DBG
      DbgPrint("Example: In DriverEntry.");
      DbgPrint("Example:   RegistryPath   =   %ws.",
RegistryPath->Buffer);
      #endif

      DriverObject->DriverUnload = UnloadRoutine;
      DriverObject->MajorFunction[IRP_MJ_CREATE]=
Create;
      DriverObject->MajorFunction[IRP_MJ_CLOSE]    =
Close;
      DriverObject->MajorFunction[IRP_MJ_READ]     =
Read;
      DriverObject->MajorFunction[IRP_MJ_WRITE]    =
Write;
```

```
	// Действия по созданию символьной ссылки
	RtlInitUnicodeString(&devName,
L"\\Device\\EXAMPLE1");

	// Создаем наш Functional Device Object (FDO)
и получаем
	// указатель на созданный FDO в нашей
переменной fdo.
	status = IoCreateDevice(DriverObject,

	sizeof(EXAMPLE_DEVICE_EXTENSION),
                                        &devName,
// может быть и NULL

	FILE_DEVICE_UNKNOWN,
                                        0,
                                        FALSE,  //
без эксклюзивного доступа
                                        &fdo);
	//адрес буфера находится в поле
AssociatedIrp.SystemBuffer IRP пакета
	fdo->Flags |= DO_BUFFERED_IO;

	if(!NT_SUCCESS(status)) return status;
	//  Получаем  указатель  на  область,
предназначенную под
	// структуру расширение устройства
	PEXAMPLE_DEVICE_EXTENSION dx =
		(PEXAMPLE_DEVICE_EXTENSION)fdo-
>DeviceExtension;
	dx->fdo = fdo;  //  Сохраняем  обратный
указатель

	#if DBG
	DbgPrint("Example:  FDO  %X,
DevExt=%X.",fdo,dx);
	#endif

	// Создаем символьную ссылку
	UNICODE_STRING symLinkName;
```

```
	#define	SYM_LINK_NAME
	L"\\DosDevices\\Example1"
	RtlInitUnicodeString(&symLinkName,
SYM_LINK_NAME);
	dx->ustrSymLinkName = symLinkName;
	// Создаем символьную ссылку
	status = IoCreateSymbolicLink(&symLinkName,
&devName);
	if (!NT_SUCCESS(status))
	{ // при неудаче - удалить Device Object и
вернуть управление
		IoDeleteDevice(fdo);
		return status;
	}
	#if DBG
	DbgPrint("Example: DriverEntry successfully
completed.");
	#endif
	return status;
}
#pragma code_seg() // end INIT section
```

Рассмотрим приведенный код. Первый блок:

```
	VOID	UnloadRoutine (IN	PDRIVER_OBJECT
DriverObject);
	NTSTATUS Read		(IN
PDEVICE_OBJECT fdo, IN PIRP Irp);
	NTSTATUS Write		(IN
PDEVICE_OBJECT fdo, IN PIRP Irp);
	NTSTATUS Create		(IN
PDEVICE_OBJECT fdo, IN PIRP Irp);
	NTSTATUS Close		(IN
PDEVICE_OBJECT fdo, IN PIRP Irp);
```

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

Следующий блок:

```
unsigned char *buff;
unsigned char data = 0x21;
```

содержит объявления глобальных переменных: buff – указателя на unsigned char, предназначенный для хранения адреса системного буфера и data – unsigned char переменной для хранения данных при чтении и записи, которой присвоено произвольно выбранное однобайтное число 33 записанное в шестнадцатеричном виде. Тип unsigned char выбран, так как размер переменных данного типа – 1 байт.

Блок:

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT
DriverObject,
                     IN PUNICODE_STRING
RegistryPath)
{…}
```

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.

Первый блок функции:

```
NTSTATUS status = STATUS_SUCCESS;
PDEVICE_OBJECT  fdo;
```

```
UNICODE_STRING  devName;
```

содержит объявление локальных переменных. Переменная status типа NTSTATUS предназначена для хранения возвращаемого статуса (ей присвоена системная константа STATUS_SUCCESS, соответствующая успешному завершению операции). Переменная fdo типа PDEVICE_OBJECT – указатель на структуру типа DEVICE_OBJECT объекта устройства, с которым работает драйвер. Переменная devName типа UNICODE_STRING – имя устройства в формате Unicode.

Блок:

```
DriverObject->DriverUnload = UnloadRoutine;
DriverObject->MajorFunction[IRP_MJ_CREATE]=
Create;
DriverObject->MajorFunction[IRP_MJ_CLOSE]   =
Close;
DriverObject->MajorFunction[IRP_MJ_READ]    =
Read;
DriverObject->MajorFunction[IRP_MJ_WRITE]   =
Write;
```

записывает в поле DriverUnload объекта драйвера адрес функции UnloadRoutine, 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.

Инструкция:

```
RtlInitUnicodeString(&devName,
L"\\Device\\EXAMPLE1");
```

преобразует строку "\\Device\\EXAMPLE1" (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.

Инструкция:

```
status = IoCreateDevice(DriverObject,

sizeof(EXAMPLE_DEVICE_EXTENSION),
                        &devName,
                        FILE_DEVICE_UNKNOWN,
                        0,
                        FALSE,
                        &fdo);
```

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 FASLE; и адрес структуры объекта устройства.

