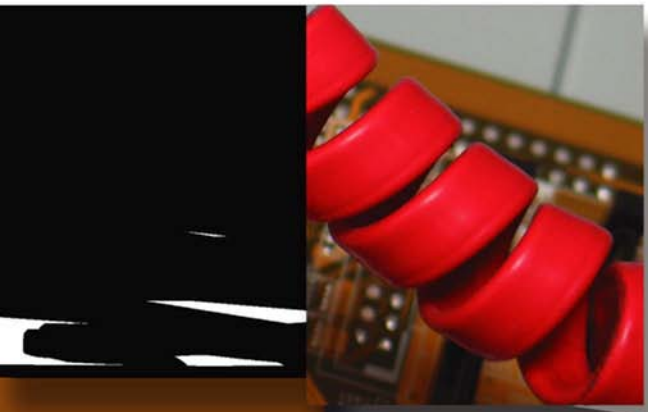
ВСЕВОЛОД НЕСВИЖСКИЙ

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 70×100 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

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

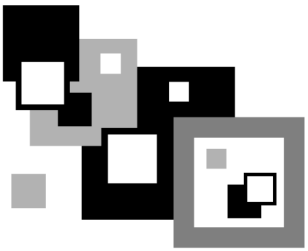
Для написания и отладки примеров были использованы оболочки Visual C++ 6.0 и Visual Studio 2008. Программисты, работающие в Visual C++ .NET, также могут без проблем выполнять представленные в книге примеры.

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 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 — 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
```

```
   WORD Type : 1; // тип шлюза
   WORD DPL : 2; // уровень привилегий для дескриптора шлюза
   WORD IsRead : 1; // проверка наличия сегмента
   WORD Task_Offse_16_31; // старшее слово смещения для шлюза задачи
} Sluice_Handle;
#pragma pack ( )
   // функция доступа к портам
   bool CallAccess ( PVOID FunctionAddress, WORD wPort, PDWORD pdwValue,
                     BYTE bSize );
}; // окончание класса
```

Листинг 1.12. Файл IO32_0.cpp

```
#include "stdafx.h"
#include "IO32_0.h"
// реализация класса
IO32_0 :: IO32_0 ( )
{
   // пустой конструктор
}
// две функции для обработки портов ввода-вывода
__declspec ( naked ) void _outPortValue ( )
{
   _asm
   {
     cmp cl, 1 // если размер данных равен байту,
     je _Byte // записываем байт
    cmp cl, 2 // если размер данных равен слову,
    je _Word // записываем слово
    cmp cl, 4 // если размер данных равен двойному слову,
    je _Dword // записываем двойное слово
_Byte:
    mov al, [ebx]
    out dx, al // записываем байт в порт
    retf // выходим
_Word:
    mov ax, [ebx]
    out dx, ax // записываем слово в порт
    retf // выходим

_Dword:
    mov eax, [ebx]
```

```
    out dx, eax // записываем двойное слово в порт
    retf // выходим
  }
}
__declspec ( naked ) void _inPortValue ( )
{
  _asm
  {
    cmp cl, 1 // если размер данных равен байту,
    je _Byte // читаем байт
    cmp cl, 2 // если размер данных равен слову,
    je _Word // читаем слово
    cmp cl, 4 // если размер данных равен двойному слову,
    je _Dword // читаем двойное слово
_Byte:
    in al, dx // читаем байт из порта
    mov [ebx], al
    retf
_Word:
    in ax, dx // читаем слово из порта
    mov [ebx], ax
    retf

_Dword:
    in eax, dx // читаем двойное слово из порта
    mov [ebx], eax
    retf
  }
}
// функция поиска свободного дескриптора в системе
bool IO32_0 :: CallAccess ( PVOID FunctionAddress, WORD wPort,
                            PDWORD pdwValue, BYTE bSize )
{
   WORD wNum = 1; // счетчик циклов
   WORD Addr[3]; // адрес для нашей задачи
   GDT gdt;
   GDT_HANDLE *pHANDLE;
   // получаем регистр GDTR начиная с 286 процессора
   _asm sgdt [gdt]
   // переходим ко второму дескриптору сегмента данных
   pHANDLE = ( GDT_HANDLE* ) ( gdt.Base + 8 );
   // выполняем поиск свободного дескриптора в таблице GDT
   for ( wNum = 1; wNum < ( gdt.Limit / 8 ); wNum++ )
```

```
   {
      // если указанные поля структуры равны 0,
      // свободный дескриптор найден
      if ( pHANDLE->IsRead == 0 && pHANDLE->DPL == 0 &&
           pHANDLE->Type == 0 && pHANDLE->Access  == 0 )
      {
        // определяем параметры для дескриптора шлюза
        Sluice_Handle *pSluice;
        pSluice = ( Sluice_Handle* ) pHANDLE;
        // младшее слово адреса нашей функции
        pSluice->Task_Offset_0_15 = LOWORD ( FunctionAddress );
        // селектор сегмента с наивысшим уровнем привилегий
        pSluice->Segment_S = 0x28;
        pSluice->DWORD_Count = 0;
        pSluice->NULL_5_7 = 0;
        pSluice->Access = 0x0C;
        pSluice->Type = 0;
        pSluice->DPL = 3;
        pSluice->IsRead = 1;
        // старшее слово адреса нашей функции
        pSluice->Task_Offse_16_31 = HIWORD ( FunctionAddress );
        // заполняем адрес для дальнего вызова
        Addr[0] = 0x00;
        Addr[1] = 0x00;
        Addr[2] = ( wNum << 3 ) | 3;
        // передаем наши параметры
        _asm
        {
          mov ebx, [pdwValue] // передаваемое значение
          mov cl, [bSize] // размер передаваемого значения
          mov dx, [wPort] // номер порта ввода-вывода
          // выполняем дальний вызов
          call fword ptr [Addr]
        }
        // обнуляем дескриптор
        memset ( pHANDLE, 0, 8 );
        return true; // выходим из функции
      }
   // проверяем следующий дескриптор
   pHANDLE++;
   }
   // не найдено ни одного свободного дескриптора
   return false;
}
```

```
// общедоступные функции ввода-вывода
bool IO32_0 :: outPort ( WORD wPort, DWORD dwValue, BYTE bSize )
{
   return CallAccess ( ( PVOID ) _outPortValue, wPort, &dwValue, bSize );
}
bool IO32_0 :: inPort ( WORD wPort, PDWORD pdwValue, BYTE bSize )
{
   return CallAccess ( ( PVOID ) _inPortValue, wPort, pdwValue, bSize );
}
```

Вот у нас и получился полноценный класс для работы с аппаратными портами. Используется он таким же образом, как и предыдущие классы. Принцип работы класса `IO32_0` построен по следующему алгоритму: вначале выполняем поиск свободного дескриптора в таблице глобальных дескрипторов (GDT). Для получения начального указателя на таблицу применяется системная команда `SGDT`. После этого производится поиск свободного дескриптора (не используемого системой). Если дескриптор найден, то присваиваем ему указатель на заполненный дескриптор шлюза (`Sluice_Handle`). Через структуру шлюза определяем адрес функции, на которую будет передано управление посредством дальнего вызова, а также общие параметры доступа и уровень привилегий. Далее записываем аргументы для функции и выполняем непосредственно дальний вызов, который будет обработан процессором с максимальным приоритетом. Таким образом, мы получим доступ к аппаратным портам, и операционная система не будет мешать этому.

Вот и все, что мне хотелось рассказать о методах работы с портами ввода-вывода в Windows. Используйте наиболее удобный для вас вариант, чтобы без проблем работать с примерами из последующих глав книги.

1.6. Определение параметров оборудования

Не секрет, что для работы с каким-либо оборудованием, необходимо определить, где оно находится, другими словами, определить выделенные системой порты, адреса в памяти, номера прерываний. Конечно, многие стандартные устройства, такие как мышь, клавиатура, системный динамик, таймер по традиции имеют строго определенные параметры конфигурации. Эти параметры не зависят от версии операционной системы, и после выполнения автоматической конфигурации оборудования всегда одинаковы. Однако этого нельзя сказать про новые устройства и интерфейсы, которые появились относитель-

но недавно (за последние 10 лет) и продолжают активно разрастаться и дополняться. В связи с этим, операционная система автоматически распределяет необходимые ресурсы для работы с всевозможным оборудованием и нет никакой гарантии того, что, например, номера портов для доступа к одному и тому же устройству будут идентичны на двух различных компьютерах. Для решения этой проблемы необходимо всегда перед обращением к установленным в системе устройствам определять их конфигурационные параметры. Замечу, что при рассмотрении современных интерфейсов (в частности, USB) я не буду приводить фиксированные адреса портов ввода-вывода, а буду лишь указывать их смещения относительно базового адреса. Поэтому, воспользовавшись информацией, представленной в этом разделе, вы сможете самостоятельно получить необходимые данные для доступа к оборудованию непосредственно в вашей системе.

Итак, существует простая возможность, позволяющая достаточно легко получить конфигурационные параметры установленного в системе оборудования. К таким параметрам относятся в первую очередь номера портов ввода-вывода, выделенные прерывания и адреса в памяти. Все что нам потребуется — это написать несколько функций для доступа к системному реестру. Именно там, в удобной, но несколько запутанной форме, расположены все основные сведения о компьютере. Увидеть эти данные можно как с помощью диспетчера оборудования, так и при непосредственном просмотре файла реестра.

Сразу замечу, что размещение данных (пути) в реестре, описывающих имеющиеся устройства, отличается для профессиональных систем (Windows NT/2000/XP/2003/Vista). Кроме того, доступ к реестру организовать несколько сложнее по сравнению с пользовательскими системами Windows 9X/ME.

Данные о конфигурации оборудования находятся в следующих разделах реестра: "HKEY_DYN_DATA\Config Manager\Enum" для Windows 95/98/ME и "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum" для Windows NT/2000/XP/2003/Vista. В подразделе "Enum" перечислены все доступные системе устройства и их основные параметры. В первую очередь, нас интересуют расположение устройства (параметр "HardWareKey") и выделенные системой ресурсы (параметр "Allocation"). С помощью значения "HardWareKey" мы определим имя и класс устройства, а двоичный массив "Allocation" поможет вычислить порты ввода-вывода, адреса памяти, каналы IRQ и DMA. Остальные параметры в данном случае не имеют значения. Итак, напишем функцию, которая просканирует реестр в поисках информации об установленном оборудовании для Windows 9X/ME. Пример такой функции показан в листинге 1.13.

Листинг 1.13. Определение основных параметров устройств в Windows 9X/ME

```
#include "stdafx.h"
// определяем макросы
// раздел реестра HKEY_DYN_DATA
#define HKDD_ENUM   "Config Manager\\Enum"
#define HKLM_ENUM   "Enum\\" // раздел реестра HKEY_LOCAL_MACHINE
#define MAX_DEVICES 200 // возможное количество устройств в системе
// перечисляем основные типы данных
enum DataTypes9x
{
   Memory_Type_9x = 0x00000001, // адрес в памяти
   Port_Type_9x = 0x00000002, // адрес порта ввода-вывода
   DMA_Type_9x = 0x00000003, // номер канала DMA
   IRQ_Type_9x = 0x00000004 // номер прерывания
};
// определяем структуры для описания данных
typedef struct _Memory9x // обработка адресов
{
   DWORD cbSize; // размер данных
   DWORD dwType; // тип данных
   DWORD Reserved; // не используется
   long int Base_Memory; // базовый адрес
   long int End_Memory; // завершающий адрес
   DWORD Reserved2; // не используется
   long int Align_Memory; // выравнивание данных
   long int CountBytes; // количество выделенных байтов в памяти
   long int MinAddrres; // минимальное значение адреса
   long int MaxAddrres; // максимальное значение адреса
   DWORD Reserved3; // не используется
} MEMORY9X, *PMEMORY9X;
// структура для хранения портов ввода-вывода
typedef struct _Port9x
{
   DWORD cbSize; // размер данных
   DWORD dwType; // тип данных
   DWORD Reserved; // не используется
   WORD Base_Port; // адрес базового порта
   WORD End_Port; // адрес последнего порта
   DWORD Reserved2; // не используется
   WORD IOR_Align; // выравнивание данных
   WORD CountPorts; // количество портов
   WORD IOR_Min; // минимальный адрес порта
```

```
   WORD IOR_Max; // максимальный адрес порта
   DWORD Reserved3; // не используется
} PORT9X, *PPORT9X;
// структура для хранения канала DMA
typedef struct _DMA9X
{
   DWORD cbSize; // размер данных
   DWORD dwType; // тип данных
   BYTE Reserved; // не используется
   BYTE Channel; // номер канала DMA
   WORD Reserved2; // не используется
} DMA9X, *PDMA9X;
// структура для хранения номера прерывания
typedef struct _IRQ9X
{
   DWORD cbSize; // размер данных
   DWORD dwType; // тип данных
   WORD Reserved; // не используется
   WORD Number; // номер прерывания
   DWORD Reserved2; // не используется
} IRQ9X, *PIRQ9X;
// структура для хранения имени и класса устройства
typedef struct _DeviceParams9x
{
   char HardWareKey[MAX_PATH]; // путь к устройству в реестре
   char DeviceDesc[150]; // название устройства
   char Class[50]; // класса устройства
   DWORD Base_Memory[16]; // базовые адреса в памяти
   DWORD End_Memory[16]; // завершающие адреса в памяти
   WORD Base_Port[16]; // адреса базовых портов ввода-вывода
   WORD End_Port[16]; // номера завершающих портов ввода-вывода
   BYTE DMA_Channel[8]; // номера выделенных каналов DMA
   BYTE IRQ_Number[16]; // номера выделенных прерываний
} DEVICEPARAMS9X, *PDEVICEPARAMS9X;
// определяем функцию поиска устройств в реестре
void GetDevicesConfig_9X ( );
// массив структур DEVICEPARAMS9X для описания устройств
DEVICEPARAMS9X devices[MAX_DEVICES];
// реализация функции
void GetDevicesConfig_9X ( )
{
   HKEY phKey = NULL, phKey2 = NULL; // дескрипторы разделов реестра
   DWORD dwKey = 0; // счетчик разделов реестра
   DWORD dwSize = 0; // размер значения параметра реестра
```

```
DWORD dwTypeParametr = REG_SZ; // тип параметра реестра
char section_name[MAX_PATH]; // имя найденного раздела
char new_path[MAX_PATH]; // полный путь к разделу
char hard_key[150]; // описание параметра
int i = 0; // счетчик циклов
BYTE byBuffer[300]; // буфер для данных
// счетчики для определенных типов данных
unsigned int uMemory_index = 0, uPort_index = 0, uDMA_index = 0,
             uIRQ_index = 0;
MEMORY9X temp9x; // временный буфер для форматированных данных
// структуры для хранения определенных данных
MEMORY9X memory9x;
PORT9X port9x;
DMA9X dma9x;
IRQ9X irq9x;
// обнуляем структуры
ZeroMemory ( &memory9x, sizeof ( MEMORY9X ) );
ZeroMemory ( &port9x, sizeof ( PORT9X ) );
ZeroMemory ( &dma9x, sizeof ( DMA9X ) );
ZeroMemory ( &irq9x, sizeof ( IRQ9X ) );
// обнуляем главную структуру данных
for ( i = 0; i < MAX_DEVICES; i++ )
  ZeroMemory ( &devices[i], sizeof ( DEVICEPARAMS9X ) );
// открываем раздел реестра "HKEY_DYN_DATA\Config Manager\Enum"
if( !RegOpenKeyEx ( HKEY_DYN_DATA, HKDD_ENUM, 0, KEY_READ, &phKey )
                    == ERROR_SUCCESS )
  return; // выходим из функции в случае ошибки
while ( 1 ) // обрабатываем все найденные подразделы
{
  // получаем очередное имя подраздела
  if ( RegEnumKey ( phKey, dwKey, section_name, MAX_PATH )
                    == ERROR_NO_MORE_ITEMS )
     break; // если все подразделы найдены, завершаем цикл
  // собираем полный путь к найденному подразделу
  strcpy ( new_path, HKDD_ENUM );
  strcat ( new_path, "\\" );
  strcat ( new_path, section_name );
  // открываем подраздел
  if ( RegOpenKeyEx ( HKEY_DYN_DATA, new_path, 0, KEY_QUERY_VALUE,
                      &phKey2 )== ERROR_SUCCESS )
  {
    // предварительно определяем размер возвращаемых данных
    if ( RegQueryValueEx ( phKey2, "HardWareKey", NULL,
                 &dwTypeParametr,  NULL, &dwSize ) == ERROR_SUCCESS )
```

```
{
  // получаем путь к устройству
  RegQueryValueEx ( phKey2, "HardWareKey", NULL, NULL,
                  ( LPBYTE ) hard_key, &dwSize );
  // сохраняем полученное значение
  strcpy ( devices[dwKey].HardWareKey, hard_key );
}
// определяем размер следующего значения
if ( RegQueryValueEx ( phKey2, "Allocation", NULL,
             &dwTypeParametr,  NULL, &dwSize ) == ERROR_SUCCESS )
{
   // получаем параметры устройства
   RegQueryValueEx ( phKey2, "Allocation", NULL, NULL,
                   ( LPBYTE ) byBuffer, &dwSize );
   // восстанавливаем и форматируем данные
   int index = 8;
   ZeroMemory ( &temp9x, sizeof ( MEMORY9X ) );
   // анализируем данные и выделяем нужное
   while ( byBuffer[index] > 0 )
   {
     // помещаем данные в структуру
     memmove ( ( void* ) &temp9x, &byBuffer[index], 8 );
     // выделяем тип данных
     switch ( temp9x.dwType )
     {
      case Memory_Type_9x: // адреса в памяти
        if  ( uMemory_index < 16 )
        {
          // помещаем данные о памяти в структуру
          memmove ( ( void* ) &memory9x, &byBuffer[index],
                      sizeof ( MEMORY9X ) );
          // сохраняем базовый адрес
          devices[dwKey].Base_Memory[uMemory_index] =
                                          memory9x.Base_Memory;
          // сохраняем завершающий адрес
          devices[dwKey].End_Memory[uMemory_index] =
                                          memory9x.End_Memory;
          uMemory_index++; // увеличиваем счетчик
          // обнуляем структуру
          ZeroMemory ( &memory9x, sizeof ( MEMORY9X ) );
         }
         break;
```

```
          case Port_Type_9x: // адреса портов ввода-вывода
            {
              memmove ( ( void* ) &port9x, &byBuffer[index],
                          sizeof ( PORT9X ) );
              devices[dwKey].Base_Port[uPort_index] =
                                              port9x.Base_Port;
              devices[dwKey].End_Port[uPort_index] = port9x.End_Port;
              uPort_index++;
              ZeroMemory ( &port9x, sizeof ( PORT9X ) );
            }
            break;
          case DMA_Type_9x: // каналы прямого доступа к памяти
            {
              memmove ( ( void* )&dma9x, &byBuffer[index],
                          sizeof ( DMA9X ) );
              devices[dwKey].DMA_Channel[uDMA_index] = dma9x.Channel;
              uDMA_index++;
              ZeroMemory ( &dma9x, sizeof ( DMA9X ) );
            }
            break;
          case IRQ_Type_9x: // номер прерывания
            {
              memmove ( ( void* ) &irq9x, &byBuffer[index],
                          sizeof ( IRQ9X ) );
              devices[dwKey].IRQ_Number[uIRQ_index] = irq9x.Number;
              uIRQ_index++;
              ZeroMemory ( &irq9x, sizeof ( IRQ9X ) );
            }
            break;
        }
      // увеличиваем смещение в буфере данных
      index = index + temp9x.cbSize;
      }
  }
  // закрываем подраздел
  if ( phKey2 ) RegCloseKey ( phKey2 );
}
else
{
  // если не удалось открыть подраздел, переходим к следующему
  continue;
}
// увеличиваем счетчик разделов
dwKey++;
```

```
  // обнуляем счетчики
  uMemory_index = 0;
  uPort_index = 0;
  uDMA_index = 0;
  uIRQ_index = 0;
}
// закрываем корневой раздел
if ( phKey ) RegCloseKey ( phKey );
phKey = NULL;
i = 0;
// получаем описание найденного устройства
for ( i; i < dwKey; i++ )
{
  // собираем путь к устройству
  strcpy ( new_path, HKLM_ENUM );
  strcat ( new_path, devices[i].HardWareKey );
  // открываем подраздел
  if ( RegOpenKeyEx ( HKEY_LOCAL_MACHINE, new_path, 0,
                      KEY_QUERY_VALUE, &phKey ) == ERROR_SUCCESS )
  {
    // определяем размер значения параметра
    if ( RegQueryValueEx ( phKey, "DeviceDesc", NULL, &dwTypeParametr,
                           NULL, &dwSize ) == ERROR_SUCCESS )
    {
      // получаем значение параметра
      RegQueryValueEx ( phKey, "DeviceDesc", NULL, NULL,
                        ( LPBYTE ) hard_key, &dwSize );
      // сохраняем в основной структуре
      strcpy ( devices[i].DeviceDesc, hard_key );
    }
    // закрываем подраздел
    if ( phKey ) RegCloseKey ( phKey );
  }
}
// определяем имя класса для найденных устройств
for ( i = 0; i < dwKey; i++ )
{
  // собираем путь к устройству
  strcpy ( new_path, HKLM_ENUM );
  strcat ( new_path, devices[i].HardWareKey );
  // открываем подраздел
  if (RegOpenKeyEx ( HKEY_LOCAL_MACHINE, new_path, 0, KEY_QUERY_VALUE,
                     &phKey ) == ERROR_SUCCESS )
```

```
    {
        // определяем размер значения параметра
        if ( RegQueryValueEx ( phKey, "Class", NULL, &dwTypeParametr,
                               NULL, &dwSize ) == ERROR_SUCCESS )
       {
         RegQueryValueEx ( phKey, "Class", NULL, NULL,
                         ( LPBYTE ) hard_key, &dwSize );
         strcpy ( devices[i].Class, hard_key );
       }
       if ( phKey ) RegCloseKey ( phKey );
    }
  }
}
```

Для обработки данных были созданы пять специальных структур, четыре из которых хранят определенный тип данных, а последняя связывает все полученные данные в общий блок описателя устройства. После этого с помощью функции `RegOpenKeyEx` открыли раздел реестра "HKEY_DYN_DATA\Config Manager\Enum". Данный раздел является динамическим, т. е. создается заново при каждом запуске операционной системы, поэтому не стоит полагаться на случай и сохранять текущую конфигурацию оборудования в файл или куда-то еще. Правильнее будет перед обращением к оборудованию повторно вызывать функцию `GetDevicesConfig_9X`. После успешного открытия раздела посредством функции `RegEnumKey` получаем все имена подразделов. В качестве счетчика эта функция использует переменную `dwKey`, возвращая при каждом обращении новое значение. Когда все имена подразделов переданы, функция завершается с кодом `ERROR_NO_MORE_ITEMS`. В открытом подразделе получаем значение параметра "HardWareKey" и сохраняем его в структуру `DEVICEPARAMS`. Оно содержит определенный путь в реестре и понадобится нам позже для определения класса и типа устройства. Далее с помощью функции `RegQueryValueEx` (для получения сразу нескольких значений рекомендуется использовать функцию `RegQueryMultipleValues`) считываем значение двоичного параметра "Allocation". Этот параметр позволяет получить основные характеристики устройства: адреса доступа в памяти и адреса ввода-вывода, номер прерывания и канал DMA. И в завершении, восстанавливаем имя класса и тип для каждого найденного устройства.

А теперь рассмотрим, как аналогичную задачу можно решить в профессиональных системах Windows NT/2000/XP/2003/Vista. Здесь, как я уже говорил, код будет несколько сложнее и запутаннее. Пример функции, определяющей доступные устройства, показан в листинге 1.14.

Листинг 1.14. Определение основных параметров устройств в Windows NT/2000/XP/2003/Vista

```
#include "stdafx.h"
// путь к разделу в реестре, описывающему установленное оборудование
#define HKLM_ENUM_NT  "SYSTEM\\CurrentControlSet\\Enum"
#define MAX_DEVICES 200 // возможное количество устройств в системе
// дополнительные флаги
#define PORT_MEMORY 0x0000 // адрес порта в памяти
#define PORT_IO 0x0001 // адрес порта ввода-вывода
#define MEMORY_READ_WRITE 0x0000 // память для записи и чтения
#define MEMORY_READ_ONLY 0x0001 // память только для чтения
#define MEMORY_WRITE_ONLY 0x0002 // память только для записи
// тип канала DMA
#define DMA_8 0x0000
#define DMA_16 0x0001
#define DMA_32 0x0002
// типы данных
enum DataTypesNT
{
   Port_Type = 1, // адрес порта ввода-вывода
   IRQ_Type = 2, // номер прерывания
   Memory_Type = 3, // адрес в памяти
   DMA_Type = 4, // номер канала DMA
   BusNumber_Type = 6 // номер шины
};
// определяем тип значения для хранения адресов
typedef LARGE_INTEGER PHYSICAL_ADDRESS;
// определяем структуры для описания данных
#pragma pack ( 4 )
typedef struct _IDDATA
{
   unsigned char Type; // тип возвращаемых данных
   unsigned char Share; // общий доступ к данным
   unsigned short Flags; // дополнительный флаг описания
   // общая область памяти
   union
   {
      struct // обработка портов ввода-вывода
      {
        PHYSICAL_ADDRESS Start; // базовый адрес
        unsigned long Length; // ширина порта
      } Port;
```

```
      struct // обработка номера прерывания
      {
        unsigned long Level; // уровень доступа
        unsigned long Number; // номер выделенного прерывания
        unsigned long Reserved; // резерв
      } IRQ;
      struct // обработка адресов в памяти
      {
        PHYSICAL_ADDRESS Start; // базовый адрес
        unsigned long Length; // ширина адресного пространства
      } Memory;
      struct
      {
        unsigned long Channel; // номер выделенного канала DMA
        unsigned long Port; // номер порта
        unsigned long Reserved; // резерв
      } DMA;
      struct // обработка номера шины
      {
        unsigned long Start; // базовый адрес шины
        unsigned long Length; // ширина адресного пространства
        unsigned long Reserved; // резерв
      } BusNumber;
   } u;
} IDDATA, *PIDDATA;
#pragma pack ( )
typedef struct _DATA_LIST
{
   unsigned short Version; // номер версии
   unsigned short Revision; // дополнительный номер версии
   unsigned long Count; // полное количество данных об устройстве
   IDDATA Id[16]; // массив структур для получения данных
} DATA_LIST, *PDATA_LIST;
typedef struct _DATA_LIST_ALL
{
   DWORD Reserved; // резерв
   unsigned long Reserved2; // резерв
   DATA_LIST DataList; // структура DATA_LIST
} DATA_LIST_ALL, *PDATA_LIST_ALL;
typedef struct _DATA
{
   unsigned long Count; // количество найденных описателей устройства
   DATA_LIST_ALL All_List[2]; // массив структур DATA_LIST_ALL
} DATA, *PDATA;
```

```
// основная структура описателя устройства
typedef struct _DeviceParamNT
{
   char DeviceDesc[100]; // текстовое описание устройства
   char FriendlyName[80]; // дополнительное описание устройства
   char Class[50]; // имя класса устройства
   PHYSICAL_ADDRESS  Base_Memory[16]; // базовые адреса в памяти
   PHYSICAL_ADDRESS  End_Memory[16]; // завершающие адреса в памяти
   unsigned short TypeBaseMemory[16]; // тип адреса в памяти
   PHYSICAL_ADDRESS  Base_Port[16]; // базовый порт ввода-вывода
   PHYSICAL_ADDRESS  End_Port[16]; // завершающий порт ввода-вывода
   unsigned short TypeBasePort[16]; // тип порта ввода-вывода
   BYTE DMA_Channel[8]; // номер выделенного канала DMA
   unsigned short TypeDMA[8]; // тип канала DMA
   BYTE IRQ_Number[16]; // номер выделенного прерывания
   unsigned int BusNumber[8]; // номер шины для некоторых типов устройств
} DEVICEPARAMSNT, *PDEVICEPARAMSNT;
// определяем функцию поиска устройств в реестре
void GetDevicesConfig_NT ( );
// массив структур DEVICEPARAMSNT для описания устройств
DEVICEPARAMSNT devices_NT[MAX_DEVICES];
// реализация функции
void GetDevicesConfig_NT ( )
{
   // объявляем переменные
   HKEY phKey = NULL, phKey2 = NULL, phKey3 = NULL, phKey4 = NULL,
                      phKey5 = NULL; // дескрипторы разделов реестра
   // счетчики разделов и размеры значений параметров в реестре
   DWORD dwKey = 0, dwKey2 = 0, dwKey3 = 0, dwSize = 0, cbName = 256;
   // счетчики данных
   unsigned int uPort_index = 0, uIRQ_index = 0, uMemory_index = 0,
                uDMA_index = 0, uBus_index = 0;
   // счетчик найденных устройств
   unsigned int uIndex = 0;
   // различные пути в реестре
   char section_name[513];
   char new_path[MAX_PATH];
   char new_path2[MAX_PATH];
   char new_path3[MAX_PATH];
   char new_path4[MAX_PATH];
   char new_path5[MAX_PATH];
   // буфер для данных
   char hard_key[150];
```

