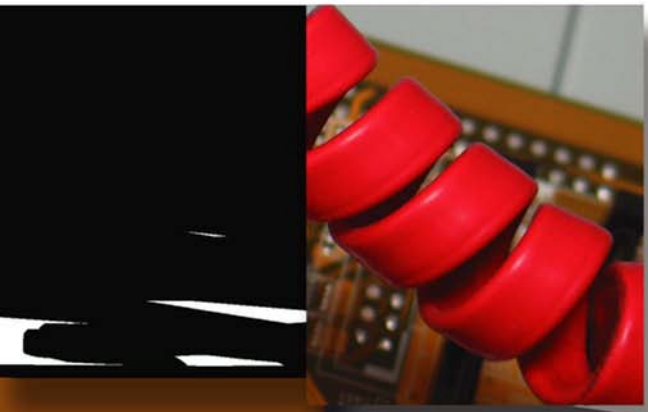
ВСЕВОЛОД НЕСВИЖСКИЙ

bhv®

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

Всеволод Несвижский

ПРОГРАММИРОВАНИЕ АППАРАТНЫХ СРЕДСТВ В WINDOWS

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2008

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

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 22.07.08.
Формат $70\times100^{1}/_{16}$. 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

В современном мире, где, как грибы после дождя, появляются все новые и новые языки программирования, многие люди наивно причисляют себя к программистам, владея одним или двумя популярными скриптовыми имитаторами. К ним я отношу, в первую очередь, такие, как Visual Basic, C# и им подобные. Ни в коем случае не принижая высокий уровень интеграции и удобства данных языков, при всем желании не могу их поставить в один ряд с C, C++ и, тем более, с ассемблером. И если в первом случае простота и легкость написания кода приводят к стандартным программам (клонам) и "раздутым" дистрибутивам, то во втором — все зависит от фантазии и мастерства программиста. Конечно, мне могут возразить, что сегодня решающим фактором является время, а не дисковое пространство, но и это составляет всего лишь часть всей правды. Гораздо более важным моментом следует считать надежность и переносимость программного обеспечения. Хорошо конечно, что есть такая операционная система, как Windows, под которую писать программы легко и удобно. А если ее не станет или она полностью изменит свою структуру, а фирма Microsoft очередной раз "порекомендует" изучить новый скриптовый язык? Разработчикам ничего не останется, как последовать рекомендациям или же, наконец, перейти на полноценный язык программирования. Одним словом, можно сколько угодно гадать о будущем, но оно от этого ничуть не изменится, поэтому изучайте C или C++, не думая о завтрашнем дне.

Книга, которую вы держите в руках, ориентирована на тех, кто любит и ценит C++, кто, несмотря на все заманчивые "предложения" ведущих поставщиков программных средств разработки, выбирают гибкость, мощь и безграничность полета фантазии. Весь представленный материал логически разделен на две части: программирование аппаратного обеспечения и общее программирование в операционных системах Windows. При этом учитываются и самые новые версии Windows, и несправедливо устаревшие.

Книга посвящена программированию базовых компонентов любого персонального компьютера: мыши, клавиатуры, процессора, системных устройств, дисковой подсистемы, мониторинга питания и температур, видео и звука. Кроме того, уделено внимание популярным сегодня интерфейсам, таким как USB, IEEE 1394 и др. Рассматриваются базовые методы программирования данных устройств — посредством прямого доступа через порты ввода-вывода. Все примеры представлены на языке C++.

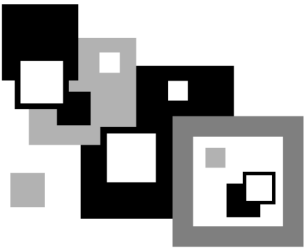
Программные требования

Для написания и отладки примеров были использованы оболочки Visual C++ 6.0 и Visual Studio 2008. Программисты, работающие в Visual C++ .NET, также могут без проблем выполнять представленные в книге примеры.

Тестирование проводилось в операционных системах Windows ME, Windows 2000, Windows XP и Windows Vista. Возникающие проблемы из-за различий в версиях учтены и выделены при рассмотрении материала.