Блок:

```
fdo->Flags |= DO_BUFFERED_IO;

if(!NT_SUCCESS(status)) return status;

PEXAMPLE_DEVICE_EXTENSION dx =
      (PEXAMPLE_DEVICE_EXTENSION)fdo-
>DeviceExtension;
dx->fdo = fdo;
```

первой инструкцией 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. Вторая 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. Третья инструкция извлекает из созданного объекта устройства указатель на структуру расширения устройства и приводит к определенному нам типу указателя PEXAMPLE_DEVICE_EXTENSION. Заключительная инструкция записывает в структуру расширения устройства адрес структуры объекта устройства (это необходимо для работы процедуры выгрузки).

Заключительный блок:

```
	UNICODE_STRING symLinkName;
	#define	SYM_LINK_NAME
	L"\\DosDevices\\Example1"
	RtlInitUnicodeString(&symLinkName,
SYM_LINK_NAME);
	dx->ustrSymLinkName = symLinkName;

	status = IoCreateSymbolicLink(&symLinkName,
&devName);
	if (!NT_SUCCESS(status))
	{
		IoDeleteDevice(fdo);
		return status;
	}
	return status;
```

содержит действия по созданию символьной ссылки на устройство. 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 удаляет объект устройства и возвращает ошибочный статус. Завершающая инструкция возвращает успешный статус.

Определение функции DriverEntry() заключено в блок:

```
#pragma code_seg("INIT")
…
#pragma code_seg()
```

Это означает, что после выполнения функции DriverEntry она будет полностью выгружена из памяти (так как новых обращений к ней уже не предполагается).

Функция CompleteIrp

Вспомогательная функция CompleteIrp реализует действия по завершению обработки IRP пакета с заданным кодом завершения. Данная функция предназначена для внутренних нужд драйвера и нигде не регистрируется. Ее параметры – указатель на IRP пакет, статус, численный параметр info (при значении, отличном от нуля, чаще всего содержит число байт, переданных клиенту или

полученных от клиента драйвера). Возвращаемое значение – переданный в функцию статус.

```
NTSTATUS CompleteIrp( PIRP Irp, NTSTATUS
status, ULONG info)
{
      Irp->IoStatus.Status = status;
      Irp->IoStatus.Information = info;
      IoCompleteRequest(Irp,IO_NO_INCREMENT);
      return status;
}
```

Первой инструкцией полю Status структуры IoStatus, которая в свою очередь является полем структуры IRP-пакета присваивается переданный статус. Второй инструкцией полю Information структуры IoStatus присваевается переданный численный параметр info. Третьей инструкцией вызывается системная функция IoCompleteRequest, которая информирует диспетчер ввода-вывода о том, что драйвер завершил обработку запроса на ввод или вывод и возвращает пакет диспетчеру. Параметр IO_NO_INCREMENT – константа, сигнализирующая о том, что у вызвавшего операцию ввода-вывода потока не поменяется приоритет в связи с ожиданием ввода-вывода. Завершающая инструкция возвращает переданный в функцию статус операции.

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. В данном примере наша функция обработки запросов чтения/записи ничего полезного не делает, и ее регистрация выполнена только для демонстрации, как это могло бы быть в более «развитом» драйвере.

```
// Read/Write: Берут на себя обработку запросов
// чтения/записи и завершает обработку IRP вызовом
CompleteIrp
// с числом переданных/полученных байт (BytesTxd)
равным нулю.
// Аргументы:
// Указатель на объект нашего FDO
// Указатель на структуру IRP, поступившего от
Диспетчера ввода/вывода
NTSTATUS Read(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
{
	// Задаем печать отладочных сообщений
	#if DBG
	DbgPrint("Example: in Read.");
	#endif
	ULONG	BytesTxd	=	0;//	Число
переданных/полученных байт (пока 0)
	NTSTATUS	status	=	STATUS_SUCCESS;
//Завершение с кодом status
	PIO_STACK_LOCATION
IrpStack=IoGetCurrentIrpStackLocation(Irp);
	//	Получаем	указатель	на	расширение
устройства
	PEXAMPLE_DEVICE_EXTENSION dx =
		(PEXAMPLE_DEVICE_EXTENSION)fdo-
>DeviceExtension;
	// Записываем в буффер содержимое data
	buff		=		(PUCHAR)Irp-
>AssociatedIrp.SystemBuffer;
	*buff = data;
	// Передан 1 байт
	BytesTxd = 1;
	// Получаем текущее значение уровня IRQL -
приоритета,
```

```
	// на котором выполняется поток (вообще
говоря, целое число):
	KIRQL irql,
		currentIrql = KeGetCurrentIrql();
	#if DBG
	DbgPrint("Example: In Read (fdo= %X)\n",fdo);
	if(currentIrql==PASSIVE_LEVEL)
		DbgPrint("Example:		PASSIVE_LEVEL
(val=%d)",currentIrql);
	#endif
	// Завершение IRP
	return CompleteIrp(Irp,status,BytesTxd);
}

NTSTATUS Write(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
{
	// Задаем печать отладочных сообщений
	#if DBG
	DbgPrint("Example: in Write.");
	#endif
	ULONG	BytesTxd	=	0;//	Число
переданных/полученных байт (пока 0)
	NTSTATUS	status	=	STATUS_SUCCESS;
//Завершение с кодом status
	PIO_STACK_LOCATION
IrpStack=IoGetCurrentIrpStackLocation(Irp);
	//	Получаем	указатель	на	расширение
устройства
	PEXAMPLE_DEVICE_EXTENSION dx =
		(PEXAMPLE_DEVICE_EXTENSION)fdo-
>DeviceExtension;
	#if DBG
	DbgPrint("Example:		Method		:
DO_BUFFERED_IO.");
	#endif
	// Записываем в data содержимое буффера
	buff		=		(PUCHAR)Irp-
>AssociatedIrp.SystemBuffer;
	data = *buff;
	BytesTxd = 1;
	// Получаем текущее значение уровня IRQL -
приоритета,
```

```
	// на котором выполняется поток (вообще
говоря, целое число):
	KIRQL irql,
		currentIrql = KeGetCurrentIrql();
	#if DBG
	DbgPrint("Example: In Write (fdo=
%X)\n",fdo);
	if(currentIrql==PASSIVE_LEVEL)
		DbgPrint("Example: PASSIVE_LEVEL
(val=%d)",currentIrql);
	#endif
	return CompleteIrp(Irp,status,BytesTxd); //
Завершение IRP
}
```