```
// структура для получения данных об устройстве
DATA cfg;
// обнуляем структуру перед использованием
for ( i = 0; i < 200; i++ )
    ZeroMemory ( &devicesNT[i], sizeof ( DEVICEPARAMSNT ) );
// открываем раздел реестра для перечисления подразделов
if ( !RegOpenKeyEx ( HKEY_LOCAL_MACHINE, HKLM_ENUM_NT, 0, KEY_READ,
                     &phKey ) == ERROR_SUCCESS )
      return;
while ( 1 )
{
 // определяем размер значения
 cbName = 256;
 // перечисляем подразделы в открытом разделе
 if ( RegEnumKeyEx ( phKey, dwKey, section_name, &cbName, NULL, NULL,
                     NULL, NULL ) == ERROR_NO_MORE_ITEMS )
    break; // выходим из цикла, если все подразделы получены
  // собираем путь для доступа к вложенному разделу
  strcpy ( new_path, "" );
  strcpy ( new_path, HKLM_ENUM_NT );
  strcat ( new_path, "\\" );
  strcat ( new_path, section_name );
  // открываем вложенный раздел
  if ( RegOpenKeyEx ( HKEY_LOCAL_MACHINE, new_path, 0, KEY_READ,
                      &phKey2 ) == ERROR_SUCCESS )
  {
    // сбрасываем счетчик
    dwKey2 = 0;
    while ( 1 )
    {
      // определяем размер значения
      cbName = 256;
      // перечисляем подразделы в открытом разделе
      if ( RegEnumKeyEx ( phKey2, dwKey2, new_path2, &cbName, NULL,
                          NULL, NULL, NULL ) == ERROR_NO_MORE_ITEMS )
           break; // выходим из цикла, если все подразделы получены
      // собираем путь для доступа к вложенному разделу
      strcpy ( new_path3, "" );
      strcpy ( new_path3, new_path );
      strcat ( new_path3, "\\" );
      strcat ( new_path3, new_path2 );
      // открываем вложенный раздел
      if ( RegOpenKeyEx ( HKEY_LOCAL_MACHINE, new_path3, 0, KEY_READ,
                          &phKey3 ) == ERROR_SUCCESS )
```

```
{
  // сбрасываем счетчик
  dwKey3 = 0;
  while ( 1 )
  {
    // определяем размер значения
    cbName = 256;
    if ( RegEnumKeyEx ( phKey3, dwKey3, new_path4, &cbName,
              NULL, NULL, NULL, NULL ) == ERROR_NO_MORE_ITEMS )
         break; // выходим из цикла
    // собираем путь для доступа к вложенному разделу
    strcpy ( new_path5, "" );
    strcpy ( new_path5, new_path3 );
    strcat ( new_path5, "\\" );
    strcat ( new_path5, new_path4 );
    // открываем вложенный раздел
    if(RegOpenKeyEx ( HKEY_LOCAL_MACHINE, new_path5, 0,
                      KEY_READ, &phKey4 ) == ERROR_SUCCESS )
    {
      dwSize = 150; // размер данных
      // получаем описание устройства
      RegQueryValueEx ( phKey4, "DeviceDesc", NULL, NULL,
                      ( LPBYTE ) hard_key, &dwSize );
      // сохраняем в основную структуру
      strcpy ( devicesNT[uIndex].DeviceDesc, hard_key );
      strcpy ( hard_key, "" );
      // получаем имя класса устройства
      dwSize = 50;
      RegQueryValueEx ( phKey4, "Class", NULL, NULL,
                      ( LPBYTE ) hard_key, &dwSize );
      strcpy ( devicesNT[uIndex].Class, hard_key );
      strcpy ( hard_key, "" );
      // получаем дополнительное описание устройства
      dwSize = 80;
      RegQueryValueEx ( phKey4, "FriendlyName", NULL, NULL,
                      ( LPBYTE ) hard_key, &dwSize );
      strcpy ( devicesNT[uIndex].FriendlyName , hard_key );
      strcpy ( hard_key, "" );
      // собираем путь для доступа
      // основным параметрам устройства
      strcat ( new_path5, "\\Control" ); // Windows 2000/XP/SR3
      // strcat ( new_path5, "\\LogConf" ); // Windows NT
```

```
// открываем раздел реестра
if ( RegOpenKeyEx ( HKEY_LOCAL_MACHINE, new_path5, 0,
                    KEY_READ, &phKey5 ) == ERROR_SUCCESS )
{
  dwSize = sizeof ( DATA ); // определяем размер данных
  // обнуляем структуру данных
  ZeroMemory ( &cfg, sizeof ( DATA ) );
  // получаем данные из реестра
  RegQueryValueEx ( phKey5, "AllocConfig", NULL, NULL,
                  ( LPBYTE ) &cfg, &dwSize );
  // определяем количество описателей устройства
  for ( int i = 0; i < cfg.Count; i++ )
  {
    for ( int j = 0; j < cfg.All_List[i].DataList.Count;
          j++ )
    {
      // определяем тип данных
      switch ( cfg.All_List[i].DataList.Id[j].Type )
      {
      case Port_Type: // порт ввода-вывода
       // получаем базовый порт
       devicesNT[uIndex].Base_Port[uPort_index] =
          cfg.All_List[i].DataList.Id[j].u.Port.Start;
       // получаем завершающий порт
       devicesNT[uIndex].End_Port[uPort_index].LowPart =
      cfg.All_List[i].DataList.Id[j].u.Port.Start.LowPart
      + cfg.All_List[i].DataList.Id[j].u.Port.Length — 1;
      // получаем тип порта ввода-вывода
      if ( cfg.All_List[i].DataList.Id[j].Flags &
           PORT_MEMORY )
        devicesNT[uIndex].TypeBasePort[uPort_index] =
           PORT_MEMORY;
      else if ( cfg.All_List[i].DataList.Id[j].Flags &
           PORT_IO )
        devicesNT[uIndex].TypeBasePort[uPort_index] =
           PORT_IO;
      // увеличиваем счетчик
      uPort_index++;
       break;
      case IRQ_Type: // номер прерывания
        devicesNT[uIndex].IRQ_Number[uIRQ_index] =
            cfg.All_List[i].DataList.Id[j].u.IRQ.Number;
        uIRQ_index++;
         break;
```

```
    case Memory_Type: // адреса в памяти
      // получаем базовый адрес
      devicesNT[uIndex].Base_Memory[uMemory_index] =
         cfg.All_List[i].DataList.Id[j].u.Memory.Start;
      // получаем завершающий адрес
     devicesNT[uIndex].End_Memory[uMemory_index].LowPart
= cfg.All_List[i].DataList.Id[j].u.Memory.Start.LowPart
+ cfg.All_List[i].DataList.Id[j].u.Memory.Length — 1;
      // определяем тип памяти
      if ( cfg.All_List[i].DataList.Id[j].Flags &
           MEMORY_READ_WRITE )
        devicesNT[uIndex].TypeBaseMemory[uPort_index] =
           MEMORY_READ_WRITE;
      else if ( cfg.All_List[i].DataList.Id[j].Flags &
           MEMORY_READ_ONLY )
        devicesNT[uIndex].TypeBaseMemory[uPort_index] =
           MEMORY_READ_ONLY;
        devicesNT[uIndex].TypeBaseMemory[uPort_index] =
           MEMORY_READ_ONLY;
        devicesNT[uIndex].TypeBaseMemory[uPort_index] =
           MEMORY_WRITE_ONLY;
       uMemory_index++;
       break;
      case DMA_Type: // номер канала DMA
       // получаем номер канала
       devicesNT[uIndex].DMA_Channel[uDMA_index] =
           cfg.All_List[i].DataList.Id[j].u.DMA.Channel;
       // определяем тип канала
       if ( cfg.All_List[i].DataList.Id[j].Flags &
            DMA_8 )
        devicesNT[uIndex].TypeDMA[uPort_index] =  DMA_8;
       else if ( cfg.All_List[i].DataList.Id[j].Flags &
            DMA_16 )
       devicesNT[uIndex].TypeDMA[uPort_index] =  DMA_16;
       else if ( cfg.All_List[i].DataList.Id[j].Flags &
            DMA_32 )
       devicesNT[uIndex].TypeDMA[uPort_index] =  DMA_32;
       uDMA_index++;
       break;
      case BusNumber_Type: // тип шины
       devicesNT[uIndex].BusNumber[uBus_index] =
       cfg.All_List[i].DataList.Id[j].u.BusNumber.Start;
```

```
                        break;
                      }
                    }
                  }
                  // закрываем раздел и обнуляем счетчики
                  if ( phKey5 ) RegCloseKey ( phKey5 );
                  uPort_index = 0;
                  uIRQ_index = 0;
                  uMemory_index = 0;
                  uDMA_index = 0;
                  uBus_index = 0;
                }
                // закрываем вложенный раздел
                if ( phKey4 ) RegCloseKey ( phKey4 );
                // увеличиваем счетчик найденных устройств
                uIndex++;
              }
              // увеличиваем счетчик для поиска следующего подраздела
              dwKey3++;
            }
            // закрываем вложенный раздел
            if ( phKey3 ) RegCloseKey ( phKey3 );
          }
          // увеличиваем счетчик для поиска следующего подраздела
          dwKey2++;
        }
        // закрываем вложенный раздел
        if ( phKey2 ) RegCloseKey ( phKey2 );
      }
     // увеличиваем счетчик для получения следующего подраздела
     dwKey++;
    }
    // закрываем корневой раздел и выходим из функции
    if ( phKey ) RegCloseKey ( phKey );
}
```

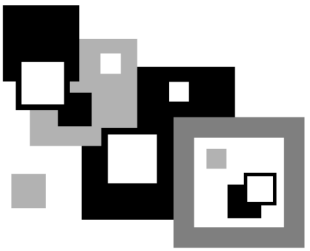
Как видите, основные принципы получения информации об установленном оборудовании в Windows NT не сильно отличаются от предыдущего примера. Изменилась только структура для хранения данных и названия разделов в реестре. Замечу только, что ключевой раздел, в котором хранятся описатели устройств, может иметь одно из двух возможных названий: "LogConf" для Windows NT 4.0 и более ранних версий или "Control" для Windows 2000/XP/2003/Vista.

1.7. Драйверы и Windows Vista

Как уже говорилось раньше, для написания драйверов в операционной системе Windows Vista используется новая драйверная модель WDF. Главной причиной создания данной модели является повышение безопасности использования всевозможного оборудования различных производителей и отказоустойчивости системы. Другими словами, фирма Microsoft, разработавшая новую модель, предприняла очередную попытку защитить в первую очередь ядро системы от несанкционированного доступа.

Поэтому теперь для написания драйвера ядра вначале следует определить платформу: 32- или 64-разрядную. Далее написать собственно драйвер. И, наконец, подписать драйвер и получить соответствующий сертификат соответствия. Без подписи драйвер не будет работать в Windows Vista. Однако есть исключения. Можно использовать драйвер, прилагаемый к данной книге, для загрузки в режиме ядра в 32-разрядной Windows Vista. В 64-битной версии этот драйвер работать не будет. Более подробную информацию о работе драйверов в Windows Vista можно получить на официальном сайте фирмы Microsoft **http://www.microsoft.com**.

ГЛАВА 2



Мышь

Было время, когда мышь была скорее экзотикой, чем необходимым компонентом компьютера. В самом деле, первые используемые операционные системы не имели графического интерфейса, и все общение с ними было построено исключительно на работе с клавиатурой. Но, как всегда, извечная человеческая лень и стремление приблизить компьютер к реальной жизни подтолкнули разработчиков к созданию графического представления компьютерных данных, отображаемых на дисплее. Операционные системы стали приобретать красивый и удобный интерфейс, в котором вся основная информация предоставлялась пользователю в графическом виде, к слову, более привычном и понятном человеческому глазу. Вот тут-то и появилась необходимость в простых устройствах, позволяющих легко управлять современными компьютерными программами. Одним из таких устройств и явилась мышь. Прошло немного времени, и она завоевала безоговорочную популярность среди обычных пользователей. Естественно, это не могло не отразиться на разработчиках программного обеспечения: им пришлось добавлять в свои программы поддержку мыши. Вот о том, как реализуется работа с мышью в современных операционных системах Windows, я и хотел бы рассказать в данной главе. Вниманию читателей предложено два способа программирования современного устройства мыши:

1. С использованием аппаратных портов.
2. С использованием возможностей стандартного интерфейса Win32 API.

Каждый из представленных выше способов имеет свои достоинства и недостатки, поэтому выбор оптимального варианта должен отвечать в первую очередь требованиям программиста по использованию мыши в своей программе. Как правило, возможностей Win32 API вполне достаточно. Если же вы пишете драйвер для мыши или хотите использовать расширенные возможности, недоступные в Win32 API, выбор очевиден — доступ через аппаратные пор-

ты. Использование портов вообще является универсальным и самым гибким способом доступа к мыши (или любому другому устройству), но для этого необходимо хорошо знать устройство мыши и иметь полную документацию на выбранный стандарт (например, PS/2 или USB). Если по устройству мыши различных стандартов информации достаточно, то найти в свободном доступе полную спецификацию для полноценного программирования очень не просто. Такая информация, как правило, предоставляется разработчиками стандартов за деньги.

На сегодняшний день существует несколько основных стандартов для интерфейса мыши: последовательный (Serial Mouse), PS/2 и USB. Последовательный интерфейс (*RS-232C*) является одним из самых старых и практически вытеснен современными интерфейсами PS/2 и USB. Использование универсальной последовательной шины (*USB* — Universal Serial Bus) для подключения мыши является скорее данью моде, а не практической целесообразностью. Она не дает важных преимуществ перед PS/2, а только занимает порт USB. Существует огромное количество других более важных устройств (принтер, сканер, веб-камера), которые можно подключать к USB-порту. Для мыши специально выделен собственный порт PS/2, и следует использовать именно его.

2.1. Общие сведения

Как уже было отмечено ранее, стандарт PS/2 использует последовательный протокол передачи данных. Если посмотреть на заднюю стенку компьютера, то можно увидеть небольшой круглый разъем, имеющий шесть контактных отверстий. Точнее говоря, их там два, для мыши и для клавиатуры, поскольку последняя использует тот же стандарт и протокол для передачи данных. Клавиатура рассматривается в следующей главе книги, а здесь только замечу, что оба устройства подключаются к контроллеру клавиатуры. Назначение контактов разъема PS/2 представлена в табл. 2.1.

Таблица 2.1. *Назначение контактов разъема PS/2*

Номер контакта	Описание
1	DATA (линия данных)
2	Не используется
3	GND 0 V (земля)
4	+5 V (питание)
5	CLK (синхронизация по времени)
6	Не используется

Принцип работы PS/2 мыши достаточно прост: данные со встроенной в мышь микросхемы поступают на контроллер клавиатуры, находящийся на материнской плате (например, интегрированный контроллер Intel 8042 или VIA 8242). На этом этапе происходит также синхронизация последовательно передаваемых данных. Контроллер клавиатуры преобразует поступающую информацию и (посредством BIOS) передает эти данные драйверу мыши. Двунаправленный последовательный обмен данных между мышью и компьютером (контроллером клавиатуры) происходит через линию данных в зависимости от сигнала синхронизации. Например, установка компьютером нулевого значения на линии синхронизации, позволяет запретить передачу данных. Устройство постоянно генерирует сигнал синхронизации. Прежде чем передать данные, компьютер устанавливает линию синхронизации в 0. Далее следует передача данных и линия данных устанавливается в 0, а линия синхронизации в 1. Данные передаются в следующем порядке:

1. Стартовый бит (всегда должен быть равен 0).
2. Восемь бит данных, начиная с младшего.
3. Следует один бит четности.
4. Стоповый бит (всегда равен 1).

Любая мышь PS/2 имеет минимум два счетчика, отслеживающих движение: по оси *X* и по оси *Y*. Значение каждого счетчика определяется 9-разрядным (старший разряд является знаковым) двоичным числом и связанным с ним флагом переполнения. Каждое значение счетчика и состояние кнопок мыши передается на компьютер в виде пакета данных. Размер одного такого пакета данных равен трем байтам. Формат пакета данных, передаваемых двухкнопочной мышью PS/2, представлен в табл. 2.2. Для трехкнопочных мышей и более пакет данных расширен до четырех байтов.

Таблица 2.2. *Формат пакета данных мыши PS/2*

Байт	Бит							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	YV	XV	YS	XS	1	0 (M)	R	L
2	X7	X6	X5	X4	X3	X2	X1	X0
3	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

Обозначения в табл. 2.2 имеют следующий смысл:

- L — состояние левой кнопки мыши (имеет значение 1, если кнопка нажата);

- R — состояние правой кнопки мыши (имеет значение 1, если кнопка нажата);
- XS и YS — знаковый разряд перемещения по осям (имеет значение 1, если установлен знак минус);
- XV и YV — состояние переполнения при перемещении по осям (имеет значение 1, если произошло переполнение);
- X0—X7 — движение по оси *X*;
- Y0—Y7 — движение по оси *Y*;
- M — состояние средней кнопки мыши. Для двухкнопочной мыши это значение равно 0.

После посылки очередного пакета счетчики движения сбрасываются в 0, а диапазон возможных значений лежит в промежутке между –255 и 255. Если диапазон превышен, устанавливается бит переполнения, и дальнейшая работа счетчиков движения возможна только после сброса. Сброс счетчиков происходит также при получении устройством мыши любой команды от компьютера, кроме `Resend` (`0xFE`). Описание команд приведено в *разд. "Использование портов" этой главы*.

Мышь PS/2 имеет два важных свойства, определяющих ее работу: частоту дискретизации и чувствительность. *Частотой дискретизации* называется число выборок в секунду, передаваемых устройством мыши на компьютер. Поддерживаются следующие значения: 200, 100, 80, 60, 40, 20 и 10 выборок за одну секунду. По умолчанию используется 100 выборок в секунду. Изменить это значение можно с помощью команды `Set Sample Rate` (`0xF3`).

Под *чувствительностью* понимается количество отсчетов, фиксируемых счетчиками движения на расстоянии одного миллиметра. Стандартом оговорены несколько уровней чувствительности: 1 отсчет, 2 отсчета, 4 отсчета и 8 отсчетов. По умолчанию используется 4 отсчета на один миллиметр. Кроме того, на чувствительность может влиять так называемый *режим масштабирования*. По умолчанию он равен 1:1, не влияя на чувствительность мыши. Можно также установить дополнительный масштаб 1:2, который существенно увеличивает чувствительность устройства. В табл. 2.3 представлены значения счетчика в режиме масштабирования 1:2. Принцип работы масштабирования устроен на дополнительном алгоритме пересчета, применяемом к счетчикам движения, перед началом передачи данных на компьютер.

Таблица 2.3. *Значения счетчика перемещения в режиме 1:2*

Значение счетчика движения	Пересчитанное значение
0	0
1	1

Таблица 2.3 *(окончание)*

Значение счетчика движения	Пересчитанное значение
2	1
3	3
4	6
5	9
N > 5	2 * N

Мышь PS/2 поддерживает четыре стандартных режима работы:

1. *Режим сброса* — данный режим может быть установлен после подачи питания или с помощью команды `Reset` (`0xFF`).
2. *Потоковый режим* — задан по умолчанию и является основным режимом работы мыши PS/2. В нем осуществляется передача данных между устройством мыши и компьютером. Если мышь находится в дистанционном режиме, то можно применить команду `Set Remote Mode` (`0xF0`) для перевода ее в потоковый режим.
3. *Дистанционный режим* — позволяет управлять передачей данных от мыши на компьютер только по запросу последнего. Компьютер может запросить тип устройства (`Get Device ID`), состояние (`Status Request`), а также данные (`Read Data`). Для ввода мыши в данный режим используется команда `Set Remote Mode` (`0xF0`).
4. *Эхо-режим* — позволяет любой посланный компьютером байт вернуть обратно. Может применяться для тестирования устройства мыши. Включить данный режим можно с помощью команды `Set Wrap Mode` (`0xEE`). Для выхода из режима следует применить команду `Reset Wrap Mode` (`0xEC`) или `Reset` (`0xFF`).

После входа в режим сброса устройство мыши выполняет самодиагностику и устанавливает следующие параметры: частота дискретизации равна 100 выборок в секунду, чувствительность равна 4 отсчета на один миллиметр, масштабирование равно 1:1, передача данных отключена. В случае успешного самотестирования мышь посылает компьютеру значение `0xAA`, а в случае ошибки — значение `0xFC`. Кроме того, после кода тестирования (`0xAA` или `0xFC`) мышь посылает идентификатор устройства, всегда равный `0x00`. В любом случае, при получении данных от мыши общий контроллер клавиатуры вырабатывает стандартное прерывание IRQ12, которое закреплено за мышью и является постоянным для большинства компьютеров.

Вот и все общие сведения о мыши стандарта PS/2, с которыми мне хотелось познакомить читателя. Детальное описание команд приводится в следующем разделе этой главы.

2.2. Использование портов

Доступ к устройству посредством аппаратных портов всегда был и останется самым мощным и универсальным. Он дает возможность непосредственно обратиться к контроллеру мыши, интегрированному на материнской плате. Самыми важными его преимуществами являются следующие:

- поддержка любых новых возможностей, появляющихся в расширениях стандарта;
- отсутствие обращений к прерываниям BIOS и DOS;
- заметное повышение скорости в приложениях реального времени.

Наряду с достоинствами, есть и несколько недостатков:

- требуется отличное знание самого устройства и наличие документации к нему;
- полный учет всех возможных вариантов взаимодействия по требуемому протоколу;
- владение языком ассемблера или C (C++). К сожалению, современные языки (Visual Basic, C#, Delphi, Java и т. д.) все дальше и дальше "уводят" начинающих программистов от интересного, но сложного, к более простому, но очень ограниченному.

Мне бы хотелось рассмотреть документированные способы работы с мышью PS/2. Еще раз повторюсь о том, что информации по данной теме очень мало и здесь представлена вся имеющаяся в свободном доступе информация.

Итак, для непосредственной работы с мышью PS/2 используется два порта контроллера клавиатуры, адресуемые через регистры `64h` и `60h`. Регистр `64h` имеет двойное назначение. В режиме чтения он служит регистром статуса и позволяет получить текущую информацию о состоянии мыши. Этот регистр имеет размер 8 бит и представлен в табл. 2.4.

Таблица 2.4. *Регистр состояния (64h)*

Бит	Описание
0	Наличие данных в выходном буфере мыши (0 — выходной буфер пуст, 1 — буфер содержит данные)
1	Наличие данных во входном буфере мыши (0 — входной буфер пуст, 1 — буфер содержит данные)
2	Результат самотестирования (0 — сброс, 1 — тест прошел успешно)
3	Использование портов контроллером клавиатуры (0 — запись в порт 60h, 1 — запись в порт 64h)

Таблица 2.4 *(окончание)*

Бит	Описание
4	Состояние клавиатуры (0 — заблокирована, 1 — включена)
5	Наличие данных в дополнительном выходном буфере мыши (0 — выходной буфер пуст, 1 — буфер содержит данные)
6	Ошибка тайм-аута (0 — ошибка отсутствует, 1 — ошибка). В случае ошибки, следует повторить передачу данных, используя команду `Resend`
7	Ошибка четности (0 — ошибка отсутствует, 1 — ошибка) указывает на последнюю ошибку, произошедшую при передаче данных

В режиме записи регистр `64h` служит для передачи команд контроллеру клавиатуры. Через запись в этот регистр команды осуществляется управление контроллером клавиатуры. Назначения битов в командном регистре перечислены в табл. 2.5.

Таблица 2.5. *Регистр команд (64h)*

Бит	Описание
0	Прерывания для клавиатуры (0 — отключить, 1 — включить)
1	Прерывания для мыши (0 — отключить, 1 — включить)
2	Системный флаг (1 — инициализация через самотестирование, 0 — инициализация по питанию)
3	Не используется
4	Доступ к клавиатуре (0 — открыт, 1 — закрыт)
5	Доступ к мыши (0 — открыт, 1 — закрыт)
6	Трансляция скан-кодов (0 — не использовать, 1 — использовать)
7	Резерв

Список команд, используемых для мыши PS/2, перечислен в табл. 2.6.

Таблица 2.6. *Команды управления для мыши*

Код команды	Описание
`20h`	Прочитать регистр команд
`60h`	Записать байт в регистр команд
`A7h`	Отключить порт мыши PS/2
`A8h`	Включить порт мыши PS/2

Таблица 2.6 *(окончание)*

Код команды	Описание
A9h	Выполнить самотестирование мыши (байт на выходе равен 00h)
AAh	Выполнить самотестирование контроллера (байт на выходе равен 55h)
C0h	Прочитать входной порт
C1h	Скопировать из входного порта младший разряд байта
C2h	Скопировать из входного порта старший разряд байта
D0h	Прочитать расширенную информацию для выходного порта (0 бит — сброс процессора, 1 бит — канал A20 включен, 2 бит — передача данных от мыши PS/2, 3 бит — синхросигнал от мыши PS/2, 4 бит — выходной буфер заполнен, 5 бит — выходной буфер мыши PS/2 заполнен, 6 бит — синхросигнал для клавиатуры, 7 бит — передача данных от клавиатуры)
D1h	Записать байт данных в выходной порт
D3h	Записать байт данных в выходной буфер мыши PS/2
D4h	Записать байт данных для мыши PS2
E0h	Тестирование порта (возвращает тестовое значение для порта)

Для обмена данными с мышью используется регистр 60h, называемый *регистром данных*. После выполнения какой-либо команды в данный регистр помещается код ошибки, по которому можно судить о результате операции. Возможные значения кодов ошибок представлены в табл. 2.7. Если команда управления является расширенной (имеет размер два байта), то первый байт команды должен быть записан в регистр команд, а второй байт — в регистр данных.

Таблица 2.7. *Коды ошибок мыши*

Код ошибки	Описание
AAh	Самотестирование контроллера успешно выполнено
FAh	Команда успешно передана
FCh	Ошибка самотестирования мыши
FEh	Запрос на повторную передачу данных (команда Resend)

После того, как были рассмотрены основные команды контроллера клавиатуры для управления устройством мыши PS/2, приведу несколько примеров их использования. Пример кода, позволяющий протестировать интерфейс мыши PS/2, показан в листинге 2.1.

Листинг 2.1. Тестирование интерфейса мыши PS/2

```
DWORD dwResult  = 0; // переменная для хранения результата
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду самотестирования
outPort ( 0x64, 0xA9, 1 );
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// самотестирование мыши успешно завершено
```

Если вы заметили, мы постоянно проверяем регистр статуса (точнее бит 2) на наличие данных во входном буфере. В реальных программах дополнительно следует проверять ошибки тайм-аута (бит 6) и четности (бит 7).

Рассмотрим еще один пример, позволяющий программно отключить мышь PS/2 (см. листинг 2.2).

Листинг 2.2. Отключение мыши PS/2

```
DWORD dwResult  = 0; // переменная для хранения результата
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
```

```
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду отключения мыши PS/2
outPort ( 0x64, 0xA7, 1 );
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// мышь успешно заблокирована
```

В обоих примерах используются команды управления контроллером клавиатуры, но для мыши PS/2 существует еще дополнительный набор команд, разработанных специально для нее. Некоторые производители мышей могут добавлять свои специфические команды, которые можно найти в технической документации, предоставляемой разработчикам. Список всех стандартных команд для мыши PS/2 представлен в табл. 2.8.

Таблица 2.8. *Набор команд для мыши PS/2*

Код команды	Команда	Описание
FFh	Reset	Программный сброс
FEh	Resend	Проверка данных, переданных от мыши
F6h	Set Defaults	Установка параметров работы по умолчанию
F5h	Disable	Отключение
F4h	Enable	Включение
F3h	Set Sample Rate	Установка частоты дискретизации
F2h	Read Device Type	Получение идентификатора устройства
F0h	Set Remote Mode	Включение дистанционного режима работы
EEh	Set Wrap Mode	Включение эхо-режима
ECh	Reset Wrap Mode	Отключение эхо-режима
EBh	Read Data	Чтение данных

Таблица 2.8 *(окончание)*

Код команды	Команда	Описание
EAh	Set Stream Mode	Установка потокового режима передачи данных
E9h	Status Request	Получение текущего состояния
E8h	Set Resolution	Установка разрешения
E7h	Set Scaling 2:1	Установка режима масштабирования 2:1
E6h	Set Scaling 1:1	Установка режима масштабирования 1:1

Процесс вызова каждой дополнительной команды PS/2 немного отличается от прямого управления контроллером клавиатуры. Вначале выполняется команда D4h (записать байт данных для мыши PS2), а только потом в регистр данных записывается код команды мыши PS/2 (например, FFh для сброса мыши). Говоря проще, каждая дополнительная команда PS/2 имеет размер два байта (D4h и любой код команды), где первый байт записывается в регистр команд (64h), а второй байт — в регистр данных (60h). Если команда PS/2 требует передачи дополнительных данных (например, новое значение частоты дискретизации), то эта информация должна быть записана в регистр данных (60h), а первый байт D4h также записывается в командный регистр. Далее приведены примеры использования команд для мыши PS/2, в которых можно будет наглядно убедиться в этом. А теперь рассмотрим набор команд управления мышью PS/2 более подробно.

2.2.1 Команда *Reset* (*FFh*)

Команда Reset (FFh) предназначена для программного сброса устройства мыши PS/2. После успешного выполнения команды (через 300—500 мс) мышь возвращает код FAh и комбинацию AAh и 00h. После этого мышь устанавливает режим сброса. Далее, в листинге 2.3 приводится упрощенный пример кода для программного сброса мыши PS/2.

Листинг 2.3. Выполнение сброса контроллера мыши PS/2

