

ВСЕВОЛОД НЕСВИЖСКИЙ



ПРОГРАММИРОВАНИЕ АППАРАТНЫХ СРЕДСТВ В WINDOWS

2-е издание



ПОРТЫ ВВОДА-ВЫВОДА

ПРОГРАММИРОВАНИЕ
МЫШИ, КЛАВИАТУРЫ
СИСТЕМНЫХ УСТРОЙСТВ,
ДИСКОВОЙ ПОДСИСТЕМЫ

МОНИТОРИНГ ПИТАНИЯ,
ТЕМПЕРАТУР, ВИДЕО
И ЗВУКА

ИНТЕРФЕЙСЫ USB,
IEEE 1394 И ДР.

ОСОБЕННОСТИ
ПРОГРАММИРОВАНИЯ
В ОС WINDOWS ME/2000/XP
И VISTA

НЕДОКУМЕНТИРОВАННЫЕ
СПОСОБЫ ДОСТУПА
К ОБОРУДОВАНИЮ

ПРАКТИЧЕСКИЕ ПРИМЕРЫ
НА VISUAL C++ 6.0
И VISUAL STUDIO 2008

PRO

ПРОФЕССИОНАЛЬНОЕ
ПРОГРАММИРОВАНИЕ

+ CD

УДК 681.3.068
ББК 32.973.26-018.1
Н55

Несвижский В.

Н55 Программирование аппаратных средств в Windows. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2008. — 528 с.: ил. + CD-ROM — (Профессиональное программирование)

ISBN 978-5-9775-0263-4

Книга посвящена программированию базовых компонентов персонального компьютера: мыши, клавиатуры, процессора, системных устройств, дисковой подсистемы, а также систем мониторинга питания, температур, видео и звука. Уделено внимание популярным интерфейсам USB, IEEE 1394 и др. Рассмотрены особенности программирования в операционных системах Windows ME/2000/XP и Vista. Приведено большое количество простых и понятных примеров, написанных на языке C++. Для написания и отладки примеров были использованы оболочки Visual C++ 6.0 и Visual Studio 2008. Во втором издании рассмотрены особенности программирования для ОС Windows Vista. Прилагаемый компакт-диск содержит исходные коды всех примеров и системные драйверы для работы с аппаратными портами ввода-вывода.

Для программистов

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Игорь Шишигин</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Юрий Рожко</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн серии	<i>Инны Тачиной</i>
Оформление обложки	<i>Елены Беляевой</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 22.07.08.
Формат 70×100¹/₁₆. Печать офсетная. Усл. печ. л. 42,57.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0263-4

© Несвижский В., 2008
© Оформление, издательство "БХВ-Петербург", 2008

Оглавление

Введение	7
Программные требования	8
Поддержка	8
Глава 1. Общие сведения	9
1.1. Использование функций ввода-вывода	10
1.2. Использование функции <i>DeviceIoControl</i>	14
1.3. Использование драйвера	17
1.4. Использование ассемблера.....	27
1.5. Недокументированный доступ к портам.....	28
1.6. Определение параметров оборудования	33
1.7. Драйверы и Windows Vista.....	50
Глава 2. Мышь	51
2.1. Общие сведения	52
2.2. Использование портов	56
2.2.1 Команда <i>Reset (FFh)</i>	61
2.2.2. Команда <i>Resend (FEh)</i>	62
2.2.3. Команда <i>Set Defaults (F6h)</i>	63
2.2.4. Команда <i>Disable (F5h)</i>	63
2.2.5. Команда <i>Enable (F4h)</i>	63
2.2.6. Команда <i>Set Sample Rate (F3h)</i>	63
2.2.7. Команда <i>Read Device Type (F2h)</i>	65
2.2.8. Команда <i>Set Remote Mode (F0h)</i>	65
2.2.9. Команда <i>Set Wrap Mode (EEh)</i>	66
2.2.10. Команда <i>Reset Wrap Mode (ECh)</i>	66
2.2.11. Команда <i>Read Data (EBh)</i>	66
2.2.12. Команда <i>Set Stream Mode (EAh)</i>	66
2.2.13. Команда <i>Status Request (E9h)</i>	66