Рассмотрим код функций подробнее. Функция:

```
	NTSTATUS Read(IN PDEVICE_OBJECT fdo, IN PIRP
Irp)
```

отвечает за операцию чтения. Адрес данной функции при загрузке драйвера (DriverEntry) 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 поступает указатель на объект устройства и указатель на IRP пакет. Возвращает функция статус совершенной операции.

Первый блок функции:

```
	ULONG BytesTxd = 0;
	NTSTATUS status = STATUS_SUCCESS;
	PIO_STACK_LOCATION
IrpStack=IoGetCurrentIrpStackLocation(Irp);
	PEXAMPLE_DEVICE_EXTENSION dx =
```

```
            (PEXAMPLE_DEVICE_EXTENSION)fdo-
>DeviceExtension;
```

проводит подготовительные операции: объявляет переменную для хранения числа переданных байт (начальное значение – ноль) и переменную статуса операции (начальное значение – STATUS_SUCCESS, успешная операция). Третья инструкция вызовом функции IoGetCurrentIrpStackLocation получает адрес вершины IRP-стека по переданному IRP-пакету. Сам адрес в дальнейшем не используется, но данный вызов необходим для корректного получения параметров пакета (таковы механизмы работы диспетчера ввода-вывода). Четвертая инструкция извлекает адрес структуры расширения устройства из структуры объекта устройства.

Завершающий блок:

```
      buff              =              (PUCHAR)Irp-
>AssociatedIrp.SystemBuffer;
      *buff = data;
      BytesTxd = 1;
      return CompleteIrp(Irp,status,BytesTxd);
```

проводит основные операции, необходимые для осуществления чтения. В глобальную переменную buff записывается адрес системного буфера, извлекаемый поля структуры IRP AccociatedIrp.SystemBuffer. Второй инструкцией производится непосредственно чтение: в начало системного буфера копируется значение глобальной переменной data. Третья инструкция устанавливает счетчик прочитанных байт в единицу, четвертая – вызовом описанной выше функции CompleteIrp информирует диспетчер ввода-вывода об успешном завершении операции чтения. Таким образом, вызовы операций чтения для нашего устройства

возвращают текущее значение глобальной переменной data.

Функция:

```
NTSTATUS Write(IN PDEVICE_OBJECT fdo, IN PIRP
Irp)
```

отвечает за запись и во многом походит на функцию Read. Адрес данной функции при загрузке драйвера 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 (как и на вход функции Read) поступает указатель на объект устройства и указатель на IRP пакет, а возвращает она статус совершенной операции.

Начальный блок функции Write:

```
ULONG BytesTxd = 0;
NTSTATUS status = STATUS_SUCCESS;
PIO_STACK_LOCATION
IrpStack=IoGetCurrentIrpStackLocation(Irp);
PEXAMPLE_DEVICE_EXTENSION dx =
      (PEXAMPLE_DEVICE_EXTENSION)fdo-
>DeviceExtension;
buff              =              (PUCHAR)Irp-
>AssociatedIrp.SystemBuffer;
```

совпадает с соответствующим блоком Read (объявление переменных, получение вершины стека, извлечение расширения устройства, получение адреса системного буфера).

Следующий блок:

```
data = *buff;
BytesTxd = 1;
return CompleteIrp(Irp,status,BytesTxd);
```

производит запись: в глобальную переменную data копируется значение, хранящееся в начале системного буфера (адрес которого был извлечен из IRP пакета), счетчик переданных байт устанавливается в единицу, диспетчер информируется о завершении операции вывода. Таким образом, вызовы операций чтения драйвера изменяют значение переменной data на переданное при вызове.