```
DWORD dwResult  = 0; // переменная для хранения результата
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
```

```
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду передачи данных
outPort ( 0x64, 0xD4, 1 );
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду Reset
outPort ( 0x60, 0xFF, 1 );
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// программный сброс мыши завершен
```

Дополнительно, после выполнения команды сброса, можно проверить регистр `60h` на наличие кода `FAh`.

2.2.2. Команда *Resend* (*FEh*)

Команда `Resend` (`FEh`) предназначена для проверки данных, поступивших от мыши. Компьютер посылает эту команду, если переданные от мыши данные имеют недопустимое значение. Получив ее, мышь выполняет повторную передачу последнего пакета данных. Если данные опять ошибочны, то компьютер может повторить запрос командой `Resend` либо выполнить программный сброс командой `Reset`. В любом случае регистр данных (`60h`) будет содержать значение `FAh`.

2.2.3. Команда *Set Defaults* (*F6h*)

Команда Set Defaults (F6h) позволяет установить все параметры мыши, используемые по умолчанию. После записи в регистр данных кода FAh мышь установит следующие значения: масштаб — 1:1, разрешение — 4 отсчета на миллиметр, частота дискретизации — 100 выборок и потоковый режим.

2.2.4. Команда *Disable* (*F5h*)

Команда Disable (F5h) позволяет отключить обмен данных между мышью и компьютером в потоковом режиме. Кроме этого, все счетчики движения сбрасываются в нулевое состояние. При этом все другие операции могут осуществляться. Выполнение команды подтверждается записью в регистр данных кода FAh.

2.2.5. Команда *Enable* (*F4h*)

Команда Enable (F4h) позволяет продолжить передачу данных между мышью и компьютером, если установлен потоковый режим. Выполнение команды подтверждается записью в регистр данных кода FAh.

2.2.6. Команда *Set Sample Rate* (*F3h*)

Команда Set Sample Rate (F3h) позволяет установить частоту дискретизации для мыши PS/2. Команда является двухбайтовой. Вначале в регистр данных записывается код команды F3h. Мышь подтверждает получение команды записью в регистр данных кода FAh. После этого в регистр данных записывается второй байт команды, который содержит значение частоты дискретизации. Мышь вторично подтверждает получение, записав в регистр данных код FAh. Возможные значения частоты дискретизации перечислены в табл. 2.9.

Таблица 2.9. *Значения кодов частоты дискретизации*

Значение кода	Частота дискретизации, выборок/сек
00h	10
01h	20
02h	40
03h	60
04h	80
05h	100
06h	200

Для лучшего понимания работы с данной командой приведу пример ее использования в листинге 2.4.

Листинг 2.4. Установка частоты дискретизации для мыши PS/2

```
DWORD dwResult  = 0; // переменная для хранения результата
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
       if ( (dwResult & 0x02 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду передачи данных
outPort ( 0x64, 0xD4, 1 );
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
       if ( (dwResult & 0x02 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду Set Sample Rate
outPort ( 0x60, 0xF3, 1 );
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
       if ( (dwResult & 0x02 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду передачи данных
outPort ( 0x64, 0xD4, 1 );
iTimeWait = 50000;
```

```
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт новое значение частоты 100 выборок
outPort ( 0x60, 0x64, 1 );
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// проверяем наличие кода FAh в регистре данных
inPort ( 0x60, &dwResult, 1 );
if ( dwResult != 0xFA)
        return MY_ERROR;
// новая частота дискретизации 100 выборок в секунду установлена
```

2.2.7. Команда *Read Device Type* (*F2h*)

Команда `Read Device Type` (`F2h`) позволяет получить идентификатор устройства. Для мыши PS/2 это значение всегда равно `00h`. Мышь подтверждает получение команды записью в регистр данных кода `FAh`.

2.2.8. Команда *Set Remote Mode* (*F0h*)

Команда `Set Remote Mode` (`F0h`) позволяет включить дистанционный режим управления мышью PS/2. Мышь подтверждает получение команды записью в регистр данных кода `FAh`. В этом режиме мышь передает данные на компьютер только после подачи последним запроса чтения с помощью команды `EBh`.

2.2.9. Команда *Set Wrap Mode* (*EEh*)

Команда `Set Wrap Mode` (`EEh`) позволяет установить эхо-режим для мыши PS/2. Мышь подтверждает получение команды записью в регистр данных кода `FAh`. В этом режиме любой байт, кроме `FFh` (`Reset`) и `ECh` (`Reset Wrap Mode`), будет возвращен устройством мыши на компьютер.

2.2.10. Команда *Reset Wrap Mode* (*ECh*)

Команда `Reset Wrap Mode` (`ECh`) позволяет выйти из эхо-режима и восстановить прежний режим. Если передача данных в потоковом режиме была заблокирована командой `Disable`, то прежний режим не будет восстановлен. Кроме того, выполнение данной команды в любом другом режиме не будет иметь никакого результата. Мышь подтверждает получение команды записью в регистр данных кода `FAh`.

2.2.11. Команда *Read Data* (*EBh*)

Команда `Read Data` (`EBh`) позволяет получить текущие значения счетчиков перемещения (по осям *X* и *Y*), а также информацию о состоянии кнопок. После успешной передачи данных счетчики перемещения сбрасываются в ноль. Мышь подтверждает получение команды записью в регистр данных кода `FAh`. Если мышь находится в дистанционном режиме, то выполнение данной команды является единственным способом получения данных.

2.2.12. Команда *Set Stream Mode* (*EAh*)

Команда `Set Stream Mode` (`EAh`) позволяет активизировать потоковый режим передачи данных для мыши PS/2. Мышь подтверждает получение команды записью в регистр данных кода `FAh`. В этом режиме данные от мыши поступают при любом движении или изменении состояния кнопок.

2.2.13. Команда *Status Request* (*E9h*)

Команда `Status Request` (`E9h`) позволяет получить пакет данных (три байта) о текущем состоянии счетчиков движения и кнопок мыши PS/2. Формат данных представлен в табл. 2.10. Мышь подтверждает получение команды записью в регистр данных кода `FAh`. Например, если выполнить команду `Reset`, то `Status Request` возвратит 4 байта в следующей очередности: `FAh`, `00h`, `02h` и `64h`. Первый байт указывает на завершение команды. Второй байт — масштаб 1:1, принятый по умолчанию. Третий байт — разрешение 4 отсчета на миллиметр. Четвертый байт — частоту дискретизации 100 выборок в секунду.

Таблица 2.10. *Формат данных состояния мыши*

Байт	Бит							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	0	Режим	Передача	Масштаб	0	Левая кнопка	Средняя кнопка	Правая кнопка
2	0	0	0	0	0	0	Разрешение	
3	Частота дискретизации							

Описание табл. 2.10:

- Правая кнопка

  Определяет состояние правой кнопки мыши: 1 — кнопка нажата, 0 — кнопка отпущена.

- Средняя кнопка

  Определяет состояние средней кнопки мыши: 1 — кнопка нажата, 0 — кнопка отпущена.

- Левая кнопка

  Определяет состояние левой кнопки мыши: 1 — кнопка нажата, 0 — кнопка отпущена.

- Масштаб

  Определяет текущий режим масштабирования: 1 — режим 2:1, 0 — 1:1.

- Передача

  Определяет передачу данных для потокового режима: 1 — передача разрешена, 0 — передача запрещена.

- Режим

  Определяет текущий режим работы мыши: 1 — дистанционный, 0 — потоковый.

- Разрешение

  Определяет текущее разрешение мыши. Может принимать значение от `0h` до `3h`.

- Частота дискретизации

  Определяет текущую частоту дискретизации мыши. Может принимать значение от 10 до 200.

Для наглядности приведу простой пример использования команды `Status Request` в листинге 2.5.

Листинг 2.5. Использование команды `Status Request`

```
// переменная для хранения результата
DWORD dwResult  = 0;
// переменные для хранения трех байтов текущего состояния мыши
BYTE bStaus_1 = 0, bStaus_2 = 0, bStaus_3 = 0
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
       if ( (dwResult & 0x02 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду передачи данных
outPort ( 0x64, 0xD4, 1 );
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
       if ( (dwResult & 0x02 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт код команды Status Request
outPort ( 0x60, 0xE9, 1 );
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
       if ( (dwResult & 0x20 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// проверяем код выполнения команды FAh
inPort ( 0x60, &dwResult, 1 );
if ( dwResult != 0xFA)
       return MY_ERROR;
```

```
// ожидаем данные
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x20 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// получаем первый байт состояния
inPort ( 0x60, &dwResult, 1 );
// сохраняем значение первого байта в переменной
bStaus_1 = (BYTE) dwResult;
// ожидаем данные
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x20 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// получаем второй байт состояния
inPort ( 0x60, &dwResult, 1 );
// сохраняем значение второго байта в переменной
bStaus_2 = (BYTE) dwResult;
// ожидаем данные
iTimeWait = 50000;
// проверяем, успешно ли передана команда
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x20 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
```

```
// получаем третий байт состояния
inPort ( 0x60, &dwResult, 1 );
// сохраняем значение третьего байта в переменной
bStaus_3 = (BYTE) dwResult;
```

2.2.14. Команда *Set Resolution* (*E8h*)

Команда `Set Resolution` (`E8h`) предназначена для установки разрешения мыши PS/2. Команда является двухбайтовой. Вначале в регистр данных записывается код команды `E8h`. Мышь подтверждает получение команды записью в регистр данных кода `FAh`. После этого в регистр данных записывается второй байт команды, который содержит новое значение разрешения. Мышь вторично подтверждает получение, записав в регистр данных код `FAh`. Возможные значения кода разрешения перечислены в табл. 2.11.

Таблица 2.11. *Значения кодов разрешения*

Значение кода	Разрешение, отсчетов/мм
00h	1
01h	2
02h	4
03h	8

2.2.15. Команда *Set Scaling 2:1* (*E7h*)

Команда `Set Scaling 2:1` (`E7h`) предназначена для установки режима масштабирования 2:1, позволяющего улучшить чувствительность мыши за счет нелинейного пересчета датчиков движения (см. табл. 2.3). Мышь подтверждает получение команды записью в регистр данных кода `FAh`.

2.2.16. Команда *Set Scaling 1:1* (*E6h*)

Команда `Set Scaling 1:1` (`E6h`) позволяет восстановить режим масштабирования, используемый по умолчанию и равный 1:1. Мышь подтверждает получение команды записью в регистр данных кода `FAh`.

На этом можно завершить описание команд управления мышью PS/2. Хочу только заметить, что желательно в своих программах проверять различные ошибки (см. табл. 2.7) завершения, а не только код `FAh`. Это позволит точнее определить проблему сбоя и принять соответствующие меры.

Теперь рассмотрим, какие возможности для поддержки устройства мыши предлагает нам фирма Microsoft.

2.3. Использование Win32 API

Набор возможностей, предоставляемый этим программным интерфейсом, небогат, но позволяет решить основные вопросы использования мыши в операционных системах Windows. Общие темы, которые мы рассмотрим здесь, можно свести к короткому списку:

- Настройка мыши.
- Работа с курсором.

2.3.1. Настройка мыши

Удобство работы с мышью в программе часто служит определяющим фактором, по которому судят о качестве софта. Особенно это касается пользователей, использующих левую руку в качестве основной, проще говоря, левшей. Мало кто из разработчиков программного обеспечения думает об этом. Конечно, в настройках Windows есть возможность поменять местами левую и правую кнопки мыши, но у рядового пользователя в связи с этим возникают проблемы. Во-первых, нужно неплохо знать систему, чтобы найти нужные настройки. Во-вторых, такую операцию приходится выполнять достаточно часто, что отвлекает от основной работы на компьютере. Было бы гораздо проще, если бы каждый разработчик программного обеспечения думал об этом заранее и добавлял в свою программу возможность менять значения левой и правок кнопок мыши. Это добавит программе только плюс.

Программно решить эту задачу в Windows не сложно. Существует два способа, о которых я хотел бы рассказать. Прежде всего, замечу, что оба способа изменяют глобальные настройки системы, поэтому следует восстанавливать стандартные значения перед закрытием своей программы. Первый способ состоит из вызова функции `SwapMouseButton`. Данная функция имеет всего один аргумент (тип BOOL), который определяет направление операции. Чтобы поменять местами левую и правую кнопки мыши, присвойте аргументу значение TRUE. Для восстановления значения по умолчанию вызовите функцию со значением FALSE. Функция `SwapMouseButton` возвратит нулевое значение, если до этого назначение кнопок не менялось, и единицу в обратном случае. Второй способ состоит в использовании универсальной функции `SystemParametersInfo` с установленной опцией `SPI_SETMOUSEBUTTONSWAP`. Для лучшего понимания просмотрите примеры, представленные в листингах 2.6 и 2.7.

Листинг 2.6. Использование функции `SwapMouseButton`

```
// напишем свою функцию для смены кнопок мыши
bool LeftToRight ( bool bLR )
```

```
{
       if ( SwapMouseButton ( bLR ) != 0 )
       // состояние кнопок уже изменено
       return false;
return true;
}
// используем нашу функцию в коде
// . . .
void MouseButtons ( )
{
       if ( !LeftToRight ( true ) )
       // если состояние кнопок уже менялось, восстановим по умолчанию
              LeftToRight ( false );
}
```

Для того чтобы узнать текущее состояние кнопок, можно воспользоваться функцией `GetSystemMetrics` с аргументом `SM_SWAPBUTTON`. Если значения левой и правой кнопок были изменены, функция возвратит ненулевое значение, иначе результат будет равен нулю.

Листинг 2.7. Использование функции `SystemParametersInfo`

```
// напишем свою функцию для смены кнопок мыши
bool LeftToRight ( bool bLR )
{
   SystemParametersInfo ( SPI_SETMOUSEBUTTONSWAP, (UINT) bLR, NULL,
SPIF_UPDATEINIFILE );
}
```

Вызов функции `SystemParametersInfo` записывает в системный файл WIN.INI значение в виде цифры (например, "1"), однако при последующей загрузке Windows параметры мыши могут остаться прежними, независимо от ваших действий. Происходит это потому, что панель управления тоже записывает в указанный системный файл информацию о состоянии кнопок мыши, но она использует буквенное обозначение (например, вместо 1 слово "Yes"). Приоритет всегда будет отдан буквенному значению ("Yes" или "No"), поэтому, чтобы избежать таких неожиданностей, следует использовать функцию `SwapMouseButton`.

Кроме изменения состояния левой и правой кнопок мыши, возможно программно менять время реакции системы на двойной щелчок. Для этого используется функция `SetDoubleClickTime`, имеющая всего один аргумент, ко-

торый задает максимальное значение времени ожидания (в миллисекундах) между первым и вторым щелчком мыши. Значение по умолчанию, используемое в Windows, равно 500 мс. Для установки времени реакции системы на двойной щелчок по умолчанию достаточно вызвать функцию `SetDoubleClickTime` с нулевым значением аргумента. В случае ошибки функция возвращает 0 и следует вызвать `GetLastError` для получения детального описания возникшей ситуации.

Кроме `SetDoubleClickTime`, существует функция `GetDoubleClickTime`, которая позволяет узнать текущее значение времени реакции системы на двойной щелчок мыши. Эта функция не имеет аргументов и возвращает тип UINT, характеризующий время реакции системы на двойной щелчок мыши (в миллисекундах). В листинге 2.8 приводится пример работы с функциями `SetDoubleClickTime` и `GetDoubleClickTime`.

Листинг 2.8 Использование функций `SetDoubleClickTime` и `GetDoubleClickTime`

```
// напишем собственную функцию установки времени двойного щелчка мыши
bool SetDblClickTime ( unsigned int uTime )
{
   unsigned int uCurrentTime = 0;
   // получаем текущее значение
   uCurrentTime = GetDoubleClickTime ( );
   // если оно совпадает с устанавливаемым, выходим из функции
   if ( uTime == uCurrentTime )
       return false;
   // если нет, устанавливаем новое значение
   if ( SetDoubleClickTime ( uTime ) == 0 )
      {
         // произошла ошибка, вызываем GetLastError, если нужно
         return false;
       }
return true;
}
```

Как видно из примеров, работа с этими функциями не представляет никаких проблем. Хочу только заметить, что изменение времени реакции между двумя щелчками мышью действует для всей операционной системы.

Кроме описанного способа, существует еще один, использующий универсальную функцию `SystemParametersInfo`. Для установки текущего значения времени применяется параметр `SPI_SETDOUBLECLICKTIME`.

Существует еще один параметр, которым можно управлять программно — скорость движения указателя мыши по экрану. Для реализации этой возможности применяется уже известная нам функция SystemParametersInfo. Получить текущее значение скорости можно, вызвав ее с параметром SPI_GETMOUSESPEED, а установить новое значение — параметром SPI_SETMOUSESPEED. Диапазон возможных значений лежит в пределах от 1 до 20. По умолчанию используется 10. Пример функции, изменяющей скорость движения указателя, представлен в листинге 2.9.

Листинг 2.9. Установка скорости движения указателя мыши

```
// напишем свою функцию для управления скоростью движения указателя мыши
int SpeedMouse ( int iSpeed, bool bFlag )
{
   if ( bFlag ) // установить скорость
     {
       if ( ( iSpeed > 0 ) && ( iSpeed < 21 ) ) // проверяем диапазон
        {
          SystemParametersInfo ( SPI_SETMOUSESPEED, 0, (PVOID) iSpeed,
                                 SPIF_SENDCHANGE | SPIF_UPDATEINIFILE );
        }
     }
   else // получаем текущее значение скорости
     {
       int iCurrentSpeed = 10; // значение по умолчанию
       SystemParametersInfo ( SPI_GETMOUSESPEED, 0, &iCurrentSpeed, 0 );
       return iCurrentSpeed;
     }
return 0;
}
```

Представленная читателям функция SpeedMouse имеет два аргумента. Первый передает значение скорости, а второй указывает на тип выполняемой операции (установка или получение текущего значения скорости).

И последняя возможность управления мышью, которую я хотел бы рассмотреть, представляет собой блокировку всех событий мыши. Функция называется BlockInput и позволяет блокировать не только мышь, но и клавиатуру. Единственный аргумент функции, установленный в TRUE, блокирует мышь и клавиатуру. В случае ошибки функция возвращает 0 и следует вызвать GetLastError для получения детального описания возникшей ситуации. Пример работы данной функции приводится в листинге 2.10.

Листинг 2.10. Блокировки мыши и клавиатуры

```
// обязательно включите файл Winable.h
#include <Winable.h>
// пишем функцию блокировки
bool LockMouse ( bool Action )
{
    if ( BlockInput ( Action ) == 0 )
     {
        // ошибка выполнения
        return false;
     }
return true;
}
```

Прежде чем использовать функцию `BlockInput`, следует учесть несколько важных факторов:

- функция полностью блокирует мышь и клавиатуру, поэтому в программе следует задавать время по таймеру, после которого устройства будут разблокированы;
- восстановить доступ к устройствам можно перезагрузкой компьютера или же нажав на клавиатуре комбинацию клавиш <Ctrl>+<Alt>+<Del>.

Из сказанного можно сделать основной вывод: разработчик программы, использующий функцию `BlockInput`, должен позаботиться о полном восстановлении работы мыши и клавиатуры. В процессе блокировки программист может применять функцию `SendInput`, имитирующую нажатия кнопок мыши и клавиатуры.

И последняя возможность, о которой хотелось бы рассказать, — получение краткой информации о мыши. Чтобы определить, установлена ли в системе мышь, можно вызвать функцию `GetSystemMetrics` со значением `SM_MOUSEPRESENT`. Если мышь установлена, функция возвратит значение больше 0, иначе будет возвращен 0. В листинге 2.11 приводится пример использования этой функции.

Листинг 2.11. Проверка наличия в системе устройства мыши

```
// пишем функцию для проверки подключения мыши
bool IsMouse ( )
{
    if ( GetSystemMetrics ( SM_MOUSEPRESENT ) )
        return true; // мышь установлена
return false;
}
```

Кроме определения наличия в системе мыши, можно получить данные о количестве кнопок и присутствии колеса прокрутки. Для получения количества кнопок мыши вызываем функцию `GetSystemMetrics` со значением `SM_CMOUSEBUTTONS`, а для проверки колеса прокрутки мыши — со значением `SM_MOUSEWHEELPRESENT`. В первом случае функция вернет количество кнопок или значение 0, если мышь отсутствует. Во втором случае функция возвратит ненулевое значение, если колесо прокрутки имеется. В листинге 2.12 показано, как это делается.

Листинг 2.12. Получение информации о числе кнопок и наличии колеса прокрутки

```
// напишем функцию, определяющую число кнопок мыши
int GetMouseButtons ( )
{
    int iNum = 0;
    iNum = GetSystemMetrics (SM_CMOUSEBUTTONS );
    if ( iNum )
        return iNum; // количество кнопок
return iNum; // мышь отсутствует
}
// напишем функцию для определения колеса прокрутки
bool IsMouseWheel ( )
{
if ( GetSystemMetrics ( SM_MOUSEWHEELPRESENT ) )
        return true; // мышь установлена
return false;
}
```

2.3.2. Работа с курсором

В Windows курсор мыши представляет собой черно-белый или цветной значок и может быть статическим или динамическим (так называемый "живой" указатель). В системе всегда имеется стандартный набор курсоров для использования в различных ситуациях. Этот набор называется *системным* и может применяться в программе по умолчанию. Мы рассмотрим наиболее часто востребованную возможность работы с курсорами для создания визуального ожидания завершения какой-либо операции в программе. Те читатели, которые знакомы с классом `CWaitCursor` из библиотеки *MFC* (Microsoft Foundation Classes), сразу поймут, о чем идет речь. В каждой программе так или иначе встречаются функции, требующие определенного времени для завершения. Чтобы пользователь видел, что программа не "зависла", а про-

должает выполняться, необходимо как-то это сообщить ему. Обычно, текущий курсор (как правило, в виде стрелки) на время выполнения операции заменяется другим (как правило, в виде часов). Для реализации такой возможности мы создадим полноценный класс, который можно будет легко использовать в различных программах. Назовем наш класс `CWaitMouse`. Файл объявлений (CWaitMouse.h) представлен в листинге 2.13, а файл реализации (CWaitMouse.cpp) — в листинге 2.14.

Листинг 2.13. Файл CWaitMouse.h

```
class CWaitMouse
{
public:
       CWaitMouse ( );     // конструктор по умолчанию
       ~ CWaitMouse () { } // пустой деструктор
       // общедоступные функции
       void Wait ( );    // функция отображает курсор ожидания
       void Restore ( ); // функция восстанавливает прежний курсор
private:
      bool bStatus; // хранит текущее состояние курсора
      void SetWait ( bool bWait ); // установка курсора ожидания
};
```

Листинг 2.14. Файл CWaitMouse.cpp

```
#include " CWaitMouse.h"
// прямо в конструкторе изменяем вид курсора
CWaitMouse :: CWaitMouse ( )
{
      bStatus = false; // инициализация
      SetWait ( true ); // устанавливаем курсор ожидания
}
// функция восстанавливает прежний вид курсора
void CWaitMouse :: Restore( )
{
       SetWait ( false );
}
// функция устанавливает курсор ожидания
void CWaitMouse :: Wait ( )
{
       // проверяем, не установлен ли курсор ожидания ранее
       if ( bStatus ) return;
```