2.2.14. Команда <i>Set Resolution (E8h)</i>	70
2.2.15. Команда <i>Set Scaling 2:1 (E7h)</i>	70
2.2.16. Команда <i>Set Scaling 1:1 (E6h)</i>	70
2.3. Использование Win32 API	71
2.3.1. Настройка мыши	71
2.3.2. Работа с курсором	76
Глава 3. Клавиатура.....	81
3.1. Общие сведения	81
3.2. Использование портов	86
3.2.1. Команда <i>EDh</i>	90
3.2.2. Команда <i>EEh</i>	91
3.2.3. Команда <i>F2h</i>	91
3.2.4. Команда <i>F3h</i>	93
3.3. Использование Win32 API	100
3.3.1. Настройка клавиатуры	102
3.3.2. Использование "горячих" клавиш.....	104
3.3.3. Поддержка языков.....	108
Глава 4. Видеоадаптер.....	111
4.1. Общие сведения	111
4.2. Использование портов	112
4.2.1. Внешние регистры.....	114
4.2.2. Регистры графического контроллера.....	117
4.2.3. Регистры контроллера атрибутов.....	122
4.2.4. Регистры контроллера CRT	126
4.2.5. Регистры ЦАП	136
4.2.6. Регистры синхронизатора.....	138
4.3. Использование Win32 API	141
4.3.1. Управление графическими режимами.....	142
4.3.2. Проверка возможностей видеоадаптера.....	146
4.3.3. Управление монитором.....	148
Глава 5. Работа с видео	151
5.1. Использование MCI	152
5.2. Использование VFW	161
Глава 6. Звуковая карта	175
6.1. Использование портов	176
6.1.1. Цифровой процессор.....	177
6.1.2. Микшер	186
6.1.3. Интерфейс MIDI	196
6.2. Использование Win32 API	201

Глава 7. Работа со звуком.....	219
7.1. Создание плеера аудиодисков.....	219
7.2. Программирование MIDI	234
7.3. Доступ к файлам в формате MP3.....	241
Глава 8. Системный динамик.....	257
8.1. Программирование системного динамика.....	258
Глава 9. Часы реального времени	261
9.1. Использование портов	262
Глава 10. Таймер	269
Глава 11. Дисковая подсистема.....	275
11.1. Использование портов	275
11.1.1. Регистры флоппи-дисковода	276
11.1.2. Команды управления для флоппи-дисковода	282
11.1.3. Устройства АТА/АТАРІ	296
11.1.4. Команды управления для АТА/АТАРІ-устройств.....	302
11.2. Использование Win32 API	330
Глава 12. Пространство шины PCI	339
12.1. Общие сведения	340
12.2. Использование портов	356
12.2.1. Регистр конфигурации адреса	356
12.2.2. Регистр конфигурации данных.....	356
Глава 13. Контроллер DMA	367
Глава 14. Контроллер прерываний	375
14.1. Команда <i>ICW1</i>	377
14.2. Команда <i>ICW2</i>	377
14.3. Команда <i>ICW3</i>	377
14.4. Команда <i>ICW4</i>	378
14.5. Команда <i>OCW1</i>	378
14.6. Команда <i>OCW2</i>	379
14.7. Команда <i>OCW3</i>	380
Глава 15. Процессор.....	383
Глава 16. Аппаратный мониторинг системы.....	395

Глава 17. Параллельный и последовательный порты	421
17.1. Общие сведения	421
17.2. Использование портов	422
17.3. Использование Win32 API	432
Глава 18. Современные интерфейсы	437
18.1. Интерфейс USB	438
18.1.1. Структура запроса	440
18.1.2. Структура дескрипторов	449
18.1.3. Использование запросов	457
18.1.4. Регистры ввода-вывода	467
18.1.5. Регистры конфигурации	473
18.2. Интерфейс IEEE 1394	474
18.2.1. Описание регистров	475
18.3. Интерфейс Wireless	500
18.3.1. Регистры конфигурации шины PCI	501
18.3.2. Регистры аппаратных возможностей	503
18.3.3. Регистры радиуправления	505
18.3.4. Регистры хост контроллера	508
18.3.5. Команды и события	515
Приложение 1. Глоссарий	519
Приложение 2. Описание компакт-диска	523
Предметный указатель	524

ГЛАВА 1



Общие сведения

Прежде чем начинать программирование устройств в операционных системах семейства Windows, необходимо разобраться в основных принципах доступа к аппаратной части компьютера под этими системами. А они, к большому сожалению, довольно скудны и однообразны. Кроме того, с появлением Windows Vista возможность работы с оборудованием сводится практически к одному варианту — посредством драйверов. Существуют как минимум четыре официальных способа прямого доступа к оборудованию.