Поддержка

В книге приведено большое количество исходных кодов, размеры которых иногда превышают разумно допустимые для ручного ввода, поэтому читатели могут скопировать их с прилагаемого к книге диска. Кроме того, для программирования оборудования рекомендуется использовать драйверы, разработанные автором специально для читателей книги. Хочу подчеркнуть, что драйверы представлены исключительно для выполнения примеров из книги и не должны применяться как-либо иначе. Возникающие вопросы и замечания направляйте, пожалуйста, на e-mail: **komarovka@rambler.ru**.



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 65 535, 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 — A,
// 2 — B, 3 — C, 4 — D, 4 — E 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 `CIO32`.

Листинг 1.7. Пример использования класса `CIO32`

```
// объявляем класс
CIO32 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 `CIO32NT`, 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-

рационных 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;
```

```
      // а вместо него добавляем имя драйвера
      strcat ( szExe, "IOtrserv.sys" );
      // загружаем сервис
      bResult = _loadService ( szExe );
      // если ошибка, выходим из функции
      if ( !bResult ) return false;
      // запускаем наш сервис
      bResult = _goService ( );
      // если ошибка, выходим из функции
      if ( !bResult ) return false;
      // открываем драйвер
      hSYS = CreateFile ( "\\\\.\\IOtrserv", GENERIC_READ |
    GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL );
       // если не удалось, выходим из функции
       if ( hSYS == INVALID_HANDLE_VALUE ) return false;
    }

    if ( !DeviceIoControl ( hSYS, IOCTL_WINIO_ENABLEDIRECTIO, NULL,
                            0, NULL, 0, &dwRet, NULL ) )
         return false; // драйвер недоступен
    return true;
}
bool  CIO32NT :: _loadService ( PSTR pszDriver )
{
  SC_HANDLE hSrv;
  SC_HANDLE hMan;
  // на всякий случай выгружаем открытый сервис
  _freeService ( );
  // открываем менеджер сервисов
  hMan = OpenSCManager ( NULL, NULL, SC_MANAGER_ALL_ACCESS );
  // создаем объект сервиса из нашего драйвера
  if ( hMan )
  {
   hSrv = CreateService ( hMan, "IOtrserv", "IOtrserv",
   SERVICE_ALL_ACCESS, SERVICE_KERNEL_DRIVER, SERVICE_DEMAND_START,
   SERVICE_ERROR_NORMAL, pszDriver, NULL, NULL, NULL, NULL, NULL );
   // освобождаем менеджер объектов
   CloseServiceHandle ( hMan );
   if ( hSrv == NULL ) return false;
  }
  else
   return false;
  CloseServiceHandle ( hMan );
```

```
  return true;
}
bool  CIO32NT :: _goService ( )
{
   bool bRes;
   SC_HANDLE hSrv;
   SC_HANDLE hMan;
   // открываем менеджер сервисов
   hMan = OpenSCManager ( NULL, NULL, SC_MANAGER_ALL_ACCESS );
   if ( hMan )
   {
     // открываем сервис
     hSrv = OpenService ( hMan, "IOtrserv", SERVICE_ALL_ACCESS );
     // закрываем менеджер сервисов
     CloseServiceHandle ( hMan );
     if ( hSrv )
     {
      // запускаем сервис
      bRes = StartService ( hSrv, 0, NULL );
      // в случае ошибки закрываем дескриптор
      if( !bRes )
           CloseServiceHandle ( hSrv );
     }
     else
      return false;
   }
   else
    return false;
   return bRes;
}
bool CIO32NT :: _stopService ( )
{
   bool bRes;
   SERVICE_STATUS srvStatus;
   SC_HANDLE hMan;
   SC_HANDLE hSrv;
   // открываем менеджер сервисов
   hMan = OpenSCManager ( NULL, NULL, SC_MANAGER_ALL_ACCESS );
   if ( hMan )
   {
     // открываем сервис
     hSrv = OpenService ( hMan, "IOtrserv", SERVICE_ALL_ACCESS );
```

```
      // закрываем менеджер сервисов
      CloseServiceHandle ( hMan );
      if ( hSrv )
      {
       // останавливаем сервис
       bRes = ControlService ( hSrv, SERVICE_CONTROL_STOP, &srvStatus );
       // закрываем сервис
       CloseServiceHandle ( hSrv );
      }
      else
       return false;
   }
   else
    return false;
   return bRes;
}
bool CIO32NT :: _freeService ( )
{
   bool bRes;
   SC_HANDLE hSrv;
   SC_HANDLE hMan;
   // останавливаем наш сервис
   _stopService ( );
   // открываем менеджер сервисов
   hMan = OpenSCManager ( NULL, NULL, SC_MANAGER_ALL_ACCESS );
   if ( hMan )
   {
     // открываем сервис
     hSrv = OpenService ( hMan, "IOtrserv", SERVICE_ALL_ACCESS );
     // закрываем менеджер сервисов
     CloseServiceHandle ( hMan );
     if ( hSrv )
     {
       // удаляем наш сервис из системы и освобождаем ресурсы
       bRes = DeleteService ( hSrv );
       // закрываем дескриптор нашего сервиса
       CloseServiceHandle ( hSrv );
     }
     else
       return false;
   }
   else
    return false;
```

```
   return bRes;
}
// пишем функции ввода-вывода
void CIO32NT :: inPort ( WORD wPort, PDWORD pdwValue, BYTE bSize )
{
   switch ( bSize )
   {
   case 1:
      *pdwValue = _inp( wPort );
       break;
   case 2:
      *pdwValue = _inpw ( wPort );
       break;
   case 4:
       *pdwValue = _inpd ( wPort );
       break;
   }
}
void CIO32NT :: outPort ( WORD wPort, DWORD dwValue, BYTE bSize )
{
   switch ( bSize )
   {
    case 1:
        _outp ( wPort, ( BYTE ) dwValue );
        break;
    case 2:
        _outpw ( wPort, ( WORD ) dwValue );
        break;
    case 4:
        _outpd ( wPort, dwValue );
        break;
   }
}
```