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. Для ассоциирования данной функции с обработкой create-пакетов при выполнении DriverEntry адрес данной функции был записан в массив MajorFunctions объекта драйвера по индексу IRP_MJ_CREATE. В нашем примере эта функция не выполняет никаких действий, и лишь сигнализирует диспетчеру ввода-вывода о завершении операции (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).

```
// Create: Берет на себя обработку запросов с
```

```
// кодом IRP_MJ_CREATE.
// Аргументы:
// Указатель на объект нашего FDO
// Указатель на структуру IRP, поступившего от
Диспетчера ВВ
NTSTATUS Create(IN PDEVICE_OBJECT fdo,IN PIRP Irp)
{
      PIO_STACK_LOCATION         IrpStack         =
IoGetCurrentIrpStackLocation(Irp);
      #if DBG
      DbgPrint("Example: Create File is %ws",
            &(IrpStack->FileObject-
>FileName.Buffer));
      #endif
      return  CompleteIrp(Irp,STATUS_SUCCESS,0);  //
Успешное завершение
}
```

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 (регистрация производится записью адреса функции в массив MajorFunctions по индексу IRP_MJ_CLOSE при выполнении DriverEntry). 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).

```
// Close: Берет на себя обработку запросов с
// кодом IRP_MJ_CLOSE.
// Аргументы:
// Указатель на объект нашего FDO
//  Указатель на структуру IRP, поступившего от
Диспетчера
// ввода/вывода
NTSTATUS Close(IN PDEVICE_OBJECT fdo,IN PIRP Irp)
{
      #if DBG
      DbgPrint("Example: In Close handler.");
      #endif
      return   CompleteIrp(Irp,STATUS_SUCCESS,0);//
Успешное завершение
}
```

Рабочая процедура выгрузки драйвера

Процедура UnloadRoutine выполняет завершающую работу перед тем как драйвер будет выгружен системой.

```
//  UnloadRoutine: Выгружает драйвер, освобождая
оставшиеся объекты
// Вызывается системой, когда необходимо выгрузить
драйвер.
// Как и процедура AddDevice, регистрируется иначе
чем
//  все остальные рабочие процедуры и не получает
никаких IRP.
// Arguments:  указатель на объект драйвера
#pragma code_seg("PAGE")
// Допускает размещение в странично организованной
памяти
VOID UnloadRoutine(IN PDRIVER_OBJECT pDriverObject)
{
      PDEVICE_OBJECT    pNextDevObj;
```

```
	int i;
	// Задаем печать отладочных сообщений – если
сборка отладочная
	#if DBG
	DbgPrint("Example: In Unload Routine.");
	#endif
	pNextDevObj = pDriverObject->DeviceObject;
	for(i=0; pNextDevObj!=NULL; i++)
	{
		PEXAMPLE_DEVICE_EXTENSION dx =

	(PEXAMPLE_DEVICE_EXTENSION)pNextDevObj-
>DeviceExtension;
		//   Удаляем   символьную   ссылку   и
уничтожаем FDO:
		UNICODE_STRING  *pLinkName  =  &  (dx-
>ustrSymLinkName);
		// сохраняем указатель:
		pNextDevObj = pNextDevObj->NextDevice;
		#if DBG
		DbgPrint("Example: Deleted device (%d)
: pointer to FDO = %X.",
			i,dx->fdo);
		DbgPrint("Example:  Deleted  symlink  =
%ws.", pLinkName->Buffer);
		#endif
		IoDeleteSymbolicLink(pLinkName);
		IoDeleteDevice(dx->fdo);
	}
}
#pragma code_seg() // end PAGE section
```

Функция UnloadRoutine принимает в качестве параметра на вход указатель на структуру объекта драйвера. Начальный блок:

```
	PDEVICE_OBJECT	pNextDevObj;
	int i;
	pNextDevObj = pDriverObject->DeviceObject;
```

содержит объявления необходимых переменных: указателя на объект устройства, необходимый для обхода всех объектов устройств, ассоциированных с драйвером (в нашем случае такое устройство всего одно) и переменной цикла i. Текущим объектом устройством назначается устройство, извлеченное из объекта драйвера.

Цикл:

```
for(i=0; pNextDevObj!=NULL; i++)
{
      PEXAMPLE_DEVICE_EXTENSION dx =

(PEXAMPLE_DEVICE_EXTENSION)pNextDevObj-
>DeviceExtension;
      UNICODE_STRING *pLinkName = & (dx-
>ustrSymLinkName);
      pNextDevObj = pNextDevObj->NextDevice;
      IoDeleteSymbolicLink(pLinkName);
      IoDeleteDevice(dx->fdo);
}
```

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, второй – из расширения извлекается строка символьной ссылки. Третья инструкция извлекает из текущего объекта устройства ссылку на следующее (в нашем примере при первом же выполнении данной инструкции вернется NULL). Четвертая инструкция вызывает системную функцию IoDeleteSymbolicLink, удаляющую 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.

Определение функции UnloadRoutine() заключено в блок:

```
#pragma code_seg("PAGE")
…
#pragma code_seg()
```

Это означает, что код функции размещается в странично-организованной области памяти.

Компиляция и сборка драйвера Example.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=Example
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.