- ❑ Первый заключается в обычном использовании набора функции ввода-вывода: `_outp`, `_outpw`, `_outpd`, `_inp`, `_inpw`, `_inpd`. Они входят в состав библиотеки времени выполнения, но их применение очень сильно зависит от операционной системы. Практически все современные системы Windows не позволяют работать с этими функциями в свободном режиме.
- ❑ Второй способ базируется на применении универсальной функции ввода-вывода `DeviceIoControl`. Основное преимущество при ее использовании заключается в однозначной поддержке данной функции всеми системами Windows, начиная с Win 95 и заканчивая одной из последних — Windows Vista. Но у этой функции есть и серьезный недостаток — очень ограниченный диапазон применения. Да, она позволяет работать с дисковой системой, современными интерфейсами передачи данных, но получить прямой доступ к устройствам с ее помощью не удастся. Основное ее назначение сводится к трансляции предопределенных или пользовательских команд между низкоуровневыми драйверами устройств и конечными приложениями. В связи с этим, она "подчиняется" всем ограничениям и политикам безопасности, принятым в современных операционных системах Windows.
- ❑ Третий способ заключается в банальном создании драйвера (например, виртуального драйвера устройства) и позволяет получить неограниченный

доступ ко всем устройствам в системе. Кроме того, данный вариант прекрасно будет работать во всех операционных системах Windows. Основной недостаток заключается в относительной сложности написания самого драйвера, а также в необходимости создания отдельного варианта драйвера для каждой операционной системы. Например, если программа написана под Windows 98 и Windows 2000, то придется писать два разных драйвера под каждую систему. Следует заметить, что с появлением Windows Vista поменялась драйверная модель и правила написания драйверов. Теперь она называется WDF (Windows Driver Foundation). Подробнее о написании драйверов рассказано в конце данной главы.

- Последний способ заключается в использовании встроенного в Visual C++ макроассемблера. Он не позволит применить прерывания (будет "зависать" Windows), но вполне неплохо работает с аппаратными портами. Данный способ не подходит для современных операционных систем, в частности для Windows Vista.

Поскольку каждый из перечисленных способов заслуживает внимания, разберем их подробнее. Сразу замечу, что выбор одного из представленных вариантов будет зависеть в первую очередь от версии операционной системы, а уж затем — от решаемых программистом задач. И с этим ничего не поделаешь: фирма Microsoft с каждым годом все меньше и меньше оставляет возможностей прямого доступа к устройствам. С одной стороны, их можно понять, поскольку эти меры повышают общую надежность системы, но с другой стороны, блокируют развитие конкурентного и часто более качественного программного обеспечения. Как я уже говорил, сначала мы разберем официальные возможности, а затем рассмотрим существование альтернативного варианта.

1.1. Использование функций ввода-вывода

Существует шесть функций для работы с портами. Три из них используются для вывода и три для ввода данных. К функциям чтения данных относятся:

- `_inp` — позволяет считать один байт из указанного порта;
- `_inpw` — позволяет прочитать одно слово из указанного порта;
- `_inpd` — позволяет прочитать двойное слово из указанного порта.

Все эти функции имеют один аргумент, который должен указывать номер порта, из которого будут прочитаны данные. В зависимости от размера получаемых данных, нужно применять ту или иную функцию. В листинге 1.1 показано, как можно работать с этими функциями. Максимальное значение адресуемого порта ограничено 65535, что вполне достаточно для работы со всеми существующими в системе значениями.