```
       // если нет, устанавливаем
       SetWait ( true );
}
// функция обработки курсора
void CwaitMouse :: SetWait (bool bWait )
{
      // дескриптор для нового курсора
      HCURSOR hCursor = NULL;
      // дескриптор для хранения старого курсора
      static HCURSOR hStarikCursor = NULL;
      // проверяем тип операции
      if ( bWait ) // показать курсор ожидания
       {
          hCursor = LoadCursor (NULL, IDC_WAIT );
          hStarikCursor = SetCursor ( hCursor );
          bStatus = true;
       }
      else // восстановить прежний курсор мыши
       {
          SetCursor ( hStarikCursor );
          bStatus = false;
       }
} // конец функции
```

Внимательно изучив код нашего класса, вы увидите, что вся основная работа выполняется в функции `SetWait`. Для загрузки курсора ожидания мы выполнили два основных действия: загрузили стандартный курсор ожидания (`IDC_WAIT`) и заменили им текущий курсор. Загрузка курсора выполнена с помощью библиотечной функции `LoadCursor`. Данная функция имеет два параметра: первый указывает на дескриптор нашей программы (тип `HINSTANCE`), а второй определяет тип курсора для мыши. Поскольку мы используем стандартный курсор, первый аргумент можно установить в NULL. Второму аргументу присваиваем одно из значений, перечисленных в табл. 2.12.

Таблица 2.12. *Возможные значения для системных курсоров*

Константа	Описание
IDC_WAIT	Значок "Песочные часы"
IDC_ARROW	Значок стандартной стрелки
IDC_CROSS	Значок перекрестья
IDC_HELP	Значок в виде вопросительного знака со стрелкой
IDC_HAND	Значок в виде руки

Таблица 2.12 *(окончание)*

Константа	Описание
IDC_UPARROW	Значок в виде вертикальной стрелки
IDC_SIZEALL	Значок изменения размеров

После успешного завершения функция LoadCursor вернет дескриптор (HCURSOR) загруженного курсора. Для вывода нового курсора на экран необходимо вызвать функцию SetCursor. Она имеет один аргумент, куда заносится дескриптор желаемого курсора. В случае успешного завершения функция вернет дескриптор старого курсора, который мы сохраняем в статической переменной для последующего восстановления. Вот и все премудрости.

Существует возможность программно скрыть курсор с экрана. Для этого применяется функция ShowCursor. Функция имеет всего один аргумент. Если нужно скрыть курсор, вызываем эту функцию с аргументом FALSE. Хитрость состоит в том, что функция при каждом вызове увеличивает (TRUE) или уменьшает (FALSE) внутренний системный счетчик. Когда значение счетчика больше или равно 0, курсор видим на экране. Если мышь присутствует, начальное значение счетчика равно 0, иначе –1. Исходя из ранее изложенного, необходимо следить, чтобы каждый вызов функции ShowCursor имел повторный ее вызов в программе. После выполнения функция возвращает текущее значение счетчика. Примеры работы с данной функцией не привожу, поскольку она достаточно проста в использовании.

На этом мне бы хотелось завершить тему программирования мыши и перейти к не менее важному устройству — клавиатуре.

ГЛАВА 3

Клавиатура

Клавиатура является одним из самых старых, но не менее актуальных и сегодня устройств. Без нее трудно представить полноценный компьютер. Несмотря на бурное развитие программных средств голосового управления компьютером, клавиатура остается наиболее надежным и совершенным средством управления программным обеспечением. Важность ее трудно переоценить, что позволяет мне утверждать о необходимости изучения различных способов программирования этого привычного для каждого пользователя устройства.

Мы рассмотрим два основных способа программирования клавиатуры:

1. Работу с контроллером клавиатуры напрямую через порты.
2. Программирование клавиатуры в Win32 API.

Каждый программист сможет выбрать один из двух вариантов, в зависимости от поставленной задачи, но, прежде чем мы приступим к рассмотрению этих способов, позвольте мне немного рассказать о самом устройстве клавиатуры.

3.1. Общие сведения

На сегодняшний день существует два основных типа клавиатуры: AT и PS/2. Первый тип уже морально устарел и практически полностью вытеснен современным стандартом PS/2. Кроме основных типов клавиатур, существуют и другие, реже встречающиеся устройства: USB и инфракрасные. Для поддержки клавиатуры используется интегрированный в системный чип контроллер (например, Intel 8042 или VIA 8242), одновременно поддерживающий и мышь PS/2.

По количеству клавиш клавиатуры можно разделить на несколько групп:

1. Клавиатуры XT с 83 клавишами. Появились в 1981 году. Использовали 5-штырьковый разъем DIN. Передача данных была организована по однонаправленному последовательному протоколу. На данный момент полностью устарели.
2. Клавиатуры AT с 84—101 клавишами. Появились в 1984 году. Используют 5-штырьковый разъем DIN. Передача данных организована по двунаправленному последовательному протоколу.
3. Клавиатуры PS/2 с 84—101 клавишами. Появились в 1987 году. Используют 5-штырьковый разъем mini-DIN. Передача данных организована по двунаправленному последовательному протоколу.
4. Современные клавиатуры PS/2 с 101—104 (или более) клавишами. Используют 6-штырьковый разъем mini-DIN. Передача данных организована по двунаправленному последовательному протоколу.

Как правило, все клавиатуры состоят из набора клавиш и встроенного микропроцессора. Главная задача микропроцессора состоит в передаче скан-кода нажатой (отпущенной) клавиши (комбинаций клавиш) на компьютер и обработку управляющих команд от последнего. Для ускорения работы имеется встроенный буфер (обычно 16 байт), позволяющий хранить данные обмена между клавиатурой и компьютером. Для управления клавиатурой разработан набор специальных управляющих команд. Следует иметь в виду, что некоторые современные PS/2 клавиатуры могут не поддерживать отдельные команды. Команды управления рассмотрены далее в этой главе. Обмен данными между клавиатурой и компьютером (контроллером клавиатуры, расположенным на материнской плате) работает по протоколу, разработанному фирмой IBM. Встроенный в клавиатуру микропроцессор сканирует шину в ожидании нажатия (отпускания) любой клавиши. Если произошло нажатие (отпускание или удержание) какой-либо клавиши, вырабатывается определенный *скан-код* (набор скан-кодов), позволяющий компьютеру определить текущее состояние клавиатуры. Каждая клавиша имеет свой уникальный скан-код.

Стандартный набор значений скан-кодов для клавиатуры представлен в табл. 3.1, а расширенный — в табл. 3.2. Кроме того, в табл. 3.3 представлены коды управляющих символов ASCII.

Каждый переданный компьютеру скан-код (числовое значение) обрабатывается и преобразовывается в код ASCII, который и применяется для передачи смыслового содержания нажатой клавиши. Скан-код для стандартной клавиатуры (84 клавиши) имеет размер 1 байт, а для расширенной — от двух до четырех байтов. Чтобы отличить расширенный скан-код от обычного, в качестве первого байта всегда выступает значение `E0h` (например, код левой <Alt>

Таблица 3.1. *Скан-коды клавиатуры*

Клавиша	Код	Клавиша	Код	Клавиша	Код
<Esc>	01h	<Enter>	1Ch	<*>	37h
<1> и <!>	02h	<Ctrl>	1Dh	<Alt>	38h
<2> и <@>	03h	<A>	1Eh	<Space>	39h
<3> и <#>	04h	<S>	1Fh	<Caps Lock>	3Ah
<4 и <$>	05h	<D>	20h	<F1>	3Bh
<5> и <%>	06h	<F>	21h	<F2>	3Ch
<6> и <^>	07h	<G>	22h	<F3>	3Dh
<7> и <&>	08h	<H>	23h	<F4>	3Eh
<8> и <*>	09h	<J>	24h	<F5>	3Fh
<9> и <(>	0Ah	<K>	25h	<F6>	40h
<0> и <)>	0Bh	<L>	26h	<F7>	41h
<-> и <_>	0Ch	<;> и <:>	27h	<F8>	42h
<=> и <+>	0Dh	<‘> и <">	28h	<F9>	43h
<Backspace>	0Eh	<`> и <~>	29h	<F10>	44h
<Tab>	0Fh	<Left Shift>	2Ah	<Num Lock>	45h
<Q>	10h	<\> и <|>	2Bh	<Scroll Lock>	46h
<W>	11h	<Z>	2Ch	<Home>	47h
<E>	12h	<X>	2Dh	<↑>	48h
<R>	13h	<C>	2Eh	<Page Up>	49h
<T>	14h	<V>	2Fh	<->	4Ah
<Y>	15h	<B>	30h	<←>	4Bh
<U>	16h	<N>	31h	<5>	4Ch
<I>	17h	<M>	32h	<→>	4Dh
<O>	18h	< , > и < < >	33h	<+>	4Eh
<P>	19h	< . > и < > >	34h	<End>	4Fh
<[> и <{>	1Ah	</> и <?>	35h	<↓>	50h
<]> и <}>	1Bh	<Right Shift>	36h	<Page Down>	51h
<Insert>	52h	<Delete>	53h		

Таблица 3.2. *Коды для расширенной клавиатуры*

Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
<F1>	3Bh	<Alt>+<F1>	68h	<Shift>+<F1>	54h	<Ctrl>+<F1>	5Eh
<F2>	3Ch	<Alt>+<F2>	69h	<Shift>+<F2>	55h	<Ctrl>+<F2>	5Fh
<F3>	3Dh	<Alt>+<F3>	6Ah	<Shift>+<F3>	56h	<Ctrl>+<F3>	60h
<F4>	3Eh	<Alt>+<F4>	6Bh	<Shift>+<F4>	57h	<Ctrl>+<F4>	61h
<F5>	3Fh	<Alt>+<F5>	6Ch	<Shift>+<F5>	58h	<Ctrl>+<F5>	62h
<F6>	40h	<Alt>+<F6>	6Dh	<Shift>+<F6>	59h	<Ctrl>+<F6>	63h
<F7>	41h	<Alt>+<F7>	6Eh	<Shift>+<F7>	5Ah	<Ctrl>+<F7>	64h
<F8>	42h	<Alt>+<F8>	6Fh	<Shift>+<F8>	5Bh	<Ctrl>+<F8>	65h
<F9>	43h	<Alt>+<F9>	70h	<Shift>+<F9>	5Ch	<Ctrl>+<F9>	66h
<F10>	44h	<Alt>+<F10>	71h	<Shift>+<F10>	5Dh	<Ctrl>+<F10>	67h
<F11>	85h	<Alt>+<F11>	8Bh	<Shift>+<F11>	87h	<Ctrl>+<F11>	89h
<F12>	86h	<Alt>+<F12>	8Ch	<Shift>+<F12>	88h	<Ctrl>+<F12>	8Ah

Таблица 3.3. *Управляющие символы ASCII*

Код	Название	Описание
0	NUL	Конец строки или пустой символ
1	SOH	Начало заголовка
2	STX	Начало текста
3	ETX	Конец текста
4	EOT	Конец передачи
5	ENQ	Подтверждающий запрос
6	ACK	Подтверждение
7	BEL	Звуковой сигнал
8	BS	Возврат на одну позицию влево
9	HT	Горизонтальная табуляция
0A	LF	Перевод строки
0B	VT	Вертикальная табуляция
0C	FF	Переход на новую страницу
0D	CR	Возврат каретки
0E	SO	Нижний регистр

Таблица 3.3 *(окончание)*

Код	Название	Описание
0F	SI	Верхний регистр
10	DLE	Освободить канал связи
11	DC1	Управление устройством 1
12	DC2	Управление устройством 2
13	DC3	Управление устройством 3
14	DC4	Управление устройством 4
15	NAK	Подтверждение ошибки передачи
16	SYN	Синхронизация
17	ETB	Конец передаваемого блока
18	CAN	Отмена
19	EM	Конец носителя
1A	SUB	Замена
1B	ESC	Выход или переход
1C	FS	Разделитель файлов
1D	GS	Разделитель групп
1E	RS	Разделитель записей
1F	US	Разделитель полей
20	SP	Пробел
7F	DEL	Удаление

равен 38h, а правой — E0h, 38h). Кроме уникального кода нажатия, каждая клавиша имеет свой код отпускания. Как правило, этот код состоит из двух байт, первый из которых всегда равен F0h. На расширенных клавиатурах коды отпускания имеют размер три байта, где первые два байта всегда равны E0h, F0h, а третий байт является последним байтом скан-кода нажатия. Существует три основных набора скан-кодов: стандартный XT (поддерживается некоторыми современными клавиатурами), набор по умолчанию (поддерживается всеми современными клавиатурами) и набор кодов PS/2 (необязательный и может не поддерживаться современными клавиатурами). В табл. 3.17 приводится описание второго набора скан-кодов. В любом случае, рекомендую читателям скачать в Интернете и распечатать для себя все скан-коды, поддерживаемые современными клавиатурами (особенно мультимедийными).

3.2. Использование портов

Клавиатура и мышь в современных компьютерах работают по протоколу PS/2 и оба подключаются к общему контроллеру клавиатуры (например, Intel 8042), интегрированному в чип (микросхему) на материнской плате. Для клавиатуры контроллер генерирует прерывание IRQ1. Контроллер может работать в двух режимах: PS/2-совместимом и AT-совместимом. Первый режим предусматривает поддержку двух устройств: мыши и клавиатуры. Доступ к контроллеру клавиатуры осуществляется через порты `60h` и `64h`. Порт `60h` в режиме чтения получает данные из буфера клавиатуры, а в режиме записи — помещает данные в буфер. Порт `64h` тоже работает в двух режимах: в режиме чтения он является регистром состояния, а в режиме записи используется как регистр команд. Формат регистра состояния (`64h`) представлен в табл. 3.4.

Таблица 3.4. *Регистр состояния (64h)*

Бит	Описание
0	Наличие данных в выходном буфере клавиатуры (0 — выходной буфер пустой, 1 — в буфере есть данные)
1	Наличие данных во входном буфере клавиатуры (0 — входной буфер пустой, 1 — в буфере есть данные)
2	Результат самотестирования (0 — сброс, 1 — тест прошел успешно)
3	Порт, используемый для последней операции (0 — `60h`, 1 — `64h`)
4	Состояние клавиатуры (0 — заблокирована, 1 — включена)
5	Ошибка передачи (0 — ошибок нет, 1 — клавиатура не отвечает)
6	Ошибка тайм-аута (0 — ошибка отсутствует, 1 — ошибка)
7	Ошибка четности (0 — ошибка отсутствует, 1 — ошибка) указывает на последнюю ошибку, произошедшую при передаче данных

Формат регистра команд (`64h`) показан в табл. 3.5.

Таблица 3.5. *Регистр команд (64h)*

Бит	Описание
0	Прерывания для клавиатуры (0 — отключить, 1 — включить)
1	Прерывания для мыши (0 — отключить, 1 — включить)
2	Системный флаг (1 — инициализация через самотестирование, 0 — инициализация по питанию)
3	Не используется

Таблица 3.5 *(окончание)*

Бит	Описание
4	Доступ к клавиатуре (0 — открыт, 1 — закрыт)
5	Доступ к мыши (0 — открыт, 1 — закрыт)
6	Трансляция скан-кодов (0 — не использовать, 1 — использовать)
7	Резерв

Перед началом работы с клавиатурой, следует проверить наличие данных в буфере (бит 0 в регистре состояния). Кроме того, такую проверку необходимо выполнять перед любыми последующими операциями записи. После проверки буфера в регистр `64h` записывается код желаемой команды (табл. 3.6). Если команда имеет дополнительные параметры, их следует записать в регистр данных (`60h`). После выполнения команды в регистр данных (`60h`) будет помещен результат (табл. 3.7).

Таблица 3.6. *Команды управления контроллером клавиатуры*

Код команды	Описание
`20h`	Прочитать байт из регистра команд
`60h`	Записать байт в регистр команд
`A1h`	Получить номер версии производителя
`A4h`	Получить пароль (возвратит `FAh`, если пароль существует, и `F1h` в обратном случае)
`A5h`	Установить пароль (посылает строку с нулевым символом в конце)
`A6h`	Проверить пароль (сравнивает введенный с клавиатуры пароль с текущим)
`AAh`	Выполнить самотестирование контроллера (в случае успеха возвратит `55h`)
`ABh`	Проверка интерфейса клавиатуры (`00h` — все хорошо, `01h` — низкий уровень сигнала синхронизации, `02h` — высокий уровень сигнала синхронизации, `03h` — низкий уровень сигнала на линии данных, `04h` — высокий уровень сигнала на линии данных)
`ADh`	Отключить интерфейс клавиатуры (устанавливает бит 4 в регистре команд)
`AEh`	Включить интерфейс клавиатуры (очищает бит 4 в регистре команд)
`AFh`	Получить версию
`C0h`	Прочитать входной порт
`D0h`	Прочитать значение из выходного порта
`D1h`	Записать параметр в выходной порт

Таблица 3.6 *(окончание)*

Код команды	Описание
D2h	Записать параметр в буфер клавиатуры
E0h	Тестирование порта (возвращает тестовое значение для порта)

Таблица 3.7. *Коды ошибок клавиатуры*

Код ошибки	Описание
00h	Ошибка переполнения (слишком много нажатий клавиш)
AAh	Самотестирование контроллера успешно завершено
EEh	Результат эхо-режима, инициированного командой EEh
FAh	Подтверждение успешного выполнения команды
FCh	Ошибка самотестирования клавиатуры
FEh	Запрос на повторную передачу данных (команда Resend)
FFh	Ошибка клавиатуры

Рассмотрим несколько примеров использования управляющих команд. Сначала напишем код, который выполняет проверку интерфейса (команда ABh) клавиатуры (листинг 3.1).

Листинг 3.1. Проверка наличия клавиатуры

```
DWORD dwResult  = 0; // переменная для хранения результата
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере клавиатуры
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
       if ( ( dwResult & 0x02 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду проверки интерфейса
outPort ( 0x64, 0xAB, 1 );
```

Теперь попробуем отключить клавиатуру. Для этого применим управляющую команду с кодом ADh так, как это сделано в листинге 3.2.

Листинг 3.2. Отключение клавиатуры

```
DWORD dwResult  = 0; // переменная для хранения результата
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере клавиатуры
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
       if ( (dwResult & 0x02 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду отключения клавиатуры
outPort ( 0x64, 0xAD, 1 );
```

Чтобы восстановить работу клавиатуры, запишите в регистр команд код AEh. Как видно из примеров, работа с командами управления не вызывает проблем. Кроме перечисленных ранее команд управления контроллером клавиатуры, существует еще ряд дополнительных команд, которые применяются для настройки различных режимов работы устройства и управления встроенным в клавиатуру процессором. Список этих команд представлен в табл. 3.8. Команды могут иметь дополнительные параметры, которые передаются в виде дополнительного байта параметра. Код команды следует записать в порт 60h. После посылки каждой команды следует проверять бит 1 для порта 64h и только при его сбросе (равенстве 0) передавать дополнительный параметр. Кроме того, можно проверять нулевой бит в порту 60h. После выполнения каждой команды, кроме EEh и FEh, возвращается подтверждающий код FAh (читается из порта 60h). Если значение кода команды или параметра недопустимы, клавиатура вернет код FEh. Если вместо дополнительного байта параметра послать код новой команды, клавиатура выполнит именно его, "забыв" о предыдущей команде.

Таблица 3.8. *Набор команд для клавиатуры*

Код команды	Описание
EDh	Установить или сбросить состояние индикаторов
EEh	Выполнить эхо-диагностику клавиатуры
F0h	Выбрать набор скан-кодов (первый, второй или третий)

Таблица 3.8 *(окончание)*

Код команды	Описание
F2h	Получить идентификатор клавиатуры
F3h	Настроить параметры автоповтора и время задержки
F4h	Включить клавиатуру (буфер будет очищен)
F5h	Отключить клавиатуру и загрузить значения по умолчанию
F6h	Установить значения параметров, используемых по умолчанию (10 символов/с, 500 мс)
F7h	Установить режим автоповтора для всех клавиш
F8h	Установить для всех клавиш передачу кодов нажатия и отпускания
F9h	Установить для всех клавиш передачу кодов нажатия
FAh	Установить для всех клавиш передачу кодов нажатия, отпускания и режим автоповтора
FBh	Установить для указанной клавиши передачу кодов нажатия и режим автоповтора
FCh	Установить для указанной клавиши передачу кодов нажатия и отпускания
FDh	Установить для указанной клавиши передачу кодов нажатия
FEh	Выполнить повторную передачу последнего переданного байта
FFh	Выполнить команду сброса клавиатуры

Некоторые команды требуют дополнительного описания, поэтому рассмотрим их поподробнее.

3.2.1. Команда *EDh*

Команда EDh позволяет управлять состоянием индикаторов на клавиатуре. Размер команды равен двум байтам. Первый байт содержит сам код команды, а второй содержит битовую маску (табл. 3.9) для настройки индикаторов.

Таблица 3.9. *Байт состояния индикаторов*

Бит	Описание
0	Индикатор Scroll Lock (1 — включен, 0 — выключен)
1	Индикатор Num Lock (1 — включен, 0 — выключен)
2	Индикатор Caps Lock (1 — включен, 0 — выключен)
3—7	Резерв

Приведу простой пример для управления индикатором клавиши <Num Lock> (листинг 3.3).

Листинг 3.3. Управление индикатором Num Lock

```
DWORD dwResult  = 0; // переменная для хранения результата
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере клавиатуры
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем в порт команду управления индикаторами
outPort ( 0x60, 0xED, 1 );
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере клавиатуры
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
outPort ( 0x60, 0x02, 1 );
```

3.2.2. Команда *EEh*

Команда `EEh` позволяет протестировать клавиатуру на предмет работоспособности. Если в работе клавиатуры возникли сбои, то сначала следует выполнить команду сброса клавиатуры (команда `FFh`), а затем эту команду. Возвращаемое значение, отличное от `EEh`, явно укажет на сбои в работе клавиатуры. Возможно придется даже отключить и повторно включить питание для восстановления нормальной работоспособности устройства.

3.2.3. Команда *F2h*

Команда `F2h` позволяет получить идентификатор клавиатуры и убедиться в ее наличии. После выполнения команды клавиатура вернет код подтверждения

FAh, а затем идентификатор: для большинства клавиатур это два кода — ABh и 83h (для отдельных клавиатур 41h). Рассмотрим простой пример получения идентификатора клавиатуры, описанный в листинге 3.4.

Листинг 3.4. Получение идентификатора клавиатуры

```
// для удобства работы пишем отдельную функцию опроса данных в буфере
bool WaitData ( )
{
      // переменная для хранения результата
      DWORD dwResult  = 0;
      int iTimeWait = 50000;
      // проверяем наличие данных во входном буфере клавиатуры
      while ( -- iTimeWait > 0 )
      {
          // читаем состояние порта
          inPort ( 0x64, &dwResult, 1 );
          if ( (dwResult & 0x01 ) == 0x00 ) return true;
          // закончилось время ожидания
          if ( iTimeWait < 1 ) return false;
      }
return false;
}
// основной код программы
DWORD dwResult  = 0;
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере клавиатуры
while ( -- iTimeWait > 0 )
{
       // читаем состояние порта
       inPort ( 0x64, &dwResult, 1 );
       if ( (dwResult & 0x02 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
outPort ( 0x60, 0xF2, 1 ); // записываем код команды
// ждем получения первого байта
iTimeWait = 50000;
if ( WaitData ( ) ) // получен
{
    inPort ( 0x60, &dwResult, 1 );
    // проверяем на равенство коду FAh
    if ( dwResult != 0xFA )
```

```
      return MY_ERROR;
}
else // нет данных
    return MY_ERROR;
// ждем получения второго байта
iTimeWait = 50000;
if ( WaitData ( ) ) // получен
{
    inPort ( 0x60, &dwResult, 1 );
    // проверяем на равенство коду ABh
    if ( dwResult != 0xAB )
      return MY_ERROR;
}
else // нет данных
    return MY_ERROR;
// ждем получения третьего байта
iTimeWait = 50000;
if ( WaitData ( ) ) // получен
{
    inPort ( 0x60, &dwResult, 1 );
    // проверяем на равенство коду 83h
    if ( dwResult != 0x83 )
      return MY_ERROR;
}
else // нет данных
    return MY_ERROR;
// идентификатор клавиатуры успешно получен
```

В данном примере мы послали команду `F2h` контроллеру клавиатуры и последовательно получили три байта: байт подтверждения и два байта идентификатора клавиатуры.

3.2.4. Команда *F3h*

Команда `F3h` позволяет установить частоту и время задержки для автоматического повтора набранного на клавиатуре символа. Размер команды равен двум байтам. Первый байт содержит сам код команды, а второй содержит описание параметров настройки автоповтора (табл. 3.10).

Коды времени задержки и коды значений частоты автоповтора представлены соответственно в табл. 3.11 и 3.12.

Рассмотрим простой пример использования команды `F3h`, который приведен в листинге 3.5.

Таблица 3.10. *Содержание байта настройки автоповтора*

Биты	Описание
0—4	Код значения частоты автоповтора
5—6	Код значения времени задержки
7	Резерв

Таблица 3.11. *Код задержки*

Значение кода	Время задержки, мс
00h	250
01h	500
02h	750
03h	1000

Таблица 3.12. *Код частоты повторений*

Значение кода	Частота, символов/с	Значение кода	Частота, символов/с
00h	30,0	10h	7,5
01h	26,7	11h	6,7
02h	24,0	12h	6,0
03h	21,8	13h	5,5
04h	20,0	14h	5,0
05h	18,5	15h	4,6
06h	17,1	16h	4,3
07h	16,0	17h	4,0
08h	15,0	18h	3,7
09h	13,2	19h	3,3
0Ah	12,0	1Ah	3,0
0Bh	10,9	1Bh	2,7
0Ch	10,0	1Ch	2,5
0Dh	9,2	1Dh	2,3
0Eh	8,6	1Eh	2,1
0Fh	8,0	1Fh	2,0

Листинг 3.5. Использование команды F3h

```
DWORD dwResult  = 0;
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере клавиатуры
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
outPort ( 0x60, 0xF3, 1 ); // записываем код команды
int iTimeWait = 50000;
// проверяем наличие данных во входном буфере клавиатуры
while ( -- iTimeWait > 0 )
 {
    // читаем состояние порта
    inPort ( 0x64, &dwResult, 1 );
    if ( (dwResult & 0x01 ) == 0x00 ) break;
    // закончилось время ожидания
    if ( iTimeWait < 1 ) return MY_ERROR_TIME;
 }
inPort ( 0x60, &dwResult, 1 );
// проверяем на равенство коду FAh
    if ( dwResult != 0xFA ) return MY_ERROR;
iTimeWait = 50000;
// проверяем наличие данных во входном буфере клавиатуры
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x64, &dwResult, 1 );
        if ( (dwResult & 0x02 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
// записываем значение частоты и время задержки
outPort ( 0x60, 0x28, 1 );
```

Остальные функции управления достаточно просты и не требуют дополнительных примеров. Для удобства работы приведу дополнительно скан-коды клавиш (второй набор), используемых во всех современных клавиатурах (табл. 3.13).

Таблица 3.13. *Второй набор скан-кодов клавиатуры*

Клавиша	Код нажатия	Код отпускания
<F1>	05	F0, 05
<F2>	06	F0, 06
<F3>	04	F0, 04
<F4>	0C	F0, 0C
<F5>	03	F0, 03
<F6>	0B	F0, 0B
<F7>	83	F0, 83
<F8>	0A	F0, 0A
<F9>	01	F0, 01
<F10>	09	F0, 09
<F11>	78	F0, 78
<F12>	07	F0, 07
<0>	45	F0, 45
<1>	16	F0, 16
<2>	1E	F0, 1E
<3>	26	F0, 26
<4>	25	F0, 25
<5>	2E	F0, 2E
<6>	36	F0, 36
<7>	3D	F0, 3D
<8>	3E	F0, 3E
<9>	46	F0, 46
<A>	1C	F0, 1C
<B>	32	F0, 32
<C>	21	F0, 21
<D>	23	F0, 23
<E>	24	F0, 24
<F>	2B	F0, 2B
<G>	34	F0, 34
<H>	33	F0, 33
<I>	43	F0, 43
<J>	3B	F0, 3B

Таблица 3.13 *(продолжение)*

Клавиша	Код нажатия	Код отпускания
<K>	42	F0, 42
<L>	4B	F0, 4B
<M>	3A	F0, 3A
<N>	31	F0, 31
<O>	44	F0, 44
<P>	4D	F0, 4D
<Q>	15	F0, 15
<R>	2D	F0, 2D
<S>	1B	F0, 1B
<T>	2C	F0, 2C
<U>	3C	F0, 3C
<V>	2A	F0, 2A
<W>	1D	F0, 1D
<X>	22	F0, 22
<Y>	35	F0, 35
<Z>	1A	F0, 1A
(IE) <Search>	E0, 10	E0, F0, 10
(IE) <Home>	E0, 3A	E0, F0 3A
(IE) <Stop>	E0, 28	E0, F0, 28
(IE) <Refresh>	E0, 20	E0, F0, 20
(IE) <Forward>	E0, 30	E0, F0, 30
(IE) <Back>	E0, 38	E0, F0, 38
(IE) <Favorites>	E0, 18	E0, F0, 18
<Volume Up>	E0, 32	E0, F0, 32
<Volume Down>	E0, 21	E0, F0, 21
<Mute>	E0, 23	E0, F0, 23
<Play>	E0, 34	E0, F0, 34
<Stop>	E0, 3B	E0, F0, 3B
<`>	0E	F0, 0E
<->	4E	F0, 4E
<=>	55	F0, 55

Таблица 3.13 *(продолжение)*

Клавиша	Код нажатия	Код отпускания
<Backspace>	66	F0, 66
<Tab>	0D	F0, 0D
<Space>	29	F0, 29
<Caps Lock>	58	F0, 58
<Esc>	76	F0, 76
<Enter>	5A	F0, 5A
<Left Ctrl>	14	F0, 14
<Right Ctrl>	E0, 14	E0, F0, 14
<Left Alt>	11	F0, 11
<Right Alt>	E0, 11	E0, F0, 11
<Left Shift>	12	F0, 12
<Right Shift>	59	F0, 59
<Left Win>	E0, 1F	E0, F0, 1F
<Right Win>	E0, 27	E0, F0, 27
<Apps>	E0, 2F	E0, F0, 2F
<Print Screen>	E0, 12, E0 7C	E0, F0, 7C E0, F0, 12
<Scroll Lock>	7E	F0, 7E
<Pause>	E1, 14, 77 E1, F0, 14 F0, 77	Нет
<[>	54	F0, 54
<]>	5B	F0, 5B
<;>	4C	F0, 4C
<|>	52	F0, 52
<,>	41	F0, 41
<.>	49	F0, 49
</>	4A	F0, 4A
<Insert>	E0, 70	E0, F0, 70
<Home>	E0, 6C	E0, F0, 6C
<Delete>	E0, 71	E0, F0, 71
<End>	E0, 69	E0, F0, 69
<Page Up>	E0, 7D	E0, F0, 7D
<Page Down>	E0, 7A	E0, F0, 7A

Таблица 3.13 *(окончание)*

Клавиша	Код нажатия	Код отпускания
<↑>	E0, 75	E0, F0, 75
<↓>	E0, 72	E0, F0, 72
<←>	E0, 6B	E0, F0, 6B
<→>	E0, 74	E0, F0, 74
<Num Lock>	77	F0, 77
(+) <Enter>	E0, 5A	E0, F0, 5A
(+) </>	E0, 4A	E0, F0, 4A
(+) <*>	7C	F0, 7C
(+) <->	7B	F0, 7B
(+) <+>	79	F0, 79
(+) <.>	71	F0, 71
(+) <0>	70	F0, 70
(+) <1>	69	F0, 69
(+) <2>	72	F0, 72
(+) <3>	7A	F0, 7A
(+) <4>	6B	F0, 6B
(+) <5>	73	F0, 73
(+) <6>	74	F0, 74
(+) <7>	6C	F0, 6C
(+) <8>	75	F0, 75
(+) <9>	7D	F0, 7D
<Next Track>	E0, 4D	E0, F0, 4D
<Previous Track>	E0, 15	E0, F0, 15
<Wake>	E0, 5E	E0, F0, 5E
<Power>	E0, 37	E0, F0, 37
<Sleep>	E0, 3F	E0, F0, 3F

Знаком плюс (+) в табл. 3.13 обозначены клавиши на дополнительной цифровой панели клавиатуры. Существует еще один порт — 61h, используемый в современных компьютерах для управления встроенным динамиком (спикером). Он служит для чтения и записи и позволяет управлять некоторыми функциями клавиатур (ради совместимости со старыми моделями). Формат регистра 61h представлен в табл. 3.14.

Таблица 3.14. *Формат регистра 61h*

Бит	Описание
0—5	Для клавиатуры не используются
6	Низкий уровень сигнала на линии синхронизации
7	Отключение клавиатуры (1 — выключить, 0 — включить)

Как видно из таблицы, для работы с клавиатурой отведены только два бита: 6 и 7. Остальные изменять не рекомендуется. Например, чтобы отключить клавиатуру, следует написать код, как показано в листинге 3.6.

Листинг 3.6. Включение и отключение клавиатуры с помощью портов