Второй класс получился более громоздким за счет особых требований к работе драйверов в профессиональных системах. Пришлось использовать функции менеджера служб (SCM — Service Control Manager) для регистрации нашего драйвера в качестве сервиса. Только при таком условии система позволяет получить доступ к аппаратуре. Пример работы с классом `CIO32NT` приводить не буду, поскольку он ничем не отличается от предыдущего класса.

1.4. Использование ассемблера

Четвертый вариант работы с портами заключается в применении встроенного в Visual C++ ассемблера. Как известно, ассемблер содержит две команды для доступа к портам ввода-вывода: `in` и `out`. Однако далеко не в каждой операционной системе удастся воспользоваться этим способом (командами ввода-вывода), поэтому данный вариант рекомендую использовать только в Windows 95/98/ME. Для наглядности рассмотрим пример работы со встроенным ассемблером кода, показанного в листинге 1.10.

Листинг 1.10. Использование встроенного ассемблера в Visual C++

```
// простой пример функции для управления системным динамиком
void PC_dinamik ( bool bOn )
{
   switch ( bOn )
   {
   case true:
     __asm
     {
       in al,61h
       or al, 00000011b // включить динамик
       out 61h, al
     }
     break;
   case false:
     __asm
     {
       in al, 61h
       and al, 11111100b // отключить динамик
       out 61h, al
     }
     break;
   }
}
```

Встроенный макроассемблер практически ничем не отличается от полноценного ассемблера. Имеются некоторые ограничения, но, в общем, его с успехом можно применить в собственной программе. Сразу хочу заметить, что вызывать прерывания (для упрощения работы) не следует. Это в лучшем случае приведет к фатальной ошибке в программе, а в худшем "подвесит" всю систему. Кроме того, в данной книге не рассматриваются примеры работы с использованием ассемблера.

Вот мы и рассмотрели несколько документированных способов работы с оборудованием в операционных системах Windows. А теперь поговорим о том, есть ли какая-нибудь альтернатива, которая позволит программировать порты ввода-вывода без драйверов и различных ограничений.