Листинг 1.1. Пример работы с функциями чтения данных из порта

```
// подключаем необходимый файл определений
#include <conio.h>
// прочитаем значение базовой памяти в килобайтах
int GetBaseMemory ( )
{
    // объявляем переменные для получения младшего и старшего байтов
    BYTE lowBase = 0, highBase = 0;
    // читаем информацию из CMOS-памяти
    _outp ( 0x70, 0x15 );          // записываем номер первого регистра
    lowBase = _inp ( 0x71 );       // читаем младший байт
    _outp ( 0x70, 0x16 );          // записываем номер первого регистра
    highBase = _inp ( 0x71 );     // читаем старший байт
    // возвращаем размер базовой памяти в килобайтах
    return ( ( highBase << 8 ) | lowBase );
}
// напишем функцию для управления клавиатурой
void KeyBoard_OnOff ( bool bOff )
{
    BYTE state;                   // текущее состояние
    if ( bOff )                   // выключить клавиатуру
    {
        state = _inp ( 0x61 );    // получаем текущее состояние
        state |= 0x80;            // устанавливаем бит 7 в 1
        _outp ( 0x61, state );    // записываем обновленное значение в порт
    }
    else                           // включить клавиатуру
    {
        state = _inp ( 0x61 );    // получаем текущее состояние
        state &= 0x7F;            // устанавливаем бит 7 в 0

        _outp ( 0x61, state );    // записываем обновленное значение в порт
    }
}
```

Функции записи в порт имеют два аргумента. Первый позволяет указать номер порта, а второй служит для хранения передаваемых данных. К функциям записи данных относятся:

- `_outp` — позволяет записать один байт в указанный порт;
- `_outpw` — позволяет записать слово в указанный порт;
- `_outpd` — позволяет записать двойное слово в указанный порт.

После выполнения все эти функции возвращают переданное значение. Максимальное значение адресуемого порта также ограничено 65535. В листинге 1.2 представлены примеры работы с функциями записи.

Листинг 1.2. Пример работы с функциями записи данных в порт

```
// напишем функцию для программного сброса устройства ATA/ATAPI
bool ResetDrive ( )
{
    // первое устройство на втором канале ( обычно CD-ROM )
    _outp ( 0x177, 0x08 ); // пишем команду сброса 08h
    // проверяем результат выполнения
    for ( int i = 0; i < 5000; i++ )
    {
        // проверяем бит 7 BUSY
        if ( ( _inp ( 0x177 ) & 0x80 ) == 0x00 )
            return true; // команда успешно завершена
    }
    return false; // произошла ошибка
}

// напишем функцию для управления лотком CD-ROM
void Eject ( bool bOpen )
{
    int iTimeWait = 50000;
    // формат пакетной команды для открытия лотка
    WORD Eject[6]= { 0x1B, 0, 2, 0, 0, 0 };
    // формат пакетной команды для закрытия лотка
    WORD Close[6]= { 0x1B, 0, 3, 0, 0, 0 };
    // проверяем готовность устройства
    while ( -- iTimeWait > 0 )
    {
        // читаем состояние порта
        if ( ( _inp ( 0x177 ) & 0x80 == 0x00 ) &&
            ( _inp ( 0x177 ) & 0x08 == 0x00 ) ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return;
    }
    // выбираем первое устройство на втором канале
    _outp ( 0x176, 0xA0 );
    // перед посылкой пакетной команды следует проверить состояние
    iTimeWait = 50000;
    // ожидаем готовности устройства
    while ( -- iTimeWait > 0 )
```

```
{
    // читаем состояние порта
    if ( (_inp ( 0x177 ) & 0x80 == 0x00 ) &&
        ( _inp ( 0x177 ) & 0x08 == 0x00 ) ) break;
    // закончилось время ожидания
    if ( iTimeWait < 1 ) return;
}
// пишем в порт команду пакетной передачи A0h
_outp ( 0x177, 0xA0 );
// ожидаем готовности устройства к приему пакетной команды
iTimeWait = 50000;
// ожидаем готовности устройства
while ( -- iTimeWait > 0 )
{
    // читаем состояние порта
    if ( (_inp ( 0x177 ) & 0x80 == 0x00 ) &&
        ( _inp ( 0x177 ) & 0x08 == 0x01 ) ) break;
    // закончилось время ожидания
    if ( iTimeWait < 1 ) return;
}
// пишем в порт пакетную команду
if ( bOpen ) // открыть лоток
{
    for ( int i = 0; i < 6; i++)
    {
        _outpw ( 0x170, Eject[i] ); // 12-байтовая команда
    }
}
else // закрыть лоток
{
    for ( int j = 0; j < 6; j++)
    {
        _outpw ( 0x170, Close[j] ); // 12-байтовая команда
    }
}
// проверяем результат выполнения команды, если нужно
iTimeWait = 50000;
// ожидаем готовности устройства
while ( -- iTimeWait > 0 )
{
    // читаем состояние порта
    if ( (_inp ( 0x177 ) & 0x80 == 0x00 ) &&
        ( _inp ( 0x177 ) & 0x01 == 0x00 ) &&
        ( _inp ( 0x177 ) & 0x40 == 0x01 ) ) break;
```

```
// закончилось время ожидания
if ( iTimeWait < 1 ) return;
}
}
```