```
// пишем функцию управления клавиатурой
void LockUnlockKeyboard ( bool bLock )
{
   DWORD dwResult = 0;
   if ( bLock ) // блокировать
   {
     inPort ( 0x61, dwResult, 1) ;
     dwResult |= 0x80;
     outPort ( 0x61, dwResult, 1 );
   }
   else // разблокировать
   {
     inPort ( 0x61, dwResult, 1) ;
     dwResult &= 0x7F;
     outPort ( 0x61, dwResult, 1 );
   }
}
```

А теперь посмотрим, какие возможности по управлению клавиатурой предоставляет программный интерфейс Win32.

3.3. Использование Win32 API

Возможности интерфейса Win32 API "традиционно" скромны и ограниченны. Условно разделим описание всех возможностей на три части.

1. Настройка клавиатуры.
2. Использование "горячих" клавиш и акселераторов.
3. Поддержка различных языков.

Поддержка клавиатуры в операционных системах Windows осуществляется независимо от типа подключенного устройства и используемого языка. Любые нажатия клавиш (в виде скан-кодов) преобразуются системой (драйвером клавиатуры) и, в зависимости от установленного языка, отображаются в окне запущенного приложения. Каждому языку соответствует свой набор символов, а драйвер сам выполняет все необходимые преобразования. Программисту достаточно ознакомиться с универсальным набором скан-кодов так называемых *виртуальных клавиш* (основные перечислены в табл. 3.15). Каждая виртуальная клавиша имеет свое имя и числовое значение. В программах следует пользоваться по возможности именами клавиш, чтобы сохранить совместимость программ в дальнейшем.

Таблица 3.15. *Виртуальные клавиши*

Клавиша	Имя	Клавиша	Имя
<Tab>	VK_TAB	<Print Screen>	VK_SNAPSHOT
<Backspace>	VK_BACK	<Scroll Lock>	VK_SCROLL
<Enter>	VK_RETURN	<Pause>	VK_PAUSE
<Shift>	VK_SHIFT	<Insert>	VK_INSERT
<Alt>	VK_MENU	<Delete>	VK_DELETE
<Ctrl>	VK_CONTROL	<Home>	VK_HOME
<Caps Lock>	VK_CAPITAL	<End>	VK_END
<Esc>	VK_ESCAPE	<Page Up>	VK_PRIOR
<Space>	VK_SPACE	<Page Down>	VK_NEXT
<←>	VK_LEFT	<Left Win>	VK_LWIN
<→>	VK_RIGHT	<Right Win>	VK_RWIN
<↑>	VK_UP	<Apps>	VK_APPS
<↓>	VK_DOWN	<Sleep>	VK_SLEEP
(+) <0>	VK_NUMPAD0	<*>	VK_MULTIPLY
(+) <1>	VK_NUMPAD1	<+>	VK_ADD
(+) <2>	VK_NUMPAD2	<\>	VK_SEPARATOR
(+) <3>	VK_NUMPAD3	<->	VK_SUBTRACT
(+) <4>	VK_NUMPAD4	</> (деление)	VK_DIVIDE
(+) <5>	VK_NUMPAD5	<Num Lock>	VK_NUMLOCK
(+) <6>	VK_NUMPAD6	<Left Shift>	VK_LSHIFT
(+) <7>	VK_NUMPAD7	<Right Shift>	VK_RSHIFT
(+) <8>	VK_NUMPAD8	<Left Ctrl>	VK_LCONTROL
(+) <9>	VK_NUMPAD9	<Right Ctrl>	VK_RCONTROL

Таблица 3.15 *(окончание)*

Клавиша	Имя	Клавиша	Имя
<Left Alt>	VK_LMENU	<F6>	VK_F6
<Right Alt>	VK_LMENU	<F7>	VK_F7
<F1>	VK_F1	<F8>	VK_F8
<F2>	VK_F2	<F9>	VK_F9
<F3>	VK_F3	<F10>	VK_F10
<F4>	VK_F4	<F11>	VK_F11
<F5>	VK_F5	<F12>	VK_F12

Для буквенно-цифровых клавиш существуют только числовые коды, с которыми можно ознакомиться в документации фирмы Microsoft.

3.3.1. Настройка клавиатуры

Итак, первая возможность настройки клавиатуры, которую мы рассмотрим, относится к установке времени задержки для автоповтора. Для этого используется уже знакомая нам по *главе 2* функция `SystemParametersInfo` с параметром `SPI_SETKEYBOARDDELAY`. Второй аргумент функции должен указывать на время задержки. Поддерживаются четыре значения времени: 0 — 250 мс, 1 — 500 мс, 2 — 750 мс и 3 — 1000 мс. Существует также возможность получения текущего значения времени задержки, для чего служит параметр `SPI_GETKEYBOARDDELAY`. Пример кода, использующий обе эти возможности, представлен в листинге 3.7.

Листинг 3.7. Установка и получение времени задержки

```
// напишем функцию для установки времени задержки
bool SetDelay ( int iDelay )
{
    int iOldDelay = -1;
    // проверяем диапазон аргумента (должен быть от 0 до 3)
    if ( ( iDelay < 0 ) && ( iDelay > 3 ) ) return false;
    // получаем текущее значение времени задержки
    SystemParametersInfo ( SPI_GETKEYBOARDDELAY, 0, &iOldDelay, 0 );
    // если значение ошибочно или равно устанавливаемому, выходим
    if ( ( iOldDelay == -1 ) || ( iOldDelay == iDelay )
      return false;
```

```
    // устанавливаем новое значение времени задержки
    SystemParametersInfo ( SPI_SETKEYBOARDDELAY, iDelay, 0,
        SPIF_SENDCHANGE | SPIF_UPDATEINIFILE );
return true;
}
```

Кроме времени задержки, существует возможность изменить частоту автоповтора символов. Для этого следует передать в первый аргумент функции `SystemParametersInfo` параметр `SPI_SETKEYBOARDSPEED`, а во второй — новое значение частоты (от 0 до 31). Установка числа 0 соответствует 2,5 символам в секунду, а установка числа 31 — 30 символам в секунду. Остальные возможные значения лежат между ними (подробнее см. в табл. 3.12). Для получения текущего значения частоты автоповтора используется параметр `SPI_GETKEYBOARDSPEED`. В листинге 3.8 представлен пример функции, устанавливающей новое значение частоты автоповтора.

Листинг 3.8. Установка и получение частоты автоповтора

```
// напишем функцию для установки частоты автоповтора
bool SetFreq ( int iFreq )
{
    int iOldFreq = -1;
    // проверяем диапазон аргумента (должен быть от 0 до 31)
    if ( ( iFreq < 0 ) && ( iFreq > 31 ) ) return false;
    // получаем текущее значение частоты
    SystemParametersInfo ( SPI_GETKEYBOARDSPEED, 0, &iOldFreq, 0 );
    // если значение ошибочно или равно устанавливаемому, выходим
    if ( ( iOldFreq == -1 ) || ( iOldFreq == iFreq )
      return false;
    // устанавливаем новое значение частоты
    SystemParametersInfo ( SPI_SETKEYBOARDSPEED, iFreq, 0,
        SPIF_SENDCHANGE | SPIF_UPDATEINIFILE );
return true;
}
```

Как видите, настройка параметров автоповтора не представляет особых трудностей. Давайте теперь перейдем к еще одной удобной возможности управления клавиатурой. Она не имеет прямого отношения к устройству клавиатуры, но тесно связана с ней — это *частота мигания текстового курсора*. Для установки частоты мигания применяется функция `SetCaretBlinkTime`. Единственный аргумент функции задает промежуток времени (в миллисекундах) между двумя появлениями текстового курсора на экране. Существует еще функция `GetCaretBlinkTime`, позволяющая получить текущее значение време-

ни мигания курсора. Следует иметь в виду, что изменение частоты мигания курсора будет использоваться системой для всех приложений. В листинге 3.9 представлен пример функции, устанавливающей новое значение для частоты мигания курсора.

Листинг 3.9. Установка частоты мигания текстового курсора

```
// напишем функцию для установки частоты мигания курсора
bool SetFreqCaret ( int iFreq )
{
    int iOldFreq = 200;
    // проверяем диапазон аргумента (должен быть от 200 до 1200)
    if ( ( iFreq < 200 ) && ( iFreq > 1200 ) ) return false;
    // получаем текущее значение частоты
    iOldFreq = GetCaretBlinkTime ( );
    // если значение равно устанавливаемому, выходим
    if ( iOldFreq == iFreq )
      return false;
    // устанавливаем новое значение частоты
    if ( !SetCaretBlinkTime ( iFreq ) )
      return false; // произошла ошибка
return true;
}
```

При успешном завершении функция `SetCaretBlinkTime` должна возвратить ненулевое значение. Если функция вернула 0, значит, произошла ошибка, расширенное описание которой можно получить через вызов `GetLastError`. То же самое относится и к функции `GetCaretBlinkTime`.

Существует возможность программного отключения клавиатуры. Для этого применяется функция `BlockInput`, подробно рассмотренная в *главе 2.*

3.3.2. Использование "горячих" клавиш

Очень удобной возможностью работы в Windows является использование оперативных и "горячих" клавиш. Под *оперативными клавишами* понимаются комбинации клавиш для быстрого доступа к различным командам меню. Их еще называют *акселераторными клавишами*. Удобство их заключается в том, что к ним можно обращаться, когда меню закрыто и не активно. Работать с акселераторами достаточно просто. Вначале, используя редактор ресурсов, следует добавить в свою программу ресурс акселератора, присвоить ему идентификатор (например, `MY_ACCEL`) и назначить комбинации клавиш. Далее следует открыть в этом же редакторе ресурс меню и добавить к же-

лаемым пунктам описание закрепленной за ним клавиши (например, "&Открыть \t Ctrl+O"). После этого добавляем в стандартный код окна описание акселераторов, как показано в листинге 3.10.

Листинг 3.10. Добавление поддержки акселераторов в программу

```
// в функцию WinMain добавляем следующий код
// ...
// загружаем таблицу акселераторов с идентификатором MY_ACCEL
HACCEL hAccel = LoadAccelerators (hInst, MAKEINTRESOURCE ( MY_ACCEL ) );
// ...
// добавляем в цикл обработки сообщений перехватчик акселераторных клавиш
while ( GetMessage ( &msg, NULL, 0, 0 ) )
{
    if ( !TranslateAccelerator ( hWnd, hAccel, &msg ) )
      {
        TranslateMessage ( &msg );
        DispatchMessage ( &msg );
      }
}
```

Как видно из кода, для загрузки таблицы акселераторов используется функция `LoadAccelerators`. Данная функция имеет два аргумента. Первый аргумент указывает на дескриптор приложения, а второй — определяет имя ресурса для таблицы акселераторов. Можно вместо имени использовать идентификатор, обработав его макросом `MAKEINTRESOURCE`. После загрузки таблицы акселераторов необходимо установить перехватчик для всех оперативных клавиш. Роль перехватчика выполняет функция `TranslateAccelerator`. Первый аргумент функции указывает на дескриптор главного окна программы. Второй аргумент должен хранить дескриптор таблицы акселераторов (тип `HACCEL`). Третий аргумент указывает на структуру `MSG`. Вот и все премудрости. Теперь, выбор назначенной для пункта меню комбинации клавиш (например, <Ctrl>+<O>) вызовет выполнение соответствующей команды.

К сожалению, у акселераторов есть один существенный недостаток. Вы не сможете их использовать, если приложение потеряло фокус. Операционная система Windows автоматически выбирает таблицу акселераторов только для активного окна. Решить эту проблему можно с помощью "горячих" клавиш. Они представляют собой такую же комбинацию клавиш, как и акселераторы, но имеют в системе глобальный статус. Независимо от того, какая программа активна в данный момент, можно вызывать различные команды посредством "горячих" клавиш. Согласитесь, это очень удобно: ваша программа может "сидеть" в системном трее или вообще быть невидимой, но стоит на клавиа-

туре набрать заветную комбинацию и любая назначенная вами команда будет исполнена. Прежде чем рассказать, как программируются "горячие" клавиши, хочу выделить два важных правила:

1. Каждая "горячая" клавиша регистрируется в системе отдельно и должна иметь уникальный глобальный идентификатор.
2. После завершения работы с программой необходимо удалить зарегистрированные "горячие" клавиши из системы.

Для регистрации "горячей" клавиши используется функция `RegisterHotKey`. Функция имеет четыре аргумента. Первый указывает на дескриптор окна, которое будет обрабатывать специальное сообщение `WM_HOTKEY`. Второй аргумент указывает на идентификатор "горячей" клавиши, который назначается разработчиком. Это значение должно лежать в определенных пределах: для обычной программы — от `0x0000` до `0xBFFF`, для библиотеки DLL — от `0xC000` до `0xFFFF`. Чтобы избежать конфликтов с другими аналогичными значениями, для библиотек DLL следует использовать функцию `GlobalAddAtom`. Третий аргумент функции определяет модификатор для комбинации клавиш. Возможные значения для этого аргумента приведены в табл. 3.16. Можно комбинировать модификаторы между собой.

Таблица 3.16. *Типы модификатора клавиш*

Константа	Описание
`MOD_SHIFT`	Необходимо удерживать любую клавишу <Shift> на клавиатуре
`MOD_CONTROL`	Необходимо удерживать любую клавишу <Ctrl> на клавиатуре
`MOD_ALT`	Необходимо удерживать любую клавишу <Atl> на клавиатуре
`MOD_WIN`	Необходимо удерживать любую клавишу <Win> на клавиатуре

Четвертый аргумент функции задает код виртуальной клавиши (см. табл. 3.15). При успешном выполнении функции будет возвращено ненулевое значение. Если комбинация клавиш уже была зарегистрирована в системе другой программой, функция возвратит ошибку. В некоторых системах (например, Windows 2000) клавиша <F12> занята системой и ее не стоит использовать в качестве "горячей" клавиши. После регистрации всех необходимых программе "горячих" клавиш нужно добавить в функцию главного окна программы обработчик сообщения `WM_HOTKEY`. В нем будет сосредоточена вся работа по обработке зарегистрированных клавиш. После завершения работы с программой следует для каждой "горячей" клавиши вызвать функцию `UnregisterHotKey`, которая удалит регистрацию клавиши из системы. Чтобы закрепить полученные сведения, изучите листинг 3.11.

Листинг 3.11. Использование "горячих" клавиш в программе

```
// в начале файла описываем свои клавиши
#define MY_HOT_KEY_1   WM_USER + 5001
#define MY_HOT_KEY_2   WM_USER + 5002
#define MY_HOT_KEY_3   WM_USER + 5003
// в главную функцию окна добавляем поддержку клавиш
LRESULT CALLBACK MyMainProc ( HWND hWnd, UINT message,
                              UINT wParam, LONG lParam)
{
      switch (message)
        {
        case WM_CREATE: // во время создания окна регистрируем клавиши
          // для первой клавиши назначаем комбинацию <Ctrl>+<F2>
          RegisterHotKey ( hWnd, MY_HOT_KEY_1, MOD_CONTROL, VK_F2 );
          // для второй клавиши назначаем <Home>
          RegisterHotKey ( hWnd, MY_HOT_KEY_2, NULL, VK_HOME );
          // для третьей клавиши назначаем комбинацию <Ctrl>+<Shift>+<F5>
          RegisterHotKey ( hWnd, MY_HOT_KEY_3, MOD_CONTROL | MOD_SHIFT
                                                          , VK_F5 );
          break;
        case WM_HOTKEY: // обработка "горячих" клавиш
          {
           switch ( wParam ) // wParam содержит идентификатор
            {
             case MY_HOT_KEY_1:
             // выполняем какую-либо свою команду
               break;
             case MY_HOT_KEY_2:
             // выполняем какую-либо свою команду
               break;
             case MY_HOT_KEY_3:
             // выполняем какую-либо свою команду
               break;
            }
          }
          break;
        case WM_DESTROY: // завершаем программу
        // удаляем регистрацию клавиш
        UnregisterHotKey ( 0, MY_HOT_KEY_1 );
        UnregisterHotKey ( 0, MY_HOT_KEY_2 );
        UnregisterHotKey ( 0, MY_HOT_KEY_3 );
          break;
```

```
        default:
          return ( DefWindowProc (hWnd, message, wParam, lParam ) );
        }
return ( 0 );
}
```

3.3.3. Поддержка языков

Как правило, на обычном компьютере установлена поддержка не только русского, но и английского (или любого другого) или даже нескольких языков. Для переключения между ними применяется предопределенная комбинация клавиш. Мы можем из программы управлять выбором того или иного языка. За каждым поддерживаемым языком закреплен идентификационный номер. Для загрузки нового языка применяется функция `LoadKeyboardLayout`. Первый аргумент функции указывает на идентификатор языка (строка с шестнадцатеричным значением), который следует загрузить. Например, для английского языка идентификационный номер равен `0x0409`, где младшее слово указывает на идентификатор языка, а старшее — на дескриптор устройства (драйвера поддержки языка). В функцию записывается строковое значение "`00000409`". Для русского языка идентификационный номер будет равен "`00000419`" (`0x0419`). Второй аргумент функции определяет дополнительный флаг, значение которого может принимать одно из перечисленных в табл. 3.17.

Таблица 3.17. *Флаги для раскладки клавиатуры*

Значение флага	Описание
`KLF_REORDER`	Передвигает раскладку данного языка на первое место в списке
`KLF_REPLACELANG`	Если указанный идентификатор совпадает с уже имеющимся, то он все равно заменяет последний
`KLF_ACTIVATE`	Делает активной загруженную раскладку языка

При успешном выполнении функция возвратит идентификатор (тип `HKL`) загруженного языка. Для того чтобы выгрузить какой-либо язык, следует использовать функцию `UnloadKeyboardLayout`. Она имеет только один аргумент, указывающий на идентификатор языка. При успешном выполнении функция возвратит ненулевое значение. Дополнительно к описанным функциям можно применять `ActivateKeyboardLayout` для выбора текущего языка. Функция содержит два аргумента. Первый указывает на идентификатор языка, но может также принимать одно из следующих значений: `HKL_NEXT` — выбрать следующий язык из списка, `HKL_PREV` — выбрать предыдущий язык из списка. Второй аргумент определяет флаг операции. Это может быть одно из сле-

дующих значений: KLF_REORDER — переопределить список с учетом выбранного, KLF_RESET — сбросить блокировку использования заглавных букв (только для Windows 2000 и выше), KLF_SHIFTLOCK — включить блокировку использования заглавных букв (только для Windows 2000 и выше). В листинге 3.12 приводится пример работы с рассмотренными функциями.

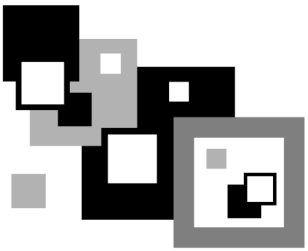
Листинг 3.12. Работа с раскладкой клавиатуры

```
// загружаем русскую раскладку клавиатуры
HKL hRus = LoadKeyboardLayout ( "00000419", KLF_REORDER );
// делаем русский язык текущим
if ( hRus != NULL )
ActivateKeyboardLayout ( hRus, KLF_REORDER );
// после работы с русской раскладкой выгружаем ее
UnloadKeyboardLayout ( hRus );
```

Существует также возможность получения информации о текущей раскладке клавиатуры. Для этого используются функции GetKeyboardLayout и GetKeyboardLayoutName. Первая функция возвращает числовой идентификатор (тип HKL) текущего языка, а вторая — строку с именем раскладки (например, "00000419").

На этом мне бы хотелось завершить описание различных способов программирования клавиатуры и перейти к рассмотрению не менее важного элемента любого компьютера, а именно видеоадаптера.

ГЛАВА 4



Видеоадаптер

Видеоадаптер является, пожалуй, одним из самых интересных устройств, входящих в состав современного компьютера. В тандеме с монитором он позволяет проникнуть в совершенно другой, виртуальный мир, расцвеченный невообразимыми красками и эффектами. Он служит своеобразным мостом между бездушными железками и ожившими образами. Вряд ли компьютер получил бы такое широкое признание людей, далеких от техники, если бы не его способность приоткрывать дверь в другой, так не похожий на наш, мир.

Это небольшое отступление я сделал не для того, чтобы повторить простые истины, а чтобы показать, насколько важное значение имеет видеоадаптер и монитор в общении простого пользователя с компьютером. Из сказанного выше разработчик программного обеспечения должен сделать один главный вывод — необходимо знать и уметь программировать возможности видеоадаптера и монитора. Не стоит всецело полагаться на готовый визуальный интерфейс Windows или библиотеку Direct X, многие задачи (например, создание динамичных игрушек) потребуют более широких знаний и умения работать с оборудованием.

В этой главе мы рассмотрим два варианта доступа к видеоадаптеру (видеоконтроллеру):

1. С использованием аппаратных портов.
2. С использованием возможностей Win32 API.

4.1. Общие сведения

Стандарты видеоадаптеров имеют достаточно большую историю. Первые монохромные адаптеры появились в начале 80-х годов прошлого века. Они поддерживали только два цвета (обычно зеленый на черном фоне) и работали

исключительно в текстовом режиме. В этом режиме на экран монитора с разрешением 80×25 (столбцов × строк) выводится стандартного размера прямоугольник (14 на 9 пикселов) с определенным символом. Далее появился стандарт — *CGA* (Color Graphics Adapter), позволяющий работать как в текстовом, так и в графическом режиме. В последнем режиме он поддерживал два разрешения: 320×200 (4 цвета) и 640×200 (2 цвета) пикселов. Были и другие стандарты (HGC, EGA), давно устаревшие и не используемые в современных видеоадаптерах.

Сегодня применяются современные *VGA*-совместимые (Video Graphics Array) видеоадаптеры, которые поддерживают большое количество текстовых и графических режимов: 4 текстовых, 256-цветный, HiColor (65536 цветов) и TrueColor (минимум 16,7 млн цветов). Для унификации управления режимами видеоадаптеров был разработан общий стандарт *VBE* (VESA BIOS EXTENSION). Он позволил определить одинаковые способы доступа к основным функциям многочисленных графических адаптеров различных фирм. Он может применяться для программирования любых VBE-совместимых видеоадаптеров. Прежде чем переходить к программированию, хочу сказать два слова о видеопамяти. Каждый видеоадаптер содержит определенный объем памяти, который применяется для промежуточного хранения данных, поступающих от компьютера. От размера видеопамяти напрямую зависят возможности самого адаптера: поддерживаемое разрешение и скорость работы. Существует простая формула, позволяющая подсчитать необходимый размер памяти для работы в том или ином разрешении. Для этого нужно перемножить значения разрешения по вертикали и горизонтали и умножить результат на количество байт в одном пикселе (16,7 млн — 3 байта, 65536 — 2 байта, 256 — 1 байт). Например, для поддержки разрешения 1024×768×16 размер памяти должен быть минимум 2 Мбайта. А теперь непосредственно приступим к изучению набора функций BIOS для программирования видеоадаптеров.

4.2. Использование портов

Перед тем, как рассказать о программировании портов видеоконтроллера, хочу сделать важное замечание: *будьте предельно ВНИМАТЕЛЬНЫ и ОСТОРОЖНЫ*. Случайно сделанная ошибка (возможны описки и в самой книге) может привести к выходу из строя дисплея вплоть до его возгорания. Перепроверяйте по несколько раз свои действия, а лучше советуйтесь со знающими специалистами. Для программирования портов видеоадаптера рекомендую использовать старый монитор, с которым не жалко будет расстаться. Несоблюдение этих простых правил может повлечь серьезные последствия, за которые читатель будет нести ответственность самостоятельно.

Стандартный VGA-совместимый видеоконтроллер поддерживает следующие типы регистров:

1. Внешние регистры.
2. Регистры графического контроллера.
3. Регистры контроллера атрибутов.
4. Регистры контроллера электронно-лучевой трубки — *CRT* (Cathode Ray Tube).
5. Регистры цифроаналогового преобразователя (ЦАП).
6. Регистры синхронизатора.

Список всех адресов регистров для стандартного видеоконтроллера показан в табл. 4.1. Рассмотрим работу с регистрами подробнее.

Таблица 4.1. *Адреса регистров видеоконтроллера*

Адрес регистра	Имя регистра
3B4h	CRTC Controller Address Register
3B5h	CRTC Controller Data Register
3BAh	Input Status #1 Register (чтение) или Feature Control Register (запись)
3C0h	Attribute Address/Data Register
3C1h	Attribute Data Read Register
3C2h	Input Status #0 Register (чтение) или Miscellaneous Output Register (запись)
3C4h	Sequencer Address Register
3C5h	Sequencer Data Register
3C7h	DAC State Register (чтение) или DAC Address Read Mode Register (запись)
3C8h	DAC Address Write Mode Register
3C9h	DAC Data Register
3CAh	Feature Control Register (чтение)
3CCh	Miscellaneous Output Register (чтение)
3CEh	Graphics Controller Address Register
3CFh	Graphics Controller Data Register
3D4h	CRTC Controller Address Register
3D5h	CRTC Controller Data Register
3DAh	Input Status #1 Register (чтение) или Feature Control Register (запись)

Следует заметить, что регистры 3B4h, 3B5h и 3BAh используются для работы в монохромном режиме, а 3D4h, 3D5h и 3DAh — в цветном.

4.2.1. Внешние регистры

Данные регистры позволяют управлять различными общими настройками, а также получать различную информацию о состоянии контроллера. В данную группу входят четыре основных регистра:

- Регистр общих настроек (Miscellaneous Output Register). Доступ: запись через порт 3C2h, чтение через порт 3CCh.
- Регистр особенностей (Feature Control Register). Доступ: запись через порт 3DAh (цветной режим) и 3BAh (монохромный режим), чтение через порт 3CAh.
- Первый регистр состояния (Input Status #0 Register). Доступен только для чтения через порт 3C2h.
- Второй регистр состояния (Input Status #1 Register). Доступен только для чтения через порт 3DAh (цветной режим) или 3BAh (монохромный режим).

4.2.1.1. Регистр общих настроек

Размер регистра равен 8 бит. Управляет частотой синхронизации, выбором страниц памяти, полярностью горизонтальных и вертикальных импульсов. Формат регистра представлен в табл. 4.2.

Таблица 4.2. *Формат регистра общих настроек*

Бит	Описание
0	Выбор адресов ввода-вывода для контроллера CRT
1	Доступ к видеопамяти
2—3	Частота синхронизации
4	Резерв
5	Выбор четной или нечетной страницы видеопамяти
6	Полярность горизонтальной синхронизации
7	Полярность вертикальной синхронизации

Описание таблицы.

- Бит 0 используется для установки адресов управления контроллера CRT (электронно-лучевой трубки). Если бит установлен в 1, то используются порты 3D4h и 3D5h. Если бит установлен в 0, то будут выбраны порты 3B4h и 3B5h.

- Бит 1 управляет доступом к видеопамяти устройства. Установка бита в 1 разрешает системе доступ к памяти видеоконтроллера. Установка бита в 0 запрещает доступ.
- Биты 2 и 3 управляют выбором частоты синхронизации. Значение 00h указывает на выбор частоты 25 МГц (или 25,175), а значение 01h — на частоту 28 МГц (или 28,322).
- Бит 4 зарезервирован и не используется.
- Бит 5 управляет выбором четной (значение равно 0) или нечетной (значение равно 1) страницы видеопамяти.
- Бит 6 управляет полярностью сигнала горизонтальной синхронизации. Значение 0 соответствует положительной полярности сигнала.
- Бит 7 управляет полярностью сигнала вертикальной синхронизации. Значение 0 соответствует положительной полярности сигнала. Может использоваться совместно с битом 6 для управления вертикальным размером экрана.

4.2.1.2. Регистр особенностей

Размер регистра равен 8 бит. Реально задействованы только 0 и 1 биты и оба являются служебными. Практическое применение этого регистра не документируется.

4.2.1.3. Первый регистр состояния

Размер регистра равен 8 бит. Бит 7 отвечает за прерывание обратного хода луча. Биты 5 и 6 указывают на присутствие дополнительных устройств. Бит 4 определяет наличие монитора. Данный регистр может иметь и другой формат, поэтому на него не стоит полагаться.

4.2.1.4. Второй регистр состояния

Размер регистра равен 8 бит. Позволяет получить информацию об обратном ходе луча. Формат регистра представлен в табл. 4.3.

Таблица 4.3. *Формат второго регистра состояния*

Бит	Описание
0	Обратный ход луча по горизонтали или вертикали
1—2	Резерв
3	Обратный ход луча по вертикали
4—7	Резерв

Описание таблицы.

- Бит 0 указывает на наличие обратного хода луча по горизонтали или вертикали. Значение 0 определяет обратный ход луча.
- Бит 3 указывает на наличие обратного хода луча только по вертикали. Если значение равно 1, то происходит обратный ход луча.

Рассмотрим примеры работы с внешними регистрами. Попробуем установить частоту синхронизации 25 МГц (листинг 4.1).

Листинг 4.1. Установка новой частоты синхронизации