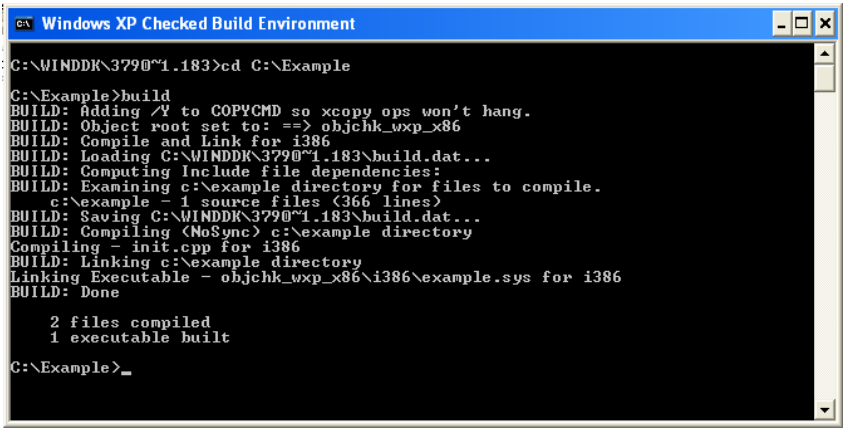


Рисунок 1 – Рабочее окно сборки драйвера под Windows XP DDK версии checked

Тестирование драйвера

Работа с драйвером Example.sys

Как уже было сказано, из всех возможных способов инсталляции и запуска драйвера Example.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 ExampleTest под отладчиком в пошаговом режиме.

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

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 работу 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 Example.sys).

```
//////////////////////////////////////////////////
/////////////////
// (Файл ExampleTest.cpp)
// Консольное приложение для тестирования драйвера
Example.sys
//////////////////////////////////////////////////
/////////////////
// Заголовочные файлы, которые необходимы в данном
приложении:
#include <windows.h>
#include <stdio.h>
#include <winioctl.h>
#include <tchar.h>
//   Имя   объекта   драйвера   и   местоположение
загружаемого файла
#define DRIVERNAME    _T("Example")
#define                                  DRIVERBINARY
_T("C:\\Example\\ExampleDriver\\Example.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 - Example
                              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:   CanTt   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( "\\\\.\\Example",
                              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
Example.sys !\n");
            return (-1);
      }
      DWORD  BytesReturned;      // Переменная для
хранения числа переданных байт
      //  Последовательно  выполняем  обращения  к
драйверу
      // с запросами на запись/чтение данных:
      unsigned char xdata = 0x21;
            if( !WriteFile(   hHandle,
                              &xdata,
                              sizeof(xdata),
                              &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);
      // Получаем 1 байт данных из драйвера.
      // По окончании данного вызова переменная
xdata должна
      // содержать значение 0x21:
      if( !ReadFile( hHandle,
                     &xdata,
                     sizeof(xdata),
                     &BytesReturned,
                     NULL))
      {
            printf( "Error with byte receive!" );
            return(-1);
      }
      // Вывод диагностического сообщения в
консольном окне:
      printf("Byte receive: BytesReturned=%d
xdata=%d\r\n",
            BytesReturned, xdata);
      // 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;
}
```

ЛАБОРАТОРНАЯ РАБОТА № 2.

ИЗУЧЕНИЕ ДРАЙВЕРА ВИРТУАЛЬНОГО УСТРОЙСТВА. ПРОГРАММНО УПРАВЛЯЕМАЯ ПЕРЕДАЧА ДАННЫХ.

Цель работы: изучение драйвера виртуального устройства, предназначенного для организации программно управляемой передачи данных (Programmed input/output, PIO).

Примечание. Приведенный ниже код драйвера PIO.sys является завершенным драйвером, который готов к компиляции и использованию в операционной системе в качестве тестового примера. По своей сути, приведенный код не может быть полноценным WDM драйвером (в силу отсутствия в нем некоторых основных рабочих процедур драйверов PnP устройств), хотя и может быть успешно скомпилирован. Драйвер PIO.sys более подходит под описание "монолитный драйвер в-стиле-NT".

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:

4. Ознакомиться с механизмом передачи данных посредством программируемого ввода-вывода и регистрами устройства.
5. Ознакомиться с описанием виртуального устройства.
6. Выполнить компиляцию и сборку драйвера средствами DDK.
7. Провести тестирование драйвера.

ВЫПОЛНЕНИЕ РАБОТЫ

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.

Программируемый ввод/вывод

Устройства, использующие программируемый ввод/вывод (Programmed I/O, PIO), осуществляют передачу данных непосредственно через регистры устройства. Драйвер должен использовать инструкцию ввода/вывода (I/O instruction) для чтения из регистра или записи в регистр устройства каждого байта данных. При больших объемах драйвер должен поддерживать адресацию буферной области и счетчик переданных данных.