Как видите, работать с функциями ввода-вывода довольно легко и комфортно. Однако прямое их использование в коде возможно лишь в Windows 95. Для работы с ними в современных системах (например, Windows 2000) предварительно придется писать драйвер и загружать в память перед выполнением кода программы. Как это делается, я расскажу позднее, а сейчас поговорим о функции `DeviceIoControl`.

1.2. Использование функции *DeviceIoControl*

Вообще говоря, данная функция так или иначе использует системные драйверы для доступа к устройствам. Эти драйверы могут входить в состав операционной системы или поставляться разработчиком программного продукта. Универсальность функции состоит в том, что она работает практически с любым драйвером, который поддерживает операции ввода-вывода.

Функция `DeviceIoControl` имеет восемь аргументов. Первый позволяет указать имя драйвера, через который будут осуществляться управление портами (в нашем случае). Второй аргумент представляет собой идентификатор кода требуемой операции, поскольку стандартный драйвер поддерживает несколько (от одной до сотни) операций и необходимо конкретно указать ему, какая из них нужна в данный момент. Третий и четвертый аргументы позволяют указать буфер для передаваемых данных и его размер. Их следует применять для операции записи, иначе установить в `NULL`. Пятый и шестой служат для получения данных от устройства (указатель на буфер данных и размер буфера). Если они не используются, то следует установить значения в `NULL`. Седьмой указывает на количество реально полученных данных. Последний аргумент является указателем на структуру `OVERLAPPED`. Она используется при асинхронном вводе-выводе. Рассмотрим примеры работы с данной функцией в Windows.

В листинге 1.3 приведен пример функции, позволяющей читать данные с жесткого диска. Она будет работать только в Windows 95/98/ME.

Листинг 1.3. Чтение сектора диска

```
#include "stdafx.h"
#define VWIN32_DIOC_DOS_DRIVEINFO 6 // код функции драйвера
#define CF_FLAG 1 // флаг переноса
```

```
// дополнительные структуры
typedef struct _DIOC_REGISTERS
{
    DWORD reg_EBX;
    DWORD reg_EDX;
    DWORD reg_ECX;
    DWORD reg_EAX;
    DWORD reg_EDI;
    DWORD reg_ESI;
    DWORD reg_Flags;
} DIOC_REGISTERS;
#pragma pack ( 1 )
typedef struct _DATABLOCK
{
    DWORD dwStartSector; // номер начального сектора
    WORD wNumSectors; // количество секторов
    DWORD pBuffer; // указатель на буфер данных
} DATABLOCK;
#pragma pack ( 0 )
// пишем функцию чтения секторов с диска
bool ReadSector ( unsigned int uDrive, DWORD dwStartSector,
                 WORD wNumSectors, LPBYTE lpBuffer )
{
    HANDLE hDriver;
    DIOC_REGISTERS reg = { 0 };
    DATABLOCK data = { 0 };
    bool bResult;
    DWORD dwResult = 0;
    // инициализируем драйвер
    hDriver = CreateFile ( "\\.\vwin32", 0, 0, NULL, 0,
                        FILE_FLAG_DELETE_ON_CLOSE, 0 );
    // если драйвер недоступен, выходим из функции
    if ( hDriver == INVALID_HANDLE_VALUE ) return false;
    // заполняем структуру данных DATABLOCK
    data.dwStartSector = dwStartSector;
    data.wNumSectors = wNumSectors;
    data.pBuffer = ( DWORD ) lpBuffer;
    // заполняем управляющую структуру
    reg.reg_EAX = 0x7305; // функция 7305h прерывания 21h
    reg.reg_EBX = ( DWORD ) &data;
    reg.reg_ECX = -1;
    reg.reg_EDX = uDrive; // номер логического диска
```

```

// вызываем функцию DeviceIoControl
bResult = DeviceIoControl ( hDriver, VWIN32_DIOC_DOS_DRIVEINFO,
    &reg, sizeof ( reg ), &reg, sizeof ( reg ), &dwResult, 0 );
// если произошла ошибка, выходим из функции
if ( !bResult || ( reg.reg_Flags & CF_FLAG ) )
{
    CloseHandle ( hDriver );
    return false;
}
return true;
}
// пример использования функции ReadSector для чтения 2-х секторов диска
// номер логического диска может быть следующим: 0 – по умолчанию, 1 – А,
// 2 – В, 3 – С, 4 – D, 4 – Е и т. д.
// выделяем память для двух секторов жесткого диска
char* buffer = NULL;
buffer = new char[512*2];
// вызываем функцию чтения секторов
ReadSector ( 3, 0, 2, ( LPBYTE ) buffer );
// освобождаем память
delete [] buffer;