1.5. Недокументированный доступ к портам

В многозадачной системе Windows для гарантированной устойчивой (относительно конечно) работы пришлось использовать так называемый *защищенный режим*, в котором эффективно разделяются различные выполняемые задачи вместе с используемыми данными. Для выполнения этой задачи потребовалось гораздо больше места под описание адресов, указываемых через сегментные регистры процессора, чем они физически могли предоставить. Было решено в сегментные регистры вместо реальных адресов загружать так называемые селекторы. *Селектор* представляет собой указатель на 8-байтный блок памяти, который содержит всю необходимую информацию о сегменте. Все эти блоки собраны в *таблицы глобальных* (GDT — Global Descriptor Table) и *локальных* (LDT — Global Descriptor Table) *дескрипторов*. Кроме того, существует и *таблица дескрипторов прерываний* (IDT — Interrupt Descriptor Table). Селектор состоит из номера дескриптора (адреса) в таблице (биты 3—15), типа таблицы (1 — LDT, 0 — GDT) в бите 2 и уровня привилегий в битах 0—1 (от 0 до 3). Уровень привилегий определяет статус выполняемой задачи. Самая высокая степень привилегий (00b) позволяет программе работать на уровне ядра. Второй уровень (01b) дает полный доступ к аппаратуре. Третий (10b) и четвертый (11b) управляют различными прикладными программами и расширениями. Не буду вдаваться в тонкости, но для доступа к портам ввода-вывода нам необходимо попасть на самый высокий уровень (нулевой), для чего необходимо методом перебора получить свободный дескриптор.

Здесь мы разберем использование наивысшего уровня привилегий только для операционных систем Windows 95/98/ME. Говорят, есть аналогичный вариант и для профессиональных систем, но мне он неизвестен. Итак, напишем еще один класс для прямого доступа к портам ввода-вывода. В листингах 1.11 и 1.12 представлены соответствующие файлы класса `IO32_0`, который нам позволит без проблем работать напрямую с любым оборудованием в операционных системах Windows 95/98/ME.

Листинг 1.11. Файл IO32_0.h

```
// объявляем класс
class IO32_0
```

```
{
public:
       IO32_0 ( ); // конструктор
       ~IO32_0 ( ) { } // пустой деструктор
// общие функции
       // прочитать значение из порта
       bool inPort ( WORD wPort, PDWORD pdwValue, BYTE bSize );
       // записать значение в порт
       bool outPort( WORD wPort, DWORD dwValue, BYTE bSize );
private:
#pragma pack ( 1 )
// объявляем структуры
// структура для поиска дескриптора в таблице
typedef struct _GDT
{
   WORD Limit; // лимит
   DWORD Base; // база
} GDT;
// описание дескриптора для сегмента данных
typedef struct _GDT_HANDLE
{
   WORD L_0_15; // биты 0-15 лимита
   WORD B_0_15; // биты 0-15 базы сегмента
   BYTE B_16_23; // биты 16-23 базы сегмента
   BYTE Access : 4; // доступ к сегменту
   BYTE Type : 1; // тип сегмента ( 1 — код, 0 — данные )
   BYTE DPL : 2; // уровень привилегий для дескриптора сегмента
   BYTE IsRead : 1; // проверка наличия сегмента
   BYTE L_16_19 : 4; // биты 16-19 лимита
   BYTE OS : 1; // определяется операционной системой
   BYTE RSV_NULL : 1; // резерв
   BYTE B_32is16 : 1; // разрядность ( 1 — 32-разрядный сегмент, 0 — 16 )
   BYTE L_Granul : 1; // гранулярность ( 1 — 4 Кб, 0 — в байтах )
   BYTE B_24_31; // биты 24-31 базы сегмента
} GDT_HANDLE;
// описание дескриптора шлюза
typedef struct _Sluice_Handle
{
   WORD Task_Offset_0_15; // младшее слово смещения для шлюза задачи
   WORD Segment_S; // селектор сегмента
   WORD DWORD_Count : 5; // число двойных слов для работы стека
   WORD NULL_5_7 : 3; // равно 0
   WORD Access : 4; // доступ к сегменту
```