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

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 объявления, необходимые для компиляции драйвера PIO.sys.

```
//=================================================
==========================
// Driver.h - заголовочный файл для драйвера
//=================================================
==========================
#pragma once
extern "C" {
#include <NTDDK.h>
}
/* 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 START_READ_CMD  0x00000004 //начать приём
пакетов => вкл. прерывания
/* Статус устройства PCI */
#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)
      //=========================================
==========================
      PUCHAR                  portBase;   // адрес
порта ввода/вывода
      //=========================================
==========================
      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.

Первый 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. Драйвер программируемого ввода-вывода использует только два регистра устройства – регистр состояния с адресом STATUS_REG и регистр данных с адресом DATA_REG.

Второй блок констант содержит коды команд устройства. Драйвер программируемого ввода-вывода данные константы не использует.

Третий блок констант содержит коды состояний (статусов) устройства. Рассматриваемый драйвер

использует только константу кода состояния BUFF_IS_EMPTY - устройство входит в данное состояние тогда, когда внутренний буфер чтения устройства пуст (данных не поступало, либо все данные уже прочитаны).

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), базовый адрес устройства, и объект прерывания, используемый устройством (в рассматриваемом драйвере не используется).

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.c
//=================================================
=======================
#include "driver.h"
// Предварительные объявления функций
static NTSTATUS CreateDevice (IN PDRIVER_OBJECT
pDriverObject,
                              IN ULONG portBase);
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);
//=================================================
=======================
// 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("PIO: 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, 0xD240);
return status;
}
//================================================
========================
// Функция:     CreateDevice
// Назначение:  Создание устройства с точки зрения
системы
// Аргументы: pDriverObject - поступает от
Диспетчера ввода/вывода
```

```
//            portBase - адрес базового регистра
порта ввода/вывода
// Возвращаемое значение:
//            NTSTATUS - в случае нормального
завершения STATUS_SUCCESS
//                        или код ошибки
STATUS_Xxx
NTSTATUS CreateDevice(IN PDRIVER_OBJECT
pDriverObject,
                      IN ULONG          portBase)
{
      NTSTATUS status;
      PDEVICE_OBJECT pDevObj;
      PDEVICE_EXTENSION pDevExt;
      // Создаем внутреннее имя устройства
      UNICODE_STRING devName;
      RtlInitUnicodeString(&devName,
L"\\Device\\PIO");
      // 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->portBase       =(PUCHAR)portBase;
      pDevExt->pIntObj        = NULL;
      //=========================================
=====
```

```
	// Создаем символьную ссылку:
	UNICODE_STRING symLinkName;
	// Для того, чтобы работало в Windows 98 & XP
:
	#define SYM_LINK_NAME L"\\DosDevices\\PIO"
	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:
		IoDeleteDevice(pDevObj);
		return status;
	}
	pDevExt->ustrSymLinkName = symLinkName;
	#if DBG
	DbgPrint("PIO: Symbolic Link is created: %ws.
\n",
                                pDevExt-
>ustrSymLinkName.Buffer);
	#endif
return STATUS_SUCCESS;
}
```

Рассмотрим приведенный код. Первый блок:

```
	static NTSTATUS CreateDevice(IN
	PDRIVER_OBJECT pDriverObject,
                                IN ULONG
	portBase);
	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);
```

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,
0xD240);
      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 и числовая константа – базовый адрес устройства, и возврат из функции статуса осуществленной операции.

Блок:

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

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

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\\PIO");
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\\PIO" (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->portBase        =(PUCHAR)portBase;
      pDevExt->pIntObj         = NULL;
```

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, базовым адресом устройства (преобразуемым из беззнакового длинного целого ULONG в указатель на беззнаковый символ PUCHAR). Так как рассматриваемый драйвер не использует прерывания, ссылка на объект прерывания выставляется в ноль.

Заключительный блок:

```
	UNICODE_STRING symLinkName;
	#define SYM_LINK_NAME L"\\DosDevices\\PIO"
	RtlInitUnicodeString(&symLinkName,
SYM_LINK_NAME);
	status = IoCreateSymbolicLink(&symLinkName,
&devName);
	if(!NT_SUCCESS(status))
	{
		IoDeleteDevice(pDevObj);
		return status;
	}
	pDevExt->ustrSymLinkName = symLinkName;
	return STATUS_SUCCESS;
```

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 удаляет объект устройства и возвращает ошибочный 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;
      #if DBG
      DbgPrint("PIO: in Write now\n");
      #endif
      ULONG BytesTxd = 0;// Число
переданных/полученных байт(пока 0)
      NTSTATUS status = STATUS_SUCCESS;
//Завершение с кодом status
      UCHAR *buff =(PUCHAR)pIrp-
>AssociatedIrp.SystemBuffer;
      while(InputLength > 0)
```

```
	{
		WRITE_PORT_UCHAR((PUCHAR)(pDevExt-
>portBase + DATA_REG), *buff);
		buff++;
		InputLength--;
		BytesTxd++;
	}
	// Получаем текущее значение уровня IRQL –
приоритета
	KIRQL Irql = KeGetCurrentIrql();
	#if DBG
	if(Irql == PASSIVE_LEVEL)
		DbgPrint("PIO:
PASSIVE_LEVEL(val=%d)",Irql);
	#endif
	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)
{
	#if DBG
	DbgPrint("PIO: in Read now\n");
	#endif
	PDEVICE_EXTENSION  pDevExt =
	(PDEVICE_EXTENSION)pDevObj->DeviceExtension;
```

```
	PIO_STACK_LOCATION pIrpStack =
IoGetCurrentIrpStackLocation(pIrp);
	UCHAR *buff =(PUCHAR)pIrp-
>AssociatedIrp.SystemBuffer;
	ULONG BytesTxd = 0; // Число
переданных/полученных байт(пока 0)
	NTSTATUS status = STATUS_SUCCESS;
//Завершение с кодом status
	*buff = READ_PORT_UCHAR((PUCHAR)(pDevExt-
>portBase + DATA_REG));
	while(READ_PORT_UCHAR((PUCHAR)(pDevExt-
>portBase + STATUS_REG))!=(UCHAR)BUFF_IS_EMPTY)
	{
		BytesTxd++;
		*(buff + BytesTxd)=
READ_PORT_UCHAR((PUCHAR)(pDevExt->portBase +
DATA_REG));
		#if DBG
		DbgPrint("%c | %c",*(buff + BytesTxd),
BytesTxd);
		#endif
	}
	BytesTxd++;
	// Получаем текущее значение уровня IRQL -
приоритета
	KIRQL Irql = KeGetCurrentIrql();
	#if DBG
	if(Irql == PASSIVE_LEVEL)
		DbgPrint("PIO:
PASSIVE_LEVEL(val=%d)",Irql);
	#endif
	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. Возвращает функция статус совершенной операции.

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;
```