```

Для профессиональных систем (Windows NT, XP или 2000) использовать DeviceIoControl не нужно. Там достаточно открыть функцией CreateFile логический диск (даже CD-ROM) и с помощью ReadFile прочитать данные с диска. Хотя стоит отметить, что пользоваться функцией DeviceIoControl в этих системах (в том числе в Windows 2003 и Windows Vista) можно для других всевозможных целей, связанных с доступом к оборудованию.

Рассмотрим еще один пример для записи данных на жесткий логический диск (листинг 1.4).

Листинг 1.4. Запись сектора диска

```

// пишем функцию для записи сектора диска
bool WriteSector ( unsigned int uDrive, DWORD dwStartSector,
    WORD wNumSectors, LPBYTE lpBuffer )
{
HANDLE hDriver;
    DIOC_REGISTERS reg = { 0 };
    DATABLOCK data = { 0 };
    bool bResult;
    DWORD dwResult = 0;

```

```
// инициализируем драйвер
hDriver = CreateFile ( "\\.\vwin32", 0, 0, NULL, 0,
                    FILE_FLAG_DELETE_ON_CLOSE, 0 );
// если драйвер недоступен, выходим из функции
if ( hDriver == INVALID_HANDLE_VALUE ) return false;
// заполняем структуру данных DATABLOCK
data.dwStartSector = dwStartSector;
data.wNumSectors = wNumSectors;
data.pBuffer = ( DWORD ) lpBuffer;
// заполняем управляющую структуру
reg.reg_EAX = 0x7305; // функция 7305h прерывания 21h
reg.reg_EBX = ( DWORD ) &data;
reg.reg_ECX = -1;
reg.reg_EDX = uDrive; // номер логического диска
reg.reg_ESI = 0x6001;
// вызываем функцию DeviceIoControl
bResult = DeviceIoControl ( hDriver, VWIN32_DIOC_DOS_DRIVEINFO,
                          &reg, sizeof ( reg ), &reg, sizeof ( reg ), &dwResult, 0 );
// если произошла ошибка, выходим из функции
if ( !bResult || ( reg.reg_Flags & CF_FLAG ) )
{
    CloseHandle ( hDriver );
    return false;
}
return true;
}
```

В последующих главах книги будут приводиться дополнительные примеры использования функции `DeviceIoControl`.

1.3. Использование драйвера

Данный способ является наиболее гибким и позволяет получить доступ ко всем устройствам в системе. Единственная сложность возникает с написанием самого драйвера. Поскольку все примеры работы с портами в книге основаны на применении драйверов, специально для читателей книги я написал и отладил два драйвера: виртуальный драйвер устройства `VxD` (Windows 98/ME) и системный драйвер `SYS` (Windows NT/2000/XP/2003/Vista). О том, где их найти, сказано во введении к книге. Здесь же я подробно объясню, как ими пользоваться. Мы напишем два класса для использования этих драйверов. Первый класс рассчитан на работу в Windows 98/ME и представлен в листингах 1.5 и 1.6.