```
DWORD dwResult  = 0; // переменная для хранения результата
// читаем байт из порта 03CCh
inPort ( 0x03CC, &dwResult, 1 );
// устанавливаем частоту 25 МГц
dwResult &= 0x0C;
// записываем значение в порт 03C2h
outPort (0x03C2, dwResult, 1 );
```

Рассмотрим еще один пример для определения момента обратного хода луча по вертикали (листинг 4.2).

Листинг 4.2. Определение момента обратного хода луча по вертикали

```
DWORD dwResult  = 0; // переменная для хранения результата
// читаем байт из порта 03DAh
int iTimeWait = 1000;
// ждем окончание обратного хода луча
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x03DA, &dwResult, 1 );
        if ( (dwResult & 0x08 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return MY_ERROR_TIME;
}
int iTimeWait = 1000;
// ждем начала следующего луча
while ( -- iTimeWait > 0 )
{
        // читаем состояние порта
        inPort ( 0x03DA, &dwResult, 1 );
        if ( (dwResult & 0x08 ) == 0x00 ) break;
```

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

4.2.2. Регистры графического контроллера

Данные регистры управляют доступом процессора к видеопамяти. Для указания адреса используется порт 3CEh, а для передачи данных — порт 3CFh. Графический контроллер поддерживает девять индексных регистров (от 0h до 8h). Каждый индексный регистр отвечает за определенные настройки. Список индексных регистров показан в табл. 4.4.

Таблица 4.4. *Список индексных регистров графического контроллера*

Код индексного регистра	Описание
00h	Регистр установки/сброса
01h	Регистр разрешения для установки/сброса
02h	Регистр сравнения цвета
03h	Регистр управления циклическим сдвигом данных
04h	Регистр выбора плоскости чтения
05h	Регистр управления режимом работы
06h	Графический регистр общего назначения
07h	Регистр сброса цветовых плоскостей
08h	Регистр битовой маски

Принцип работы с графическим контроллером следующий: в порт адреса 3CEh следует записать код индексного регистра. После этого можно передавать или получать данные через порт данных 3CFh. Для выбора другого индексного регистра необходимо опять записать его код в порт 3CEh. Рассмотрим индексные регистры подробнее.

4.2.2.1 Регистр *00h*

Размер регистра равен 8 бит. Данный регистр управляет цветовыми плоскостями, расположенными в видеопамяти. Биты с 0 по 3 представляют собой четыре плоскости. Эти биты используются режимами записи (см. регистр 05h) под номером 0 и 3.

4.2.2.2. Регистр *01h*

Размер регистра равен 8 бит. Этот регистр позволяет включать или отключать использование регистра 00h при работе с выбранной плоскостью. Биты с 0

по 3 представляют собой четыре плоскости в видеопамяти. Они используются в нулевом режиме записи и позволяют выбирать тип операции: получать ли данные для каждой плоскости от системы (процессора) или от установки соответствующего бита в индексном регистре 00h.

4.2.2.3. Регистр *02h*

Размер регистра равен 8 бит. Данный режим сравнивает выбранный цвет в регистре с цветом в памяти и возвращает результат сравнения. Биты с 0 по 3 представляют собой четыре плоскости, расположенные в видеопамяти. Процесс сравнения начинается с чтения (только в режиме чтения 1) значения цвета, после чего считывается значение цвета из памяти. В результате если биты в памяти и в регистре совпадают, то результирующий бит равен 1, иначе 0.

4.2.2.4. Регистр *03h*

Размер регистра равен 8 бит. Предназначен для выбора различных логических операций, применяемых к передаваемым от процессора данным, а также позволяет выполнять циклический сдвиг вправо передаваемых данных. Логическая операция выполняется над данными и содержимым регистра-защелки. Формат регистра показан в табл. 4.5.

Таблица 4.5. *Формат индексного регистра 03h*

Бит	Описание
0—2	Индекс сдвига
3—4	Тип логической операции
4—7	Резерв

Описание таблицы.

- Биты 0—2 определяют целое значение, на которое будет выполнена логическая операция сдвига вправо. Используется для режимов записи 0 и 3.
- Биты 3 и 4 указывают на тип логической операции и могут принимать одно из следующих значений:
  - 00h — данные передаются без изменений;
  - 01h — данные преобразуются с помощью операции AND;
  - 02h — данные преобразуются с помощью операции OR;
  - 03h — данные преобразуются с помощью операции XOR. Данное поле применяется для режимов записи под номерами 0 и 2.

4.2.2.5. Регистр *04h*

Размер регистра равен 8 бит. Предназначен для выбора плоскости в видеопамяти, которая будет использована в режиме чтения (номер 0) для получения данных. Задействованы только 0 и 1 биты, с помощью которых выбирается номер плоскости (0—3).

4.2.2.6. Регистр *05h*

Размер регистра равен 8 бит. Предназначен для установки различных режимов работы. Формат регистра представлен в табл. 4.6.

Таблица 4.6. *Формат индексного регистра 05h*

Бит	Описание
0—1	Номер режима записи
2	Резерв
3	Номер режима чтения
4	Адресация плоскостей
5	Режим чередования
6	Режим цвета
7	Резерв

Описание таблицы.

- Биты 0 и 1 позволяют выбрать режим записи. Существует четыре режима записи (от 0 до 3):
  - *режим 0*. Данные сначала сдвигаются в соответствии со значением индекса сдвига (регистр `03h`), далее выполняется операция установки/сброса (регистр `00h`), а затем указанная логическая операция (регистр `03h`). После этого данные помещаются в регистр-защелку и там с помощью битовой маски (регистр `08h`) выбираются нужные биты, а затем окончательные битовые слои записываются в видеопамять в зависимости от установки индексного регистра `02h` синхронизатора;
  - *режим 1*. Данные передаются непосредственно из 32-битного регистра-защелки в видеопамять и зависят только от установки индексного регистра `02h` синхронизатора;
  - *режим 2*. Биты 0—3 записываются в соответствующие плоскости видеопамяти. После этого в регистре-защелке выполняется указанная логическая операция, и данные обрабатываются с учетом битовой маски

(регистр 08h), а затем окончательные битовые слои записываются в видеопамять в зависимости от установки индексного регистра 02h синхронизатора;

- *режим 3*. Данные обрабатываются, как если бы биты 0—3 регистра 00h были установлены в 1 (1111b). После этого выполняется операция сдвига в соответствии со значением индекса сдвига (регистр 03h), а затем используется логическая операция AND. Далее данные обрабатываются с учетом битовой маски (регистр 08h), а затем окончательные битовые слои записываются в видеопамять в зависимости от установки индексного регистра 02h синхронизатора.

- Бит 3 определяет один из возможных режимов чтения: 0 или 1. В режиме чтения 0 байт считывается с одной из четырех плоскостей (0—3). Номер плоскости выбирается в регистре 04h. В режиме чтения 1 выполняется сравнение цвета в видеопамяти и цвета, указанного в регистре 02h. Битовые плоскости, игнорируемые в регистре 07h, не сравниваются. В результате сравнения возвращается бит сравнения между цветом текущим и устанавливаемым.

- Бит 4, установленный в 1, позволит выбирать нечетные адреса плоскостей (1 или 3) для передаваемых от процессора данных, использующих нечетные адреса.

- Бит 5, установленный в 1, позволит регистрам графического контроллера обрабатывать поток данных последовательно-параллельным способом: нечетные биты записываются в нечетные плоскости или четные биты записываются в четные битовые плоскости.

- Бит 6, установленный в 1, позволит включить поддержку 256-цветного режима. Установка бита в 0 позволит использовать только 16 цветов.

4.2.2.7. Регистр *06h*

Размер регистра равен 8 бит. Предназначен для установки различных параметров. Формат регистра представлен в табл. 4.7.

Таблица 4.7. *Формат индексного регистра 06h*

Бит	Описание
0	Управление режимом
1	Управление четностью
2—3	Выбор диапазона видеопамяти

Описание таблицы.

- Бит 0, установленный в 1, включает графический режим, а запись 0 переводит видеоадаптер в текстовый режим.
- Бит 1, установленный в 1, позволит управлять четностью. При этом нечетные адреса будут записаны в нечетную плоскость, а четные — в четную плоскость памяти.
- Биты 2—3 позволяют установить доступный видеоадаптеру диапазон памяти. Поддерживаются следующие значения: `00h` — `A0000h–BFFFFh` (128 Кбайт), `01h` — `A0000h—AFFFFh` (64 Кбайта), `02h` — `B0000h—B7FFFh` (32 Кбайта) и `03h` — `B8000h—BFFFFh` (32 Кбайта).

4.2.2.8. Регистр *07h*

Размер регистра равен 8 бит. Позволяет игнорировать указанные цветовые плоскости. Для выбора нужной плоскости применяются биты 0—3. Данный регистр используется для режима чтения 1. Установка любого бита указывает на то, что соответствующая плоскость будет рассмотрена при сравнении. Установка бита в 0 позволяет режиму чтения игнорировать ее.

4.2.2.9. Регистр *08h*

Размер регистра равен 8 бит. Данный регистр (все 8 бит) используется в режимах записи 0, 2 и 3. Позволяет включить или отключить возможность изменения одного или всех битов в байте адресуемых данных. Значение регистра влияет на все четыре плоскости. Если бит в регистре установлен в 1, значит, будет выбрано предыдущее значение бита. Если бит равен 0, будет использовано значение соответствующего бита из регистра-защелки.

Рассмотрим примеры работы с регистрами графического контроллера. Сначала получим значение диапазона памяти, используемой видеоконтроллером (листинг 4.3).

Листинг 4.3. Определение адреса выделенной памяти

```
DWORD dwResult  = 0; // переменная для хранения результата
// записываем значение регистра 06h в порт 03CEh
outPort (0x03CE, 0x06, 1 );
// читаем байт из порта 03CFh
inPort ( 0x03CF, &dwResult, 1 );
// выделяем биты 2 и 3
dwResult &= 0x0C;
// сравниваем результат
switch ( dwResult )
```

```
{
    case 0: // A0000h-BFFFFh (128 Кбайт)
     break;
    case 1: // A0000h-AFFFFh (64 Кбайта)
     break;
    case 2: // B0000h-B7FFFh (32 Кбайта)
     break;
    case 3: // B8000h-BFFFFh (32 Кбайта)
     break;
}
```

Рассмотрим еще один пример, использующий регистр `04h` для выбора плоскости под номером 1 (листинг 4.4).

Листинг 4.4. Выбор плоскости в видеопамяти

```
DWORD dwResult  = 0; // переменная для хранения результата
// записываем значение регистра 04h в порт 03CEh
outPort (0x03CE, 0x04, 1 );
// записываем плоскость номер 1 в порт 03CFh
outPort (0x03CF, 0x01, 1 );
```

Далее в главе будет использоваться код с явным указанием всех портов, что делается исключительно для наглядности материала, но в своих программах читатель вправе выбирать наиболее удобный для себя вариант.

4.2.3. Регистры контроллера атрибутов

Данные регистры управляют выбором палитры, цветом обрамляющей экран рамки, преобразованием байта атрибута в данные о цвете символа и фона, а также режимом прокрутки экрана. Для указания адреса используется порт `3C0h`. Для передачи данных применяется этот же порт `3C0h`. Для определения текущего режима порта `3C0h` (адрес индекса или передача данных) следует прочитать порт `3DAh`. Кроме того, можно ориентироваться на цикл работы данного порта: первое обращение к порту указывает на номер индексного регистра, а второе обращение — на передачу данных. При третьем обращении порт `3C0h` ожидает опять номера индексного регистра. И так далее. Для считывания данных может применяться порт `3C1h`. Формат регистра `3C0h` показан в табл. 4.8.

Описание таблицы.

- Биты 0—4

  В это поле следует записать номер индексного регистра, для которого должна быть выполнена операция чтения или записи.

Таблица 4.8. *Формат регистра 3C0h*

Бит	Описание
0—4	Номер индексного регистра
5	Использование палитры
6—7	Резерв

- Бит 5

  Установка этого бита в 1 закрывает доступ к внутренней палитре. Если бит установлен в 0, то будет выполнена загрузка значений цветов во внутренние регистры палитры.

Контроллер атрибутов поддерживает 20 индексных регистров (от `00h` до `14h`). Каждый индексный регистр отвечает за определенные настройки. Список индексных регистров приведен в табл. 4.9.

Таблица 4.9. *Список индексных регистров контроллера атрибутов*

Код индексного регистра	Описание
`00h—0Fh`	Регистры палитры
`10h`	Регистр управления режимом
`11h`	Регистр управления цветом рамки
`12h`	Регистр управления цветовыми плоскостями
`13h`	Регистр горизонтального панорамирования в единицах пиксела
`14h`	Регистр выбора цвета

Рассмотрим подробнее индексные регистры контроллера атрибутов.

4.2.3.1. Регистры палитры (*00h-0Fh*)

Размер каждого регистра равен 8 бит. Регистры палитры предназначены для динамического преобразования атрибута текстового символа или значения цвета (в графическом режиме) в реальный код цвета на экране. Используются только биты с 0 по 5. При установке бита в 1 будет выбран соответствующий цвет. Изменять значения внутренних регистров палитры можно только во время обратного хода луча по вертикали, иначе возникнут проблемы с выводимым на экран изображением.

4.2.3.2. Регистр *10h*

Размер регистра равен 8 бит. Предназначен для установки различных режимов работы. Формат регистра показан в табл. 4.10.

Таблица 4.10. *Формат регистра управления режимом (10h)*

Бит	Описание
0	Управление графическим режимом
1	Эмуляция монохромного дисплея
2	Использование псевдографики
3	Управление миганием символа
4	Резерв
5	Режим поэлементного панорамирования
6	Управление цветом
7	Управление регистрами палитры

Описание таблицы.

- Бит 0, установленный в 1, позволяет выбрать графический режим работы.
- Бит 1, установленный в 1, позволяет выбрать монохромный режим работы дисплея. Если бит установлен в 0, то будет использоваться цветной режим.
- Бит 2 применяется в 9-разрядных символьных режимах для того, чтобы правильно отобразить непрерывную строку символов. При установке бита в 1 каждый 9-й столбец будет закрашен фоновым цветом остальных символов.
- Бит 3, установленный в 1, разрешает мигание символа. Если этот бит равен 0, то используется насыщенность цвета фона символа (7 бит байта атрибута).
- Бит 5, установленный в 1, разрешает использование режима поэлементного панорамирования.
- Бит 6, установленный в 1, позволяет выбрать 8-битный цвет (256 цветов). Во всех остальных режимах этот бит должен быть установлен в 0.
- Бит 7, установленный в 1, позволяет использовать биты 4 и 5 регистра палитры вместо битов 0 и 1 регистра выбора цвета (`14h`).

4.2.3.3. Регистр *11h*

Размер регистра равен 8 бит. Предназначен для установки цвета рамки, обрамляющей экран. Используются все биты. Для установки цвета следует записать в этот регистр номер одного из регистров ЦАП (00h—FFh).

4.2.3.4. Регистр *12h*

Размер регистра равен 8 бит. Позволяет устанавливать доступ к определенным цветовым плоскостям. Используются биты с 0 по 3. Установка в 1 какого-либо из этих битов позволит открыть доступ к плоскости памяти, имеющей соответствующий номер (от 0 до 3).

4.2.3.5. Регистр *13h*

Размер регистра равен 8 бит. Позволяет управлять горизонтальным панорамированием (сдвигом по горизонтали вправо). Используется битовое поле с 0 по 3. Для указания нового значения следует записать в это поле целое число, измеряемое в пикселах. Сдвиг может применяться в текстовых и графических режимах. Запись в регистр нужно производить в момент обратного хода луча по вертикали.

4.2.3.6. Регистр *14h*

Размер регистра равен 8 бит. Позволяет расширить 6-разрядные значения цвета до 8-разрядных, а также заместить биты 4 и 5 регистров палитры. Формат регистра показан в табл. 4.11.

Таблица 4.11. *Формат регистра управления режимом (14h)*

Бит	Описание
0—1	Замещение 4 и 5 битов регистров палитры
2—3	Добавление 6 и 7 битов описания цвета
4—7	Резерв

Описание таблицы.

- Биты 0 и 1 управляют замещением 4 и 5 битов регистра палитры. Если в регистре управления режимом (10h) бит 7 установлен в 1, то 4 и 5 биты во всех регистрах палитры будут замещены 0 и 1 битом регистра выбора цвета.
- Биты 2 и 3 позволяют добавить два старших бита (6 и 7) к 5-разрядному представлению цвета для согласования с регистром цвета ЦАП. Этот режим может использоваться для быстрого переключения цветов ЦАП.

Рассмотрим пример работы с регистрами контроллера атрибутов. Попробуем установить графический режим работы с помощью регистра управления режимом (10h), как показано в листинге 4.5.

Листинг 4.5. Установка графического режима работы

```
// переменная для хранения результата
DWORD dwResult  = 0;
// записываем значение регистра 10h в порт 03C0h
outPort (0x03C0, 0x10, 1 );
// читаем из порта 3C0h
inPort ( 0x03C0, &dwResult, 1 );
// устанавливаем графический режим
dwResult &= 0x01;
// записываем значение регистра 10h в порт 03C0h
outPort (0x03C0, 0x10, 1 );
// записываем новое значение
outPort (0x03C0, dwResult, 1 );
```

4.2.4. Регистры контроллера CRT

Контроллер электронно-лучевой трубки управляет кадровой разверткой (формированием кадров на дисплее), а также параметрами горизонтальной развертки. Для доступа к контроллеру CRT используются два основных порта: выбора адреса (порт 3D4h) и ввода-вывода данных (порт 3D5h). Кроме этого, контроллер поддерживает 25 индексных регистров, управляющих различными функциями контроллера CRT. Полный список этих регистров показан в табл. 4.12. Для записи или чтения данных следует записать номер индексного регистра в порт 3D4h, а затем передать или получить данные через порт 3D5h.

Таблица 4.12. *Список индексных регистров контроллера CRT*

Код индексного регистра	Описание
00h	Регистр полной длины горизонтальной развертки
01h	Регистр длины горизонтальной развертки
02h	Начало хода луча по горизонтали
03h	Конец хода луча по горизонтали
04h	Начало обратного хода луча по горизонтали
05h	Конец обратного хода луча по горизонтали
06h	Количество линий растра экрана или длина вертикального хода луча

Таблица 4.12 *(окончание)*

Код индексного регистра	Описание
07h	Регистр переполнения
08h	Регистр предварительной развертки по горизонтали
09h	Максимальная высота строки развертки
0Ah	Регистр начальной позиции курсора
0Bh	Регистр конечной позиции курсора
0Ch	Старший байт в начальном адресе
0Dh	Младший байт в начальном адресе
0Eh	Старший байт в позиции курсора
0Fh	Младший байт в позиции курсора
10h	Начало обратного хода луча по вертикали
11h	Конец обратного хода луча по вертикали
12h	Количество линий в растре
13h	Регистр смещения
14h	Позиция символа подчеркивания
15h	Начало хода луча по вертикали
16h	Конец хода луча по вертикали
17h	Регистр управления режимом
18h	Регистр сравнения строк

Работать с регистрами CRT следует очень осторожно, чтобы не повредить монитор. Для защиты регистров 00h—07h от записи в регистре 11h бит 7 всегда установлен в 1. Чтобы восстановить возможность записи, следует обнулить этот бит.

4.2.4.1. Регистр *00h*

Размер регистра равен 8 бит. Предназначен для определения полного размера горизонтальной развертки. Позволяет управлять частотой обновления экрана по горизонтали, устанавливая необходимое для этого количество времени.

4.2.4.2. Регистр *01h*

Размер регистра равен 8 бит. Предназначен для управления длиной горизонтальной развертки, отображаемой на экране. Данный регистр устанавливает

порядок вывода на экран значений пикселов из памяти дисплея. Начало вывода устанавливается количеством кодовых комбинаций активного дисплея минус один.

4.2.4.3. Регистр *02h*

Размер регистра равен 8 бит. Предназначен для управления началом момента гашения луча по горизонтали. Данный регистр используется совместно с регистром `03h` для получения точки гашения луча по горизонтали.

4.2.4.4. Регистр *03h*

Размер регистра равен 8 бит. Предназначен для управления окончательным моментом гашения луча по горизонтали. Формат регистра показан в табл. 4.13.

Таблица 4.13. *Формат регистра 03h*

Бит	Описание
0—4	Указывает на конец гашения луча
5—6	Определяет количество импульсов задержки (как правило, равно 0)
7	Используется для поддержки светового пера (оставлено ради совместимости)

4.2.4.5. Регистр *04h*

Размер регистра равен 8 бит. Позволяет управлять началом обратного хода луча по горизонтали. Значение этого регистра определяет временной интервал, через который начнется передача синхронизирующих импульсов дисплею. Используется совместно с регистром `05h`.

4.2.4.6. Регистр *05h*

Размер регистра равен 8 бит. Предназначен для управления окончательным моментом гашения луча по горизонтали. Формат регистра показан в табл. 4.14.

Таблица 4.14. *Формат регистра 05h*

Бит	Описание
0—4	Указывает на конец обратного хода луча
5—6	Определяет время задержки (как правило, равно 0)
7	Содержит 5 бит дополнительно к битам 0—4

4.2.4.7. Регистр *06h*

Размер регистра 10 битов. Старшие два бита (8 и 9) располагаются в регистре переполнения (07h). Определяет количество вертикальных линий растра для активного экрана, иначе говоря, управляет значением длины обратного хода луча по вертикали.

4.2.4.8. Регистр *07h*

Размер регистра равен 8 бит. Предоставляет дополнительные биты для различных управляющих регистров. Формат данного регистра показан в табл. 4.15.

Таблица 4.15. *Формат регистра 07h*

Бит	Описание
0	Бит 8 для регистра 06h
1	Бит 8 для регистра 12h
2	Бит 8 для регистра 10h
3	Бит 8 для регистра 15h
4	Бит 8 для регистра 18h
5	Бит 9 для регистра 06h
6	Бит 9 для регистра 12h
7	Бит 9 для регистра 10h

4.2.4.9. Регистр *08h*

Размер регистра равен 8 бит. Позволяет настроить горизонтальную развертку. Кроме того, данный регистр позволяет в текстовом режиме организовать плавную прокрутку экрана. Формат регистра представлен в табл. 4.16.

Таблица 4.16. *Формат регистра 08h*

Бит	Описание
0—4	Количество линий вертикальной развертки
5—6	Управление позицией левого верхнего угла в видеопамяти
7	Резерв

Описание таблицы.

- Биты 0—4 содержат количество линий горизонтальной развертки, на которое экран должен быть прокручен вверх. Возможные значения должны лежать в диапазоне от 0 до максимальной высоты строки развертки (*см. разд. 4.2.4.10*).
- Биты 5—6 используются совместно с регистрами начального адреса для получения координат левого верхнего пиксела или символа на экране.

4.2.4.10. Регистр *09h*

Размер регистра равен 8 бит. Предназначен для управления высотой строки развертки. Кроме этого, регистр содержит дополнительные биты для некоторых других регистров. Формат данного регистра показан в табл. 4.17.

Таблица 4.17. *Формат регистра 09h*

Бит	Описание
0—4	Максимальная высота строки развертки
5	Бит 9 для регистра `15h`
6	Бит 9 для регистра `18h`
7	Режим двойного сканирования

Описание таблицы.

- Биты 0—4 определяют максимальное значение высоты одной строки развертки, иначе говоря, высоту символов на экране дисплея.
- Бит 5 определяет дополнительный старший разряд (бит 9) для регистра `15h`.
- Бит 6 определяет дополнительный старший разряд (бит 9) для регистра `18h`.
- Бит 7 управляет режимом двойного сканирования. При установке этого бита в 1 будет использоваться 400 линий. Если бит установлен в 0, то 200 линий.

4.2.4.11. Регистр *0Ah*

Размер регистра равен 8 бит. Предназначен для управления начальной позицией курсора на экране. Формат регистра показан в табл. 4.18. Используется для текстового режима.

Таблица 4.18. *Формат регистра 0Ah*

Бит	Описание
0—4	Начальная позиция линии курсора
5	Управление курсором
6—7	Резерв

Описание таблицы.

- Биты 0—4 содержат значение позиции для линии развертки, на которой будет расположен курсор.
- Бит 5 управляет отображением курсора в текстовом режиме. Установка этого бита в 1 скрывает курсор с экрана.

4.2.4.12. Регистр *0Bh*

Размер регистра равен 8 бит. Предназначен для управления конечной позицией курсора на экране. Формат регистра показан в табл. 4.19. Используется для текстового режима.

Таблица 4.19. *Формат регистра 0Bh*

Бит	Описание
0—4	Конечная позиция линии курсора
5—6	Смещение курсора
7	Резерв

Описание таблицы.

- Биты 0—4 содержат значение позиции для линии развертки, на которой будет расположен курсор.
- Биты 5—6 используются в режиме EGA для синхронизации курсора с работой дисплея. В режиме VGA могут быть установлены в ноль или дополнять биты 0—4.

4.2.4.13. Регистр *0Ch*

Размер регистра равен 8 бит. Содержит значение старшего байта (биты 8—15) начального адреса в видеопамяти, используемого для вывода информации на экран. Проще говоря, данный регистр содержит старшую часть адреса, которая определяет начальную позицию (левый верхний угол) на экране.

4.2.4.14. Регистр *0Dh*

Размер регистра равен 8 бит. Содержит значение младшего байта (биты 0—7) начального адреса в видеопамяти, используемого для вывода информации на экран. Совместно с регистром 0Ch определяет полный 16-битный адрес начальной позиции (левый верхний угол) на экране. Поскольку размер адресуемой памяти ограничен (только 16 разрядов), данные регистры могут применяться только для текстовых и некоторых графических режимах.

4.2.4.15. Регистр *0Eh*

Размер регистра равен 8 бит. Содержит значение старшего байта (биты 8—15) позиции курсора на экране.

4.2.4.16. Регистр *0Fh*

Размер регистра равен 8 бит. Содержит значение младшего байта (биты 0—7) позиции курсора на экране. Совместно с регистром 0Eh определяет позицию курсора на экране, относительно левого верхнего угла. Оба регистра используются в текстовых режимах.

4.2.4.17. Регистр *10h*

Размер регистра равен 10 битов. Два старших бита (8 и 9) расположены в регистре переполнения (07h). Регистр предназначен для управления начальным значением обратного хода луча по вертикали.

4.2.4.18. Регистр *11h*

Размер регистра равен 8 бит. Предназначен для управления конечным значением обратного хода луча по вертикали. Кроме этого, позволяет блокировать запись в регистры 00h—07h. Формат регистра показан в табл. 4.20.

Таблица 4.20. *Формат регистра 11h*

Бит	Описание
0—3	Обратный ход луча по вертикали
4—5	Резерв
6	Обновление памяти
7	Блокировка регистров 00h—07h

Описание таблицы.

- Биты 0—3 определяют значение обратного хода луча по вертикали.
- Бит 6 управляет количеством циклов обновлений (регенерации) динамической памяти за время вертикального хода луча. Возможны 2 значения: 3 цикла (31,5 кГц), если бит равен 0, и 5 циклов (15,7 кГц), если бит равен 1.
- Бит 7 используется для защиты регистров `00h`—`07h`. Если бит установлен в 1, запись вышеуказанных регистров будет невозможна, за исключением бита 4 регистра переполнения (`07h`).

4.2.4.19. Регистр *12h*

Размер регистра равен 10 бит. Два старших бита (8 и 9) расположены в регистре переполнения (`07h`). Содержит количество вертикальных линий в растре минус один для активного экрана.

4.2.4.20. Регистр 13h

Размер регистра равен 8 бит. Определяет смещение для ширины логического экрана в памяти. Для текстовых режимов значение данного регистра равно ширине экрана (в пикселах), деленной на двойной размер памяти. Для графических режимов смещение равно ширине экрана (в пикселах), деленной на произведение двойного размера памяти и количества пикселов в одном адресе памяти. Данный регистр позволяет создать виртуальный экран, превышающий физический размер.

4.2.4.21. Регистр *14h*

Размер регистра равен 8 бит. Предназначен для управления символом подчеркивания. Формат регистра показан в табл. 4.21.

Таблица 4.21. *Формат регистра 14h*

Бит	Описание
0—4	Позиция подчеркивания
5	Счетчик
6	Множитель
7	Резерв

Описание таблицы.

- Биты 0—4 определяют позицию на горизонтальной строке развертки минус 1.

- Бит 5, установленный в 1, позволяет синхронизировать значение счетчика импульсов со значением синхроимпульсов, деленным на 4.
- Бит 6, установленный в 1, позволит использовать для адресации памяти двойные слова (32 бита).

4.2.4.22. Регистр *15h*

Размер регистра равен 10 бит. Бит 8 расположен в регистре `07h`, а бит 9 в регистре `09h` (бит 5). Позволяет управлять значением начального хода луча по вертикали.

4.2.4.23. Регистр *16h*

Размер регистра равен 8 бит, но используются только биты с 0 по 6. Позволяет управлять значением конечного хода луча по вертикали.

4.2.4.24. Регистр *17h*

Размер регистра равен 8 бит. Предназначен для управления различными режимами работы. Формат регистра показан в табл. 4.22.

Таблица 4.22. *Формат регистра 17h*

Бит	Описание
0	Управление мультиплексором
1	Управление мультиплексором
2	Вертикальная синхронизация
3	Использование счетчика адреса
4	Резерв
5	Выбор адресов
6	Режим адресации
7	Управление синхронизацией

Описание таблицы.

- Бит 0 управляет битом 13 мультиплексора контроллера CRT. Если бит установлен в 1, то будет выбран счетчик адресов. Применяется для поддержки более старых контроллеров.
- Бит 1 управляет битом 14 мультиплексора контроллера CRT. Если бит установлен в 1, то будет выбран счетчик развертки по горизонтали.
- Бит 2 управляет вертикальной синхронизацией. Если бит установлен в 1, то горизонтальные синхроимпульсы делятся на два, что позволяет удвоить разрешение по вертикали.

- Бит 3 управляет счетчиком адресов. При установке бита в 1 счетчик адресов делит на два входные кодовые импульсы символов.
- Бит 5 позволяет управлять битами адресов 13 и 15. Используется для совместимости с некоторыми типами мониторов.
- Бит 6 позволяет установить желаемый режим адресации: байтами или словами. При установке этого бита в 0 будет использоваться адресация словами.
- Бит 7, установленный в 0, позволяет блокировать горизонтальные и вертикальные сигналы для обратного хода луча.

4.2.4.25. Регистр *18h*

Размер регистра равен 10 бит. Дополнительные биты 8 и 9 расположены в регистрах `07h` и `09h` соответственно. Регистр определяет разделяющую строку развертки по горизонтали.

На этом можно завершить описание регистров CRT. Рассмотрим простой пример доступа к регистру `0Ah` для удаления курсора с экрана (листинг 4.6).

Листинг 4.6. Удаление курсора с экрана