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.

Следующий блок отвечает непосредственно за пересылку данных устройству:

```
	while(InputLength > 0)
	{
		WRITE_PORT_UCHAR((PUCHAR)(pDevExt-
>portBase + DATA_REG), *buff);
		buff++;
		InputLength--;
		BytesTxd++;
	}
```

Механизм пересылки данных – побайтный. Тело цикла содержит три операции.

Первая обращается к HAL-макроопределению WRITE_PORT_UCHAR, которое записывает в порт с заданным адресом 1 байт данных. Первый аргумент макроса – конструкция:

```
(PUCHAR)(pDevExt->portBase + DATA_REG)
```

Конструкция:

```
pDevExt->portBase
```

представляет собой значение базового адреса устройства, извлекаемое из структуры расширения устройства. К ней прибавляется DATA_REG – смещение регистра данных относительно базового адреса. Таким образом результат данного сложения – фактический адрес регистра данных. Данное целочисленное значение преобразуется к указателю на однобайтный беззнаковый символ PUCHAR. Конструкция *buff извлекает значение из ячейки памяти с адресом buff.

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 побайтной пересылкой в порт данных устройства хранящихся в буфере данных в заданных переменной InputLength 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:

```
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).

Следующая инструкция:

```
      *buff    =    READ_PORT_UCHAR((PUCHAR)(pDevExt-
>portBase + DATA_REG));
```

производит чтение первого байта из устройства Она обращается к HAL-макроопределению READ_PORT_UCHAR, которое считывает из порта с заданным адресом 1 байт данных. Первый аргумент макроса – конструкция:

```
(PUCHAR)(pDevExt->portBase + DATA_REG)
```

Конструкция:

```
pDevExt->portBase
```

представляет собой значение базового адреса устройства, извлекаемое из структуры расширения устройства. К ней прибавляется DATA_REG – смещение регистра данных относительно базового адреса. Таким образом результат данного сложения – фактический адрес регистра данных. Данное целочисленное значение преобразуется к указателю на однобайтный беззнаковый символ PUCHAR. Возвращаемый макросом байт данных записывается в начальный элемент системного буфера.

Цикл:

```
      while(READ_PORT_UCHAR((PUCHAR)(pDevExt-
>portBase + STATUS_REG)) != (UCHAR)BUFF_IS_EMPTY)
      {
            BytesTxd++;
            *(buff + BytesTxd) =
            READ_PORT_UCHAR((PUCHAR)(pDevExt-
      >portBase + DATA_REG));
      }
      BytesTxd++;
```

обрабатывает чтение из буфера устройства.

Рассмотрим условие цикла. Конструкция:

```
READ_PORT_UCHAR((PUCHAR)(pDevExt->portBase + STATUS_REG)
```

аналогична ранее рассмотренным. С помощью макроопределения из регистра статуса устройства, имеющего смещение относительно базового адреса, равное STATUS_REG извлекается 1 байт данных – текущее состояние устройства. Полученное значение сравнивается с числовой константой BUFF_IS_EMTPY (статус: «Буфер пуст»), приведенной к типу UCHAR. Вход в тело цикла производится при неравенстве двух значений, т.е. тогда, когда в буфере устройства присутствуют данные.

В теле цикла счетчик считанных байт увеличивается на единицу, а затем производится чтение байта из регистра данных устройства, как в инструкции перед циклом. Отличие состоит только в том, что в теле цикла возвращаемый макросом байт данных записывается в элемент системного буфера с индексом, равным текущему значению счетчика байт (конструкция `*(buff + BytesTxd)` аналогична `buff[BytesTxd]`).

После цикла счетчик считанных байт увеличивается еще на единицу (т.к. не увеличивался при первом чтении до цикла). Таким образом из регистра данных устройства считывается 1 или более байт – все содержимое буфера устройства – и записывается в системный буфер.

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.

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. Для ассоциирования данной функции с обработкой create-пакетов при выполнении DriverEntry адрес данной функции был записан в массив MajorFunctions объекта драйвера по индексу IRP_MJ_CREATE. В нашем примере эта функция не выполняет никаких действий, и лишь сигнализирует диспетчеру ввода-вывода о завершении операции (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("PIO: 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 (регистрация производится записью адреса функции в массив MajorFunctions по индексу IRP_MJ_CLOSE при выполнении DriverEntry). 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("PIO: 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 перед тем как драйвер будет выгружен системой.