Листинг 1.5. Файл IO32.h

```
// IO32.h: interface for the CIO32 class.
#include <winioctl.h>
// определяем коды функций для чтения и записи
#define IO32_WRITEPORT CTL_CODE ( FILE_DEVICE_UNKNOWN, 1, \
                                METHOD_NEITHER, FILE_ANY_ACCESS )
#define IO32_READPORT CTL_CODE ( FILE_DEVICE_UNKNOWN, 2, \
                                METHOD_NEITHER, FILE_ANY_ACCESS )

// объявляем класс
class CIO32
{
public:
    CIO32 ( );
    ~CIO32 ( );
// общие функции
    bool InitPort ( ); // инициализация драйвера
    // функция для считывания значения из порта
    bool inPort ( WORD wPort, PDWORD pdwValue, BYTE bSize );
    // функция для записи значения в порт
    bool outPort ( WORD wPort, DWORD dwValue, BYTE bSize );
private:
// закрытая часть класса
    HANDLE hVxD; // дескриптор драйвера
    // управляющая структура
    #pragma pack ( 1 )
    struct tagPort32
    {
        USHORT wPort;
        ULONG dwValue;
        UCHAR bSize;
    };
    #pragma pack ( )
}; // окончание класса
```

Листинг 1.6. Файл IO32.cpp

```
#include "stdafx.h"
#include "IO32.h"
// реализация класса CIO32
// конструктор
CIO32 :: CIO32 ( )
```



```
{
    hVxD = NULL;
}
// деструктор
CIO32 :: ~CIO32 ( )
{
    if ( hVxD ) CloseHandle ( hVxD );
    hVxD = NULL;
}
// функции
bool CIO32 :: InitPort ( )
{
    // загружаем драйвер
    hVxD = CreateFile ( "\\.\io32port.vxd", 0, 0, NULL, 0,
        FILE_FLAG_DELETE_ON_CLOSE, NULL );
    // если драйвер недоступен, прощаемся
    if ( hVxD == INVALID_HANDLE_VALUE )
        return false;
    return true;
}
bool CIO32 :: inPort ( WORD wPort, PDWORD pdwValue, BYTE bSize )
{
    // если драйвер недоступен, прощаемся
    if ( hVxD == NULL ) return false;
    DWORD dwReturn;
    tagPort32 port;

    port.bSize = bSize;
    port.wPort = wPort;
    // читаем значение из указанного порта
    return DeviceIoControl ( hVxD, IO32_READPORT, &port,
        sizeof ( tagPort32 ), pdwValue, sizeof ( DWORD ), &dwReturn, NULL );
}
bool CIO32 :: outPort ( WORD wPort, DWORD dwValue, BYTE bSize )
{
    // если драйвер недоступен, прощаемся
    if ( hVxD == NULL ) return false;
    DWORD dwReturn;
    tagPort32 port;
    port.bSize = bSize;
    port.dwValue = dwValue;
    port.wPort = wPort;
```

```
// записываем значение в указанный порт
return DeviceIoControl ( hVxD, IO32_WRITEPORT, &port,
                        sizeof ( tagPort32 ), NULL, 0, &dwReturn, NULL );
}
```

Теперь у нас есть полноценный класс для работы с портами. Перед началом работы нужно вызвать функцию `InitPort` для загрузки виртуального драйвера устройства (`io32port.vxd`). После этого можно писать и читать любые существующие в системе порты ввода-вывода. Функции `inPort` и `outPort` имеют каждая по три аргумента. Первый позволяет указать номер порта. Второй предназначен для передачи или получения значения из порта, а третий определяет размер передаваемых данных. Драйвер поддерживает четыре типа данных: байт (1), слово (2), трехбайтовое значение (3) и двойное слово (4). Не забывайте правильно указывать размер данных, иначе результат будет некорректным. В листинге 1.7 показано, как следует работать с классом `CI032`.