```
// переменная для хранения результата
DWORD dwResult  = 0;
// записываем значение регистра 0Ah в порт 03D4h
outPort (0x03D4, 0x0A, 1 );
// читаем из порта 3D5h
inPort ( 0x03D5, &dwResult, 1 );
// если курсор уже скрыт, выходим из процедуры
if ( ( dwResult & 0x20 ) == 1 )
   return;
// отключаем курсор
dwResult &= 0x20;
// записываем значение регистра 0Ah в порт 03D4h
outPort (0x03D4, 0x0A, 1 );
// записываем новое значение
outPort (0x03D5, dwResult, 1 );
```

Большинство из описываемых регистров CRT используются попарно (например, регистры `0Ch` и `0Dh` или `0Eh` и `0Fh`).

4.2.5. Регистры ЦАП

Регистры ЦАП предназначены для преобразования цифровых данных, поступающих с видеоконтроллера, в аналоговый сигнал, который поступает на экран дисплея. ЦАП поддерживает 256 регистров (256 цветовых оттенков одновременно), каждый из которых содержит по три значения уровней для трех основных цветов: красного, зеленого и синего. Размер каждого регистра равен 18 бит (по 6 битов на каждый цвет). Значение каждого цвета может лежать в диапазоне от 0 до 63 единиц. Для доступа к регистрам ЦАП используются три порта:

1. Порт `3C7h` в режиме записи управляет адресным регистром, а в режиме чтения считывает текущее состояние ЦАП.
2. Порт `3C8h` позволяет в режиме записи указать номер регистра таблицы цветов, а в режиме чтения возвращает текущий номер регистра таблицы цветов.
3. Порт `3C9h` в режиме записи позволяет установить новое значение для выбранного регистра таблицы цветов, а в режиме чтения считывает текущее значение цвета из указанного регистра цветов.

Для записи нового значения цвета требуется выполнить подряд три операции, по одной на каждую составляющую цвета (красный, зеленый, синий). После выполнения третьей операции индекс текущего регистра будет автоматически увеличен на единицу, что упрощает запись всей таблицы цветов для 256 регистров. Рекомендуется перед записью или чтением регистров ЦАП отключать прерывания.

Для записи новых значений цветов вначале следует записать в порт `3C8h` номер регистра таблицы цветов (от 0 до 256). После этого в регистр данных (порт `3C9h`) записываются последовательно три значения оттенка цвета, по одному на каждый основной цвет (красный, зеленый, синий). Регистр адреса (`3C8h`) увеличивает номер индекса на 1, и можно сразу продолжить запись трех составляющих цвета в следующий регистр таблицы цветов. Учитывая эту особенность, есть смысл за одну операцию записи установить новые значения цветов для всей таблицы (256 регистров). Запись значений палитры цветов необходимо выполнять во время обратного хода луча. Рассмотрим основные регистры ЦАП подробнее.

4.2.5.1. Регистр *3C7h*

Размер регистра равен 8 бит. В режиме чтения используются только 0 и 1 биты. Формат регистра в режиме записи показан в табл. 4.23, а в режиме чтения — в табл. 4.24.

Таблица 4.23. *Формат регистра 3C7h (запись)*

Бит	Описание
0—7	Номер регистра таблицы цветов (0—256)

Таблица 4.24. *Формат регистра 3C7h (чтение)*

Бит	Описание
0—1	Состояние
2—7	Резерв

Описание таблицы.

- Биты 0—1 определяют текущее состояние ЦАП: 0 — ЦАП находится в режиме чтения, 11_2 (3) — ЦАП находится в режиме записи.

4.2.5.2. Регистр *3C8h*

Размер регистра равен 8 бит. Используется для записи номера регистра палитры, для которого будут устанавливаться новые значения цветовых оттенков. В режиме чтения может вернуть текущий номер регистра палитры.

4.2.5.3. Регистр *3C9h*

Размер регистра равен 8 бит, но используются только младшие 6 разрядов. Применяется для записи или чтения значения цвета из выбранного регистра палитры. Операция чтения или записи выполняется три раза, для красного, зеленого и синего цветов. Формат регистра в режиме записи показан в табл. 4.25.

Таблица 4.25. *Формат регистра 3C9h (чтение и запись)*

Бит	Описание
0—5	Значение цвета
6—7	Резерв

Существует еще один регистр, адресуемый через порт `3C6h`. Он используется для маскирования значения цвета в одном из регистров таблицы цветов. Доступен для записи и считывания. По умолчанию значение в порту равно `FFh`. При обращении к регистру будет выполнена операция `AND` между содержи-

мым этого регистра и значением выбранного регистра таблицы цветов. Не рекомендуется писать в порт 3C6h, поскольку это может привести к разрушению таблицы цветов.

4.2.6. Регистры синхронизатора

Регистры синхронизатора управляют подстройкой данных, передаваемых ЦАП, а также позволяют корректировать работу цепей синхронизатора для программирования нестандартных видеорежимов. Вся работа с цепями синхронизатора построена на двух регистрах: адресном (порт 3C4h) и регистре данных (порт 3C5h). Они позволяют получить доступ к пяти (00h—04h) индексным регистрам. Перед началом работы следует записать в адресный регистр (3C4h) номер индексного регистра, а затем можно считывать или записывать данные через порт 3C5h. Рассмотрим индексные регистры подробнее.

4.2.6.1. Регистр *00h*

Размер регистра равен 8 бит, но используются только младшие два бита (0 и 1). Управляет режимом сброса цепей синхронизации. Формат регистра показан в табл. 4.26.

Таблица 4.26. *Формат регистра 00h*

Бит	Описание
0	Асинхронный сброс
1	Синхронный сброс
2—7	Резерв

Описание таблицы.

- Бит 0, установленный в 0, позволяет выполнить асинхронный сброс цепей синхронизатора и остановить посылку синхроимпульсов. Для восстановления работы необходимо установить бит в 1.
- Бит 1, установленный в 0, позволяет выполнить синхронный сброс цепей синхронизатора и остановить посылку синхроимпульсов. Для восстановления работы необходимо установить бит в 1.

4.2.6.2. Регистр *01h*

Размер регистра равен 8 бит. Предназначен для управления режимом синхронизации. Формат регистра представлен в табл. 4.27.

Таблица 4.27. *Формат регистра 01h*

Бит	Описание
0	Размер символов
1	Резерв
2	Частота преобразования
3	Частота обновления
4	Управление частотой преобразования
5	Блокировка дисплея
6—7	Резерв

Описание таблицы.

- Бит 0 позволяет установить размер символов. Возможны два значения: 8 точек (бит равен 1) и 9 точек (бит равен 0). Изменение размера символа может понадобиться при переключении текстовых режимов работы.
- Бит 2, установленный вместе с битом 4 в 0, определяет, что преобразование данных в выходных цепях происходит при каждом синхроимпульсе, иначе — через один.
- Бит 3, установленный в 0, частота обновления точки (или символа) совпадает с частотой тактового генератора. Если бит равен 1, то частота обновления будет уменьшена в два раза. Изменение частоты повлияет на все синхронизирующие импульсы, используемые в устройстве.
- Бит 4, установленный в 1, уменьшает частоту преобразования данных в четыре раза.
- Бит 5, установленный в 1, позволит выключить дисплей (блокировать передаваемые из видеопамяти данные). Может использоваться для полноэкранных модификаций изображения.

4.2.6.3. Регистр *02h*

Размер регистра равен 8 бит. Используются младшие четыре бита (0—3). Позволяет блокировать до четырех плоскостей видеопамяти (0—3). Установка бита в 1 разрешает запись в соответствующую плоскость.

4.2.6.4. Регистр *03h*

Размер регистра равен 8 бит. Регистр предназначен для выбора шрифта. Для текстовых режимов регистр позволяет загружать в знакогенератор видеоконтроллера наборы символов, хранящихся в памяти. Формат регистра показан в табл. 4.28.

Таблица 4.28. *Формат регистра 03h*

Бит	Описание
0—1	Шрифт B
2—3	Шрифт A
4	Выбор символа из шрифта B
5	Выбор символа из шрифта A
6—7	Резерв

Примечание

Для обозначения разных наборов символов, применяемых в текстовых режимах, здесь используется буквенные обозначения: шрифт A — первый набор, B — второй набор.

Описание таблицы.

- Биты 0—1 позволяют выбрать шрифт B, если бит 3 атрибута символа установлен в 0.
- Биты 2—3 позволяют выбрать шрифт A, если бит 3 атрибута символа установлен в 1. Адрес шрифта в памяти имеет следующие значения: `000b` — `0000h`—`1FFFh`, `001b` — `4000h`—`5FFFh`, `010b` — `8000h`—`9FFFh`, `011b` — `C000h`—`DFFFh`, `100b` — `2000h`—`3FFFh`, `101b` — `6000h`—`7FFFh`, `110b` — `A000h`—`BFFFh`.
- Бит 4 определяет бит 2 для поля шрифта B.
- Бит 5 определяет бит 2 для поля шрифта A.

4.2.6.5. Регистр *04h*

Размер регистра равен 8 бит. Позволяет управлять режимом работы видеопамяти. Формат регистра показан в табл. 4.29.

Таблица 4.29. *Формат регистра 04h*

Бит	Описание
0	Резерв
1	Расширенная память
2	Выбор адресов
3	Управление операциями чтения
4—7	Резерв

Описание таблицы.

- Бит 1, установленный в 1, позволяет расширить объем адресуемой памяти от 64 до 256 Кбайт.
- Бит 2 управляет доступом к четным и нечетным плоскостям в памяти. При установке бита в 0 четные адреса работают с плоскостями 0 и 2, а нечетные — с 1 и 3.
- Бит 3, установленный в 0, позволяет последовательно считывать адресуемые данные из памяти.

Рассмотрим упрощенный пример работы с регистрами синхронизатора для выполнения синхронного сброса (листинг 4.7).

Листинг 4.7. Выполнение синхронного сброса

```
// переменная для хранения результата
DWORD dwResult  = 0;
// записываем значение регистра 00h в порт 03C4h
outPort (0x03C4, 0x00, 1 );
// синхронный сброс
outPort (0x03C5, 0x01, 1 );
// восстанавливаем синхронизацию
// записываем значение регистра 00h в порт 03C4h
outPort (0x03C4, 0x00, 1 );
// восстанавливаем сигнал синхронизации
outPort (0x03C5, 0x03, 1 );
```

На этом примере можно завершить программирование видеоконтроллера с использованием аппаратных портов и перейти к рассмотрению возможностей интерфейса Win32 API для работы с видеоадаптером и монитором.

4.3. Использование Win32 API

Программисту, работающему с интерфейсом Win32, практически не нужно думать о непосредственном взаимодействии с видеоконтроллером и монитором. Интерфейс предоставляет универсальные законченные решения для вывода графической и текстовой информации на экран. Этих возможностей вполне достаточно для разработки большинства пользовательских программ. Даже для написания игровых приложений существуют дополнительные библиотеки (например, Direct X), которые в тандеме с функциями Win32 API позволяют решить практически любые задачи. При этом скорость работы таких программ остается достаточно высокой, а если учесть мощь современных

процессоров (в том числе и графических), то вполне понятно, почему разработчики предпочитают стандартные библиотеки Win32 API непосредственному программированию аппаратных портов.

Здесь мы не будем изучать программирование графики в Windows, а рассмотрим только возможности управления видеоконтроллером и монитором, любезно "сохраненные" фирмой Microsoft для разработчиков программного обеспечения:

1. Управление графическими режимами.
2. Получение информации о возможностях видеоконтроллера.
3. Управление монитором в Windows.

4.3.1. Управление графическими режимами

В Windows существует возможность программно установить любой поддерживаемый видеоадаптером режим работы. Для этого сначала нужно получить список всех поддерживаемых режимов, а затем активизировать его. Сразу хочу заметить, что видеоадаптер, как правило, поддерживает гораздо больше режимов, чем установленный монитор, поэтому важно перед установкой нового режима точно знать, будет ли с ним работать монитор. Игнорирование этого правила может привести даже к выходу из строя монитора. Программист обычно не может заранее знать аппаратную конфигурацию потенциального пользователя, поэтому вся ответственность выбора графического режима ложится на пользователя. Однако рекомендую программистам перед установкой нового режима выводить предупреждающие сообщения, затем активизировать новый режим на несколько секунд (10—20), после чего возвращаться к предыдущему. Это позволит уберечь начинающего пользователя от грубых ошибок и добавить вашей программе только плюсы.

Для получения поддерживаемых режимов используется функция `EnumDisplaySettings`, которая имеет три аргумента. Первый аргумент является указателем на строковое значение с именем устройства вывода. Для Windows 95/98/ME это значение должно быть установлено в `NULL`. Для Windows NT/2000/XP/2003/Vista можно указать параметр `DeviceName` структуры `DISPLAY_DEVICE`. Данная структура позволяет сохранять различные параметры для устройства вывода. Используется в функции `EnumDisplayDevices` и будет рассмотрена далее в этой главе. Второй аргумент функции `EnumDisplaySettings` определяет тип возвращаемой информации и может принимать одно из следующих значений: `ENUM_CURRENT_SETTINGS` (получить текущие параметры) или `ENUM_REGISTRY_SETTINGS` (получить параметры из реестра). Кроме того, этот аргумент является своеобразным счетчиком. Данная функция должна вызываться определенное число раз (по одному для

каждого поддерживаемого режима) и второй аргумент увеличивает значение счетчика при каждом вызове функции, начиная с 0. Последний аргумент функции является указателем на структуру `DEVMODE`, в которую записывается информация о найденном режиме. Формат данной структуры очень велик и многие ее поля не имеют прямого отношения к информации о режиме, поэтому приведу описание только нужных нам полей:

- Поле `dmBitsPerPel` предназначено для хранения информации о глубине цвета (в битах) для найденного режима: 4 бита для 16 цветов, 8 — для 256 цветов, 16 — для 65536 и т. д.
- Поле `dmPelsWidth` определяет разрешение по горизонтали в пикселах.
- Поле `dmPelsHeight` определяет разрешение по вертикали в пикселах.
- Поле `dmDisplayFrequency` определяет частоту кадровой (вертикальной) развертки (частота обновления экрана) в герцах. Данное поле не поддерживается в Windows 95/98/ME.

Кроме того, перед использованием структуры `DEVMODE` следует записать в поле `dmSize` размер самой структуры (`sizeof(DEVMODE)`), а в поле `dmDriverExtra` значение 0. Теперь у нас есть все необходимые составляющие, чтобы написать собственную функцию для получения всех поддерживаемых устройством режимов. Пример такой функции показан в листинге 4.8.

Листинг 4.8. Перечисление всех поддерживаемых видеорежимов

```
// определим глобальную переменную
DEVMODE* dvMode = NULL; // указатель на структуру DEVMODE
// назовем нашу функцию GetDisplayModes
void GetDisplayModes ( HWND hComboBox )
{
// счетчик для функции EnumDisplaySettings
DWORD dwNumberMode = ( DWORD ) -1;
// буфер форматирования и вывода информации
TCHAR tszMode [100];
// выделяем необходимое количество памяти
dvMode = new DEVMODE [250]; // максимум 250 режимов, если найдутся
// если нет свободной памяти, завершаем функцию
if ( dvMode == NULL ) return;
// пишем цикл для поиска всех возможных режимов
do {
     // увеличиваем счетчик на единицу
     dwNumberMode ++;
     // заполняем требуемые поля структуры DEVMODE
     dvMode [dwNumberMode].dmSize = sizeof ( DEVMODE );
```

```
     dvMode [dwNumberMode].dmDriverExtra = 0;
} while ( EnumDisplaySettings ( NULL, dwNumberMode,
                                & dvMode [dwNumberMode] ) );
// форматируем полученные данные
 for ( int i = 0; i < dwNumberMode; i++ )
 {
  // если в поле dmDisplayFrequency стоит 0 или 1, игнорируем его
  if (dvMode [i].dmDisplayFrequency == 0 ||
      dvMode [i].dmDisplayFrequency == 1) // Windows 95-98-ME
    {
       sprintf ( tszMode, __TEXT ( "%d x %d (%d bit),  0 Hz"),
                dvMode [i].dmPelsWidth, dvMode [i].dmPelsHeight,
                dvMode [i].dmBitsPerPel );
    }
  else // Windows NT-2000-XP-2003
    {
       wsprintf ( tszMode, __TEXT ("%d x %d (%d bit), %d Hz" ),
                dvMode [i].dmPelsWidth, dvMode [i].dmPelsHeight,
                dvMode [i].dmBitsPerPel, dvMode [i].dmDisplayFrequency );
    } // end if
  // записываем отформатированную строку во внешний список
  SendMessage ( hComboBox, CB_ADDSTRING, 0, (LPARAM) (LPCTSTR) tszMode );
 } // end for
// выбираем в списке самый верхний пункт
SendMessage ( hComboBox, CB_SETCURSEL, 0, 0 );
// освобождаем память
if ( dvMode ) delete [] dvMode;
} // выходим из функции
```

Как видно из этого примера, наша функция `GetDisplayModes` имеет всего один аргумент — дескриптор окна со списком (ComboBox). После выполнения функции в списке будут размещены все найденные режимы. Поскольку некоторые системы (Windows 95, 98 и ME) не возвращают значение вертикальной частоты, мы использовали дополнительное условие, в котором проверяли значение поля `dmDisplayFrequency`. Если оно равно 0 или 1, значит, операционная система не поддерживает получение частоты. Для надежности можно проверить текущую версию Windows с помощью функции `GetVersion`. Рекомендую всем читателям в профессиональных приложениях использовать именно такой вариант.

Созданная нами функция прекрасно справляется с поставленной задачей, но имеет один неприятный сюрприз: информация о поддерживаемых режимах выводится не по порядку, а хаотично. Этот минус легко исправить, добавив код сортировки найденных режимов (например, с помощью функции `qsort`).

После того как мы определили все режимы, можно использовать функцию `ChangeDisplaySettings` для установки нового режима. Функция имеет всего два аргумента, первый из которых является указателем на структуру `DEVMODE`, а второй определяет флаг выполнения операции. Список всех флагов показан в табл. 4.30.

***Таблица 4.30.** Список флагов функции `ChangeDisplaySettings`*

Флаг	Описание
`0`	Смена режима будет выполнена динамически
`CDS_TEST`	Пробная установка требуемого режима
`CDS_UPDATEREGISTRY`	Смена режима будет выполнена динамически и сохранена в системном реестре
`CDS_SET_PRIMARY`	Позволяет выбрать текущее устройство вывода на экран первичным
`CDS_RESET`	Смена режима будет выполнена, даже если требуемый режим уже установлен ранее

После выполнения функция `ChangeDisplaySettings` возвращает определенное значение, по которому легко узнать результат операции. При успешной смене режима будет возвращено значение `DISP_CHANGE_SUCCESSFUL`. Если функция возвратит значение `DISP_CHANGE_RESTART`, значит, необходима перезагрузка компьютера для активизации выбранного графического режима. В случае невозможности установить требуемый режим, функция вернет значение `DISP_CHANGE_FAILED`, а если указанный графический режим не поддерживается, то — `DISP_CHANGE_BADMODE`. А теперь рассмотрим на практике установку нового графического режима в Windows. Для этого напишем код, как показано в листинге 4.9.

Листинг 4.9. Установка нового графического режима

```
// назовем функцию SetDisplayModes
int SetDisplayModes ( DEVMODE* mode )
{
LONG lResult = 0;
// проверяем указатель
if ( mode == NULL ) return -1;
// устанавливаем новый режим дисплея
lResult = ChangeDisplaySettings ( mode, CDS_UPDATEREGISTRY );
// обрабатываем результат операции
switch ( lResult )
```

```
{
    case DISP_CHANGE_SUCCESSFUL:
      // операция успешно завершена
      return 0;
    case DISP_CHANGE_FAILED:
      // не удалось установить требуемый режим
      return 1;
    case DISP_CHANGE_RESTART:
      // требуется выполнить перезагрузку ПК для активации режима
      return 2;
}
// неизвестная ошибка
return -1;
}
// теперь можно вызвать нашу функцию для установки режима
SetDisplayModes ( &dvMode [2] ); // выбираем из списка режим 2
```

Поскольку указатель на структуру `DEVMODE` мы сделали глобальным, можно вызывать нашу функцию `SetDisplayModes` в любом удобном месте. В качестве аргумента ей передается указатель на `DEVMODE`. Номер выбранного нами режима (2) абсолютно случаен. Кроме того, еще раз замечу, что описания режимов в массиве структур `DEVMODE` расположены хаотично, а потому нужно быть очень внимательным при передаче номера режима, а лучше отсортировать все режимы по порядку.

4.3.2. Проверка возможностей видеоадаптера

В Windows существует удобная возможность получить различную полезную информацию о параметрах текущего графического режима, а также совместимости установленного видеоадаптера с различными графическими функциями рисования и управления цветом. Со всеми этими задачами справляется одна-единственная функция `GetDeviceCaps`. Она имеет всего два аргумента: первый определяет контекст устройства (для получения текущих параметров можно установить в `NULL`), а второй — флаг для требуемого параметра. Список интересующих нас флагов представлен в табл. 4.31. Возвращаемое значение функции зависит от установленного флага.

Таблица 4.31. *Список флагов функции* `GetDeviceCaps`

Флаг	Описание
`DRIVERVERSION`	Позволяет получить версию драйвера устройства
`ASPECTX`	Относительная ширина пиксела устройства для рисования линии

Таблица 4.31 *(окончание)*

Флаг	Описание
ASPECTY	Относительная высота пиксела устройства для рисования линии
ASPECTXY	Ширина пиксела устройства по диагонали для рисования линии
BITSPIXEL	Число граничных битов для отображения цвета каждого пиксела
CLIPCAPS	Поддержка устройством прямоугольных областей отсечения (1 — есть, 0 — нет)
COLORRES	Фактическое разрешение по цвету (бит/пиксел). Следует применять только, если установлен бит RC_PALETTE для флага RASTERCAPS
CURVECAPS	Определяет поддержку вывода кривых линий (CC_NONE — нет, CC_CIRCLES — окружность, CC_ELLIPSES — эллипс, CC_ROUNDRECT — скругленный прямоугольник, CC_PIE — сектор, CC_CHORD — хорда). Возвращаемое значение является комбинацией возможных значений
HORZRES	Ширина экрана в пикселах
VERTRES	Высота экрана в строках растра
HORZSIZE	Ширина физического экрана в миллиметрах
VERTSIZE	Высота физического экрана в миллиметрах
LINECAPS	Определяет поддержку вывода кривых линий (LC_NONE — нет, LC_MARKER — маркер, LC_POLYLINE — ломаная линия, LC_WIDE — широкая линия). Возвращаемое значение является комбинацией возможных значений
LOGPIXELSX	Количество пикселов на одном логическом дюйме по ширине
LOGPIXELSY	Количество пикселов на одном логическом дюйме по высоте
NUMBRUSHES	Поддерживаемое число кистей
NUMPENS	Поддерживаемое число перьев
NUMFONTS	Поддерживаемое число шрифтов
PLANES	Количество цветовых плоскостей
NUMCOLORS	Если устройство не поддерживает цвет больше 8 битов на пиксел, будет возвращено число входов в таблице цветов, иначе -1
RASTERCAPS	Определяет поддержку растровых операций (RC_PALETTE — зависит от палитры, RC_BITBLT — растровые изображения, RC_SCALING — масштабирование). Возвращаемое значение является комбинацией возможных значений
SIZEPALETTE	Количество входов в системной палитры (должен быть установлен бит RC_PALETTE для флага RASTERCAPS)
VREFRESH	Частота вертикальной развертки в герцах (в Windows 95/98/ME не поддерживается)

Попробуем на примере воспользоваться данной функцией. В листинге 4.10 приводится удобный способ получения параметров текущего графического режима.

Листинг 4.10. Получение параметров для текущего графического режима

```
// объявляем переменные
HDC hDC = NULL; // дескриптор контекста устройства
// переменные для хранения параметров режима
DWORD dwX = 0, dwY = 0, dwBit = 0, dwHz = 0;
DWORD dwRaster = 0, dwColors = 0;
// получаем текущий контекст устройства
hDC = GetDC ( NULL );
dwX = GetDeviceCaps ( hDC, HORZRES ); // разрешение по горизонтали
dwY = GetDeviceCaps ( hDC, VERTRES ); // разрешение по вертикали
dwBit = GetDeviceCaps ( hDC, BITSPIXEL ); // разрядность цвета
dwHz = GetDeviceCaps ( hDC, VREFRESH ); // частота по вертикали
// проверяем поддержку растра
dwRaster = GetDeviceCaps ( hDC, RASTERCAPS );
// проверяем поддержку палитры
dwRaster = ( dwRaster & RC_PALETTE ) ? true : false;
 if ( dwRaster ) // палитра поддерживается
    // получаем число цветов
    dwColors = GetDeviceCaps ( hDC, SIZEPALETTE );
 else // палитра не поддерживается
    dwColors = GetDeviceCaps ( hDC, NUMCOLORS );
// освобождаем контекст устройства
ReleaseDC ( NULL, hDC );
```

4.3.3. Управление монитором

И последнее, с чем хочется познакомить читателя, — это возможности управления монитором. Выбор графических режимов мы уже разобрали, а теперь поговорим о поддержке Win32 API энергосберегающих возможностей монитора. Существует, как минимум, две операции по управлению питанием монитора: первая позволяет перевести монитор в ждущий режим, а вторая полностью выключает основные цепи питания, переводя устройство в режим сна. Во втором режиме потребление питания наименьшее. Сразу замечу, что описываемые возможности будут работать только с мониторами, поддерживающими энергосберегающие режимы (все современные устройства).

Для управления режимами мы будем применять функцию `PostMessage`. Для тех, кто не знаком с ней, скажу, что данная функция позволяет поместить

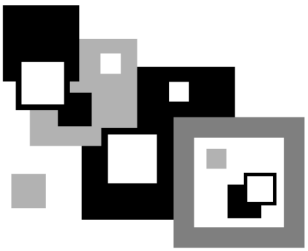
любое сообщение в общую очередь, связанную с текущим процессом, и возвратить управление без ожидания результата. Для управления монитором мы должны будем послать ему системное сообщение `WM_SYSCOMMAND` с дополнительным параметром `SC_MONITORPOWER`. Как это делается, можно посмотреть в листинге 4.11.

Листинг 4.11. Управление питанием монитора

```
// функция для перевода монитора в ждущий режим
void MonitorLowPower ( )
{
   PostMessage ( HWND_BROADCAST, WM_SYSCOMMAND, SC_MONITORPOWER, 1L );
}
// функция для выключения монитора
void MonitorOffPower ( )
{
   PostMessage ( HWND_BROADCAST, WM_SYSCOMMAND, SC_MONITORPOWER, 2L );
}
```

Как видите, все достаточно просто и эффективно. Первый аргумент функции `PostMessage` (`HWND_BROADCAST`) указывает на то, что сообщение должно быть послано всем окнам верхнего уровня (в том числе скрытым и зависшим) в системе. Второй аргумент определяет сообщение (`WM_SYSCOMMAND`). Третий и четвертый аргументы описывают дополнительные параметры управления. В данном случае это специальное значение для управления питанием монитора и код операции (1 или 2).

На этом, думаю, можно закончить рассмотрение вопросов программирования видеоконтроллера и монитора.



ГЛАВА 5

Работа с видео

Пожалуй, возможность просмотра видеофильмов является второй после игрушек причиной покупки современного домашнего компьютера. И это не удивительно, с покупкой мультимедийного компьютера на второй план уходит видеомагнитофон с его громоздкими кассетами, а также посредственным качеством изображения и звука.

Для реализации поддержки видео в операционных системах Windows фирма Microsoft разработала собственный формат, ставший стандартом де-факто. Называется он *AVI* (Audio-Video Interleaved — формат с чередованием аудио- и видеоданных). Данный формат поддерживает видеоизображение и звук, синхронизированные между собой. Файл в формате AVI имеет расширение с таким же названием (.avi) и содержит последовательность растровых изображений, а также один или более каналов звука. Формат AVI может использовать различные методы сжатия повторяющихся данных, что позволяет немного уменьшить конечный размер файла, но тем не менее обеспечивает отличное качество изображения и звука. Однако, несмотря на это, использование данного формата ограничено в основном предварительной обработкой (например, оцифровка данных с видеокамеры с последующим редактированием) исходных видеоданных. Связано это с тем, что даже несмотря на сжатие данных, выходной файл может иметь огромные размеры (зависит от разрешения и глубины цвета). Для записи же видеоданных на диски используются, как правило, другие форматы, обеспечивающие более высокую степень компрессии (например, MPEG 2, 4).

В любом случае, если вы работаете в Windows, так или иначе придется столкнуться с форматом AVI (оцифровка аналоговых данных, организация интерактивного интерфейса и т. д.). Поскольку этот формат является стандартным для Windows, фирма Microsoft позаботилась о его программной поддержке, добавив соответствующие возможности в Win32 API. В этой гла-

ве мы и попытаемся научиться работать с AVI-файлами посредством интерфейса Win32 API. Для этого мы воспользуемся двумя различными как по сложности, так и по возможностям способами:

1. Использование универсального интерфейса управления мультимедийными данными (*MCI* — Media Control Interface).
2. Использование набора функций поддержки видео в Windows (*VFW* — Video for Windows).

5.1. Использование MCI

Первый способ с использованием универсального интерфейса управления мультимедийными данными — *MCI* (Media Control Interface) наиболее простой и позволяет быстро добавить в программу поддержку файлов в формате AVI, но имеет ограниченные возможности. Идеально подходит для создания интерактивного интерфейса, а также различных анимационных эффектов. Кроме видеофайлов, он поддерживает аудиодиски (CDDA), файлы WAV и MIDI. Здесь мы рассмотрим только ту часть, которая относится к работе с видео, а в *гл*аве *7* познакомимся с программированием звука.

Для активизации интерфейса MCI необходимо добавить заголовочные файлы Mmsystem.h и Vfw.h, а также библиотеки Winmm.lib и Vfw.lib. Это необходимо для правильной инициализации функций, структур и констант, поддерживающих мультимедийные возможности для Windows.

Чтобы воспроизвести файл AVI, можно воспользоваться следующими макрокомандами:

- `MCIWndCreate`

  Является функцией. Позволяет открыть устройство MCI или AVI-файл и зарегистрировать собственное окно для просмотра видеоданных;

- `MCIWndPlay`

  Начинает воспроизведение текущего файла;

- `MCIWndHome`

  Позволяет установить исходное состояние воспроизведения (устанавливает курсор чтения в начало файла);

- `MCIWndPause`

  Приостанавливает текущее воспроизведение;

- `MCIWndResume`

  Продолжает воспроизведение, остановленное командой `MCIWndPause`;

- `MCIWndStop`

  Останавливает текущее воспроизведение;

- `MCIWndDestroy`

  Необходимо вызвать эту команду для закрытия устройства MCI и закрытия окна просмотра;

- `MCIWndClose`

  Данная макрокоманда также закрывает устройство MCI, но оставляет окно просмотра активным. В дальнейшем его можно использовать для повторного просмотра AVI-файла.

Существует еще ряд дополнительных макрокоманд, но в данном случае они нам не понадобятся. Прежде чем перейти к практике, разберем подробнее перечисленные выше макрокоманды и функции.

Функция `MCIWndCreate` располагает четырьмя аргументами, каждый из которых имеет определяющее значение: первый аргумент `hwndParent` указывает на дескриптор (`HWND`) родительского окна. Второй аргумент `hInstance` содержит дескриптор текущего приложения (`HINSTANCE`). Третий (`dwStyle`) определяет комбинацию стандартных стилей для окна. Дополнительно к ним можно указать специфические стили, используемые только в этой функции: `MCIWNDF_NOAUTOSIZEWINDOW` — не изменять размеры окна при изменении размеров отображаемых видеоданных, `MCIWNDF_NOERRORDLG` — отключить сообщения об ошибках MCI, `MCIWNDF_NOOPEN` — блокировать команды меню для открытия файла, `MCIWNDF_NOPLAYBAR` — скрывать панель инструментов, `MCIWNDF_NOTIFYPOS` — уведомлять о текущей позиции воспроизведения, `MCIWNDF_NOTIFYALL` — использовать все доступные уведомления, `MCIWNDF_SHOWPOS` — отображать текущую позицию в области заголовка. Последний четвертый аргумент функции указывает на текстовую строку с нулевым символом на конце, которая в свою очередь указывает на имя устройства MCI или AVI-файла. В случае успешного завершения функция `MCIWndCreate` возвращает дескриптор окна (`HWND`) для вывода видеоданных. Полученный дескриптор окна понадобится нам в качестве основного аргумента для используемых в дальнейшем макрокоманд.

Рассмотрим пример программы, которая позволяет воспроизводить видеофайлы в формате AVI или MPG в главном окне. Основной код программы представлен в листинге 5.1.

Листинг 5.1. Воспроизведение AVI- или MPG-файлов средствами MCI

```
// подключаем основные системные файлы
#include <windows.h>
#include <vfw.h>
#include "resource.h"
```

```
// глобальные переменные
HINSTANCE hInst;     // дескриптор программы
HWND hwndAVI = NULL; // дескриптор окна вывода данных
BOOL bPause = FALSE; // состояние воспроизведения
BOOL bOpen = FALSE;  // состояние наличия открытого файла
LPCTSTR lpszTitle = "AVI Player"; // заголовок окна
LPCTSTR lpszAppName = "My AVI Player"; // уникальное имя программы
// главная оконная функция
LRESULT CALLBACK WindowAVIProc ( HWND hWnd, UINT uMsg, WPARAM wParam,
                                 LPARAM lParam );
// функция для открытия нового видеофайла
void OpenAVIMovie ( HWND hWnd );
// главная функция программы
int APIENTRY WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow )
{
   MSG        msg;
   HWND       hWnd;
   WNDCLASSEX wc;
// определяем параметры окна программы
wc.style         = CS_HREDRAW | CS_VREDRAW;
   wc.lpfnWndProc   = ( WNDPROC ) WindowAVIProc;
   wc.cbClsExtra    = 0;
   wc.cbWndExtra    = 0;
   wc.hInstance     = hInstance;
   wc.hIcon         = NULL;
   wc.hCursor       = LoadCursor ( NULL, IDC_ARROW );
   wc.hbrBackground = ( HBRUSH ) ( COLOR_WINDOW +1 );
// имя нашего меню из файла ресурсов
   wc.lpszMenuName  = "AVIPLAYER";
   wc.lpszClassName = lpszAppName;
   wc.cbSize        = sizeof ( WNDCLASSEX );
   wc.hIconSm       = NULL;
// и регистрируем собственный класс окна в системе
   if ( !RegisterClassEx ( &wc ) )
      return ( FALSE );
hInst = hInstance;
// создаем внешний вид окна
hWnd = CreateWindow ( lpszAppName, lpszTitle, WS_OVERLAPPEDWINDOW,
300, 0, 200, 0, NULL, NULL, hInstance, NULL );
// если не удалось создать окно, завершаем программу
if ( !hWnd )
      return ( FALSE );
```

```
// отображаем окно на экране
ShowWindow ( hWnd, nCmdShow );
UpdateWindow ( hWnd );
// открываем и инициализируем устройство MCI
hwndAVI = MCIWndCreate ( hWnd, hInstance, WS_CHILD | WS_VISIBLE |
MCIWNDF_NOOPEN | MCIWNDF_NOPLAYBAR | MCIWNDF_NOTIFYSIZE |
MCIWNDF_NOERRORDLG, NULL );
  // если открыть MCI не удалось, завершаем программу
  if ( !hwndAVI )
   {
      DestroyWindow ( hWnd );
            return ( FALSE );
   }
// запускаем цикл обработки сообщений
while ( GetMessage ( &msg, NULL, 0, 0 ) )
   {
      TranslateMessage ( &msg );
      DispatchMessage ( &msg );
   }
return( msg.wParam );
}
// главная оконная функция нашей программы
LRESULT CALLBACK WindowAVIProc ( HWND hWnd, UINT uMsg, WPARAM wParam,
                                 LPARAM lParam )
{
switch ( uMsg )
   {
      // обрабатываем команды меню
      case WM_COMMAND:
            switch ( LOWORD ( wParam ) )
                {
                  // открываем файл
                  case IDM_OPEN:
                       OpenAVIMovie ( hWnd );
                    break;
                  // начинаем воспроизведение
                  case IDM_PLAY:
                        if ( hwndAVI && bOpen )
                          MCIWndPlay ( hwndAVI );
                    break;
                  // обработка паузы
                  case IDM_PAUSE:
                       bPause = !bPause;
```

```
                    // приостанавливаем воспроизведение
                    if ( bPause )
                      MCIWndPause ( hwndAVI );
                    else // продолжаем воспроизведение
                      MCIWndResume ( hwndAVI );
                  break;
                // останавливаем воспроизведение текущего файла
                case IDM_STOP:
                     MCIWndStop ( hwndAVI );
                  break;
                // переводим позицию воспроизведения в начало файла
                case IDM_RESET:
                     MCIWndHome ( hwndAVI );
                  break;
                // закрываем текущий видеофайл
                case IDM_CLOSE:
                     bOpen = FALSE;
                     bPause = FALSE;
                     MCIWndClose ( hwndAVI );
                  break;
                // завершаем работу с программой
                case IDM_EXIT:
                     // закрываем текущий видеофайл
                     MCIWndClose ( hwndAVI );
                // закрываем устройство MCI
                     MCIWndDestroy ( hwndAVI );
                // закрываем окно программы
                     DestroyWindow( hWnd );
                  break;
              }
          break;
      // завершаем работу с программой
      case WM_DESTROY :
             PostQuitMessage ( 0 );
           break;
      default:
            return ( DefWindowProc ( hWnd, uMsg, wParam, lParam ) );
   }
   return( 0L );
}
// функция для открытия нового видеофайла
void OpenAVIMovie ( HWND hWnd )
{
```

```
    // объявляем переменные
    OPENFILENAME ofn;
    static char szFile [MAX_PATH];
    static char szFileTitle [MAX_PATH];
    // обнуляем структуру OPENFILENAME
    memset ( &ofn, 0, sizeof ( OPENFILENAME ) );
// инициализируем структуру OPENFILENAME
    ofn.lStructSize = sizeof ( OPENFILENAME );
    ofn.hwndOwner = hWnd;
    ofn.lpstrFilter = "AVI Files\0*.avi\0MPG Files\0*.mpg\0
                       All Files\0*.*\0\0";
    ofn.lpstrFile = szFile;
    ofn.nMaxFile = sizeof ( szFile );
    ofn.lpstrFileTitle = szFileTitle;
    ofn.nMaxFileTitle = sizeof ( szFileTitle );
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
    // открываем файл
    if ( GetOpenFileName ( &ofn ) )
       {
          // если открыт предыдущий файл, закрываем его
          if ( bOpen )
             MCIWndClose ( hwndAVI );
    bOpen = TRUE;
    if ( MCIWndOpen ( hwndAVI, ofn.lpstrFile, 0 ) == 0 )
       {
          ShowWindow ( hwndAVI, SW_SHOW );
       }
    else // не удалось открыть выбранный файл
       {
          bOpen = FALSE;
          // выводим сообщение об ошибке
          MessageBox ( hWnd, "Не удалось открыть выбранный файл", NULL,
                       MB_ICONEXCLAMATION | MB_OK );
       }
    }
// обновляем окно просмотра
InvalidateRect ( hWnd, NULL, FALSE );
UpdateWindow ( hWnd );
}
```

Кроме основного файла, необходимо создать в редакторе ресурсов собственное меню. При этом, в ваш проект будет автоматически добавлен файл

resource.h с описанием идентификаторов меню. Примеры файла и описателя ресурсов приведены в листинге 5.2 и 5.3 соответственно.

Листинг 5.2. Вид созданного редактором ресурсов файла (расширение rc)

```
//Microsoft Developer Studio generated resource script.
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
#include "afxres.h"
#undef APSTUDIO_READONLY_SYMBOLS
// Russian resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_RUS)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32
// Menu
AVIPLAYER MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "Open File...",                IDM_OPEN
        MENUITEM "Close AVI File",              IDM_CLOSE
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                       40006
    END
    POPUP "&Video"
    BEGIN
        MENUITEM "Play",                        40003
        MENUITEM "Pause",                       40004
        MENUITEM "Stop",                        40005
        MENUITEM SEPARATOR
        MENUITEM "Reset",                       IDM_RESET
    END
END
#endif    // Russian resources
```

Листинг 5.3. Файл описателя ресурсов resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Closewnd.rc
//
```

```
#define IDM_OPEN                        40001
#define IDM_CLOSE                       40002
#define IDM_PLAY                        40003
#define IDM_PAUSE                       40004
#define IDM_STOP                        40005
#define IDM_EXIT                        40006
#define IDM_TEST                        40007
#define IDM_RESET                       40007
//
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NO_MFC                     1
#define _APS_NEXT_RESOURCE_VALUE        101
#define _APS_NEXT_COMMAND_VALUE         40008
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

Не забудьте в опциях компоновщика (**Project | Settings | Link**) добавить ссылку на библиотеку Vfw.lib. После этого скомпилируйте файл и запустите. Если все сделано правильно, то в главном окне программы будет расположено меню, состоящее из двух пунктов: **File** и **Video**. Первое меню управляет открытием и закрытием видеофайла, а также завершением работы. Оно состоит из следующих пунктов: **OpenFile**, **Close AVI File** и **Exit**. Второе меню содержит основные программы управления открытым файлом: **Play**, **Pause**, **Stop** и **Reset**. Кроме стандартных файлов AVI, вы сможете открывать и просматривать некоторые файлы в формате MPG.

Итак, у нас получилась простая программа для просмотра видеофайлов. Однако у нее имеется один недостаток: размеры главного окна программы автоматически не подгоняются под размер видеоизображения. Это неудобство очень легко исправить. Для этого необходимо добавить в программу обработку сообщений интерфейса MCI. Для корректировки размеров окна применяется сообщение `MCIWNDM_NOTIFYSIZE`. Оно служит для уведомления родительского окна об изменении размеров окна вывода изображения. Дополнительно следует обработать стандартное сообщение Windows, информирующее об изменении размеров окна `WM_SIZE`. Добавьте в основной файл программы (после обработки сообщения `WM_DESTROY`) обработку указанных сообщений, как показано в листинге 5.4.

Листинг 5.4. Обработка сообщений для корректировки размеров окна

```
// отрывок из функции WindowAVIProc
// завершаем работу с программой
      case WM_DESTROY:
              PostQuitMessage ( 0 );
            break;
// добавляем обработку размеров окна вывода
      case WM_SIZE:
      // если окно существует и не свернуто
      if ( hwndAVI && bOpen && !IsIconic ( hWnd ) )
      // изменяем его размеры
           MoveWindow ( hwndAVI, 0, 0, LOWORD ( lParam ),
                        HIWORD( lParam ), TRUE );
            break;
// обрабатываем сообщение MCIWNDM_NOTIFYSIZE
      case MCIWNDM_NOTIFYSIZE:
       // если окно не свернуто
       if ( !IsIconic ( hWnd ) )
         {
           // если файл открыт
           if ( bOpen )
             {
              // устанавливаем размеры окна вывода
              GetWindowRect ( hwndAVI, &rc );
              AdjustWindowRect ( &rc, GetWindowLong ( hWnd,
                                 GWL_STYLE ), TRUE );
              SetWindowPos ( hWnd, NULL, 0, 0, rc.right — rc.left,
       rc.bottom — rc.top, SWP_NOZORDER | SWP_NOACTIVATE | SWP_NOMOVE );
             }
           else
             {
             // восстанавливаем размеры окна по умолчанию
             SetWindowPos ( hWnd, NULL, 0, 0, 300, 200,
                            SWP_NOZORDER | SWP_NOACTIVATE | SWP_NOMOVE );
             }
         }
            break;
default :
            return ( DefWindowProc ( hWnd, uMsg, wParam, lParam ) );
```

Теперь наша программа стала более похожа на стандартный проигрыватель видеофайлов. Давайте добавим еще возможность изменения масштаба окна видеоданных. Для этого используются две макрокоманды: `MCIWndGetZoom` и

`MCIWndSetZoom`. Первая получает текущее значение, а вторая позволяет установить новое. Поддерживаются три стандартных значения масштабирования, выражаемые в процентах: 50, 100 и 200 %. При установке значения 50 %, размер окна будет уменьшен в два раза, а значение 200 % позволит увеличить нормальное изображение в два раза. Установка масштаба в 100 % возвращает нормальный размер картинки. Макрокоманда `MCIWndSetZoom` имеет два аргумента: первый определяет дескриптор окна вывода, а второй указывает значение желаемого масштаба. Добавьте в программу код, позволяющий увеличить изображение вдвое и представленный в листинге 5.5. Поместите его в функцию главного окна `WindowAVIProc`.

Листинг 5.5. Масштабирование окна видеоданных

```
// увеличиваем изображение в два раза
case IDM_ZOOMX2:
{
UINT uCurrentZoom = 0;
// получаем текущее значение
uCurrentZoom = MCIWndGetZoom ( hwndAVI );
// проверяем, имеет ли окно нормальный размер
if ( uCurrentZoom == 100 )
   MCIWndSetZoom ( hwndAVI, 200 ); // двойной экран
}
break;
```

А теперь рассмотрим второй способ воспроизведения файлов в формате AVI.

5.2. Использование VFW

Данный метод с использованием набора функций поддержки видео в Windows — *VFW* (Video for Windows) имеет больше возможностей по сравнению с первым. Он позволяет не только воспроизводить файлы AVI, но и получить к ним неограниченный доступ: расширенная информация о файле, работа с отдельными кадрами изображения, а также поддержка любых совместимых с Windows кодеков (компрессоров и декомпрессоров). Использование набора функций VFW позволяет не только создать полноценный видеоплеер, но и профессиональный редактор для обработки файлов в формате AVI.

Для организации воспроизведения видеофайла необходимы следующие базовые функции и макрокоманды:

- `AVIFileInit`

  Предназначена для загрузки и инициализации набора функций VFW;

- `AVIFileExit`

  Освобождает ресурсы системы, задействованные для VFW. Данную функцию необходимо вызвать перед завершением программы. На каждый вызов функции `AVIFileInit` должна быть вызвана функция `AVIFileExit`;

- `AVIStreamOpenFromFile`

  Позволяет открыть новый поток для указанного файла AVI;

- `AVIStreamRelease`

  Закрывает указанный поток для AVI-файла и освобождает используемые ресурсы;

- `AVIStreamGetFrameOpen`

  Извлекает первый кадр из потока и подготавливает (проводит декомпрессию) его к выводу на экран;

- `AVIStreamLength`

  Возвращает длину для указанного потока в выборках (сэмплах);

- `AVIStreamEndTime`

  Данная макрокоманда позволяет получить конечное время для указанного потока;

- `AVIStreamInfo`

  Читает информацию об указанном потоке. Каждый поток имеет область заголовка, в котором содержится различная информация: тип потока, размер кадра, размер одной выборки, длина потока и др.;

- `AVIStreamGetFrame`

  Возвращает адрес восстановленного (декомпрессированного) кадра. Кадр имеет формат упакованного *DIB*-изображения (Device-Independent Bitmap);

- `DrawDibDraw`

  Выводит растровое изображение на экран;

- `AVIStreamRead`

  Позволяет прочитать видео, аудио или другие данные из указанного потока;

- `DrawDibOpen`

  Загружает и инициализирует библиотеку для поддержки растровых изображений (DIB);

- `DrawDibClose`

  Освобождает ресурсы, занятые библиотекой обработки растровых изображений (DIB);

- `AVIStreamGetFrameClose`

  Освобождает ресурсы, выделенные для декомпрессии и подготовки кадров к выводу на экран.

А теперь на основе этих функций напишем собственный класс для просмотра видеоданных в формате AVI. Для удобства работы мы будем использовать средства библиотеки MFC. Это существенно упростит задачу и уменьшит размер конечного кода. Назовем наш класс `CAVI` и создадим его, как показано в листинге 5.6.

Листинг 5.6. Файл AVI.h класса `CAVI`

```
// Файл AVI.h: interface for the CAVI class.
#if !defined(AFX_AVI_H__15F37CE6_8FCC_4F5A_9D1A_92BCBFBB2D2F__INCLUDED_)
#define AFX_AVI_H__15F37CE6_8FCC_4F5A_9D1A_92BCBFBB2D2F__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// подключим файл объявлений для библиотеки VFW
#include "vfw.h"
// сообщение для таймера
#define TIMER_UPDATE     WM_USER + 2000
// класс CAVI
class CAVI : public CWnd
{
public:
       // конструктор и деструктор по умолчанию
       CAVI ();
       virtual ~CAVI ();
// общедоступные функции класса CAVI
       // функция для загрузки AVI-файла
       int Load ( LPCTSTR lpszAVIFile );
       // функция для установки цвета фона AVI-файла
       void SetColorMask ( COLORREF color );
       // функции управления воспроизведением
       BOOL Play ();
       BOOL Stop ();
// защищенная область класса
protected:
// функции для обработки сообщений класса
// { { AFX_MSG ( CAVI )
       afx_msg void OnDestroy ();
       afx_msg void OnPaint ();
// } } AFX_MSG
```

```
        DECLARE_MESSAGE_MAP ()
// закрытая область класса
private:
        // функция для инициализации VFW
        void InitializeAVI ();
        // функция для освобождения ресурсов VFW
        void FreeAVI ();
        // функция вывода очередного кадра на экран
        void ShowFrame ( CDC* pDC );
        // функция для загрузки маски кадра
        void SetFreeScreen ( CDC* pInDC, CDC* pOutDC, int x, int y,
                             int icx, int icy );
        // функция обратного вызова для обработки событий таймера
        static void CALLBACK UpdatePlay (HWND hWnd, UINT uMsg,
                                         UINT uEvent, DWORD dwTime );
        // переменные для хранения растровых изображений
        CBitmap m_BMPBGR;
        CBitmap* m_BMPTEMP;
        // переменная для хранения главного контекста вывода
        CDC m_DC;
        // переменная для хранения цвета маски
        COLORREF m_ColorMask;
        // переменная для хранения растрового изображения
        HDRAWDIB m_DIB;
        // переменные для хранения размеров изображения
        int m_iHeight;
        int m_iWidth;
        int m_iY;
        int m_iX;
        // переменная для хранения времени таймера
        UINT m_uTimer;
        // переменная для хранения полного числа фреймов в AVI-файле
        LONG m_CountFrame;
        // переменная для хранения текущего фрейма в AVI-файле
        UINT m_uCurFrame;
        // переменная для хранения текущего состояния воспроизведения
        BOOL m_bPlay;
        // указатель на открытый поток видео в AVI-файле
        PAVISTREAM m_pStream;
        // переменная для хранения указателя на фрейм потока видео
        PGETFRAME m_pFrame;
        // информационная структура для видеопотока
        AVISTREAMINFO AVIinfo;
};
```

Теперь напишем файл реализации класса CAVI так, как показано в листинге 5.7.

Листинг 5.7. Файл AVI.cpp класса CAVI

```
// ////////////////////////////////////////////////////////////////////
// Файл AVI.cpp: implementation of the CAVI class.
#include "stdafx.h"
#include "Video.h"
#include "AVI.h"
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] =__FILE__;
#define new DEBUG_NEW
#endif
// конструктор класса
CAVI::CAVI ()
{
    // инициализируем переменные класса
    InitializeAVI ();
    // инициализируем VFW
    AVIFileInit ();
    // инициализируем библиотеку поддержки DIB
    m_DIB = DrawDibOpen ( );
}
// деструктор класса
CAVI :: ~CAVI ( )
{
    // освобождаем библиотеку DIB
    if (m_DIB) DrawDibClose ( m_DIB );
    // освобождаем ресурсы VFW
    FreeAVI ( );
    // выгружаем библиотеку VFW
    AVIFileExit ( );
}
// карта сообщений класса CAVI
BEGIN_MESSAGE_MAP ( CAVI, CWnd )
    ON_WM_DESTROY ( )
    ON_WM_PAINT ( )
END_MESSAGE_MAP ( )
// функция инициализации
void CAVI :: InitializeAVI ( )
{
    m_pStream   = NULL;
```

```
    m_pFrame    = NULL;
    m_bPlay     = FALSE;
    m_BMPTEMP   = NULL;
    m_uCurFrame = 0;
    m_iX        = 0;
    m_iY        = 0;
}
// функция освобождения ресурсов
void CAVI :: FreeAVI ()
{
    // если идет процесс воспроизведения, останавливаем его
    if ( m_bPlay ) Stop ();
    // освобождаем ресурсы поддержки кадров
    if ( m_pFrame )
     {
       AVIStreamGetFrameClose ( m_pFrame );
       m_pFrame = NULL;
     }
    // освобождаем ресурсы видеопотока
    if ( m_pStream ) AVIStreamRelease ( m_pStream );
    // удаляем объект изображения
    if ( m_BMPBGR.m_hObject ) m_BMPBGR.DeleteObject ();
    // освобождаем главный контекст вывода
    if ( m_DC.m_hDC ) m_DC.DeleteDC ();
}
// функция вывода очередного кадра на экран
void CAVI :: ShowFrame ( CDC* pDC )
{
    LPBITMAPINFOHEADER lpBitmap;
    CBitmap BMP, *tempBMP;
    RECT rect;
    int H = 0, W = 0;
    CDC dc;
    // получаем размеры клиентской области окна
    GetClientRect ( &rect );
    // вычисляем высоту и ширину клиентской области окна
    H = rect.bottom — rect.top;
    W = rect.right — rect.left;
    // получаем из видеопотока очередной фрейм
    lpBitmap = (LPBITMAPINFOHEADER) AVIStreamGetFrame ( m_pFrame,
                                                  (LONG) m_uCurFrame );
    // создаем на его основе совместимый контекст изображения
    if ( lpBitmap )
```

```
    {
      dc.CreateCompatibleDC ( pDC );
      BMP.CreateCompatibleBitmap ( pDC, W, H );
      tempBMP = dc.SelectObject ( &BMP );
      // копируем полученное изображение в основной контекст
      dc.BitBlt ( 0, 0, W, H, &m_DC, 0, 0, SRCCOPY );
      // выводим изображение на экран
      DrawDibDraw ( m_DIB, dc.GetSafeHdc (), rect.left + m_iX,
rect.top + m_iY, m_iWidth, m_iHeight, lpBitmap, NULL, 0, 0, -1, -1, 0 );
      // обрабатываем маску
      SetFreeScreen ( &dc, pDC, rect.left, rect.top, W, H );
      // выбираем объект
      dc.SelectObject ( tempBMP );
    }
}
// функция обработки маски кадра
void CAVI :: SetFreeScreen ( CDC* pInDC