```
//================================================
=======================
// 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("PIO: 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 символьную ссылку:
            IoDeleteSymbolicLink(&pDevExt-
>ustrSymLinkName);
            #if DBG
            DbgPrint("PIO: 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;
      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, вторая инструкция вызывает системную функцию IoDeleteSymbolicLink, удаляющую символьную ссылку, извлеченную из расширения устройства. Третья инструкция извлекает из текущего объекта устройства ссылку на следующее (в нашем примере при первом же выполнении данной инструкции вернется NULL). Последняя инструкция удаляет из системы объект устройства вызовом функции IoDeleteDevice. Ссылка на структуру объекта устройства извлекается из структуры расширения.

Компиляция и сборка драйвера PIO.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=PIO
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.

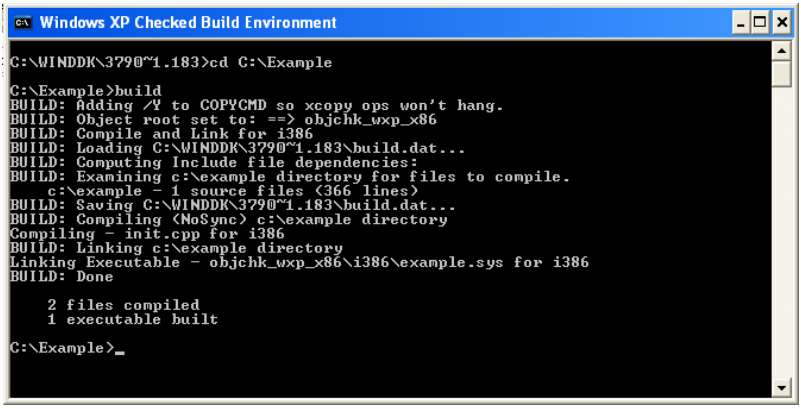


Рисунок 2 – Рабочее окно сборки драйвера под Windows XP DDK версии checked

Работа с драйвером PIO.sys

Как уже было сказано, из всех возможных способов инсталляции и запуска драйвера PIO.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 PIOTest под отладчиком в пошаговом режиме.

Перед запуском тестирующей программы PIOTest рекомендуется загрузить программу DebugView, чтобы в ее рабочем окне наблюдать сообщения, поступающие непосредственно из кода драйвера PIO.sys (отладочной сборки). Также перед запуском программы следует запустить программу PuTTY и настроить её на передачу/прием данных из виртуального устройства.

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 работу 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:

6. Обращение к функции OpenSCManager() для получения дескриптора диспетчера.

7. 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().

8. Обращение к StartService() для того, чтобы перевести службу в состояние функционирования.

9. Если служба запустилась успешно, то можно вызвать CreateFile(), для получения дескриптора, который мы будем использовать уже непосредственно при обращении к драйверу.

10. 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 PIO.sys).

```
// Заголовочные файлы, которые необходимы в данном
приложении:
#include <windows.h>
#include <stdio.h>
#include <winioctl.h>
#include <tchar.h>
#include <stdio.h>
// Имя объекта драйвера и местоположение
загружаемого файла
#define DRIVERNAME    _T("PIO")
#define DRIVERBINARY
_T("C:\\PIO\\PIODriver\\PIO.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 - PIO
                     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);
            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;
      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; // 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();
      }
      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("\\\\.\\PIO",
                              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
PIO.sys !\n");
            return(-1);
      }
      DWORD BytesReturned;    // Переменная для
хранения числа переданных байт
      // Получаем данные из драйвера
      char gets_line[100];
      unsigned char *xdata = new unsigned char [5];
      *(xdata)= 'T';
      *(xdata + 1)= 'E';
      *(xdata + 2)= 'S';
      *(xdata + 3)= 'T';
      printf("Write to DATA port\n");
      if(!WriteFile(hHandle,
                  xdata,
                  4,
                  &BytesReturned,
                  NULL))
      {
            printf("Error with byte receive!");
            return(-1);
      }
      printf("Read from DATA port");
      while(*gets(gets_line)!= '1')
      {
            printf("Printe some in PuTTY\r\n");
            if(!ReadFile(hHandle,
                        xdata,
                        4096,
                        &BytesReturned,
                        NULL))
            {
                  printf("Error with byte
receive!");
                  return(-1);
            }
            // Выводим данные в консольное окно
```

```
            while(BytesReturned > 0)
            {
                  BytesReturned--;
                  printf("%c", *xdata);
                  xdata++;
            }
      }
      // 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 № 1-2 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

PUDRV1часть.doc 605 Кбайт 20.02.2012