Листинг 1.7. Пример использования класса `CI032`

```
// объявляем класс
CI032 io;
// инициализируем драйвер
io.InitPort ( );
// теперь можно работать с портами
// для примера включим системный динамик и после 4 секунд выключим
DWORD dwResult = 0;
// читаем состояние порта
io.inPort ( 0x61, &dwResult, 1 );
dwResult |= 0x03; // включаем
// записываем значение в порт
io.outPort ( 0x61, dwResult, 1 );
// пауза 4 секунды
Sleep ( 4000 );
// читаем состояние порта
io.inPort ( 0x61, &dwResult, 1 );
dwResult &= 0xFC; // выключаем
// записываем значение в порт
io.outPort ( 0x61, dwResult, 1 );
```

Теперь подготовим второй класс `CI032NT`, позволяющий работать в профессиональных системах (Windows NT/2000/XP/2003/Vista). Сразу отмечу некоторые особенности использования драйвера ядра в Windows Vista. Поскольку драйвер, представленный на компакт-диске, написан для 32-разрядных опе-

рациональных систем, он будет работать только в 32-разрядной версии Windows Vista. Более подробно об этих ограничениях и причинах вы можете прочитать в последней части данной главы. В листингах 1.8 и 1.9 представлены файлы определений и реализации.

Листинг 1.8. Файл IO32NT.h

```
#include <winioctl.h>
// определяем коды функций драйвера
#define FILE_DEVICE_WINIO 0x00008010
#define WINIO_IOCTL_INDEX 0x810
#define IOCTL_WINIO_ENABLEDIRECTIO CTL_CODE ( FILE_DEVICE_WINIO, \
        WINIO_IOCTL_INDEX + 2, METHOD_BUFFERED, FILE_ANY_ACCESS )
#define IOCTL_WINIO_DISABLEDIRECTIO CTL_CODE ( FILE_DEVICE_WINIO, \
        WINIO_IOCTL_INDEX + 3, METHOD_BUFFERED, FILE_ANY_ACCESS )
// объявляем класс
class CIO32NT
{
public:
    CIO32NT ( );
    ~CIO32NT ( );
// общие функции
    bool InitPort ( ); // инициализация драйвера
    // функция для считывания значения из порта
    void inPort ( WORD wPort, PDWORD pdwValue, BYTE bSize );
    // функция для записи значения в порт
    void outPort ( WORD wPort, DWORD dwValue, BYTE bSize );
private:
// закрытая часть класса
    HANDLE hSYS; // дескриптор драйвера
// служебные функции
    // загрузка сервиса
    bool _loadService ( PSTR pszDriver );
    bool _goService ( ); // запуск сервиса
    bool _stopService ( ); // остановка сервиса
    bool _freeService ( ); // закрытие сервиса
}; // окончание класса
```

Листинг 1.9. Файл IO32NT.cpp

```
#include "stdafx.h"
#include "IO32NT.h"
```

```
#include <conio.h>
#include <Winsvc.h>
// реализация класса CIO32NT
// конструктор
CIO32NT :: CIO32NT ( )
{
    hSYS = NULL;
}
// деструктор
CIO32NT :: ~CIO32NT ( )
{
    DWORD dwReturn;
    if ( hSYS != INVALID_HANDLE_VALUE )
    {
        // блокируем драйвер
        DeviceIoControl ( hSYS, IOCTL_WINIO_DISABLEDIRECTIO, NULL,
                        0, NULL, 0, &dwReturn, NULL );
        CloseHandle ( hSYS ); // закрываем драйвер
    }
    // освобождаем системные ресурсы
    _freeService ( );
    hSYS = NULL;
}
// функции
bool CIO32NT :: InitPort ( )
{
    bool bResult;
    PSTR pszTemp;
    char szExe[MAX_PATH];
    DWORD dwRet;
    // открываем драйвер
    hSYS = CreateFile ( "\\.\.\.\.\IOTrserv", GENERIC_READ | GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL );
    // если не удалось, инициализируем службу сервисов
    if ( hSYS == INVALID_HANDLE_VALUE )
    {
        // получаем имя программы
        if ( !GetModuleFileName ( GetModuleHandle ( NULL ), szExe,
                                sizeof ( szExe ) ) )
            return false;
        // ищем указатель на последнюю косую черту
        pszTemp = strrchr ( szExe, '\\' );
        // убираем имя программы
        pszTemp[1] = 0;
    }
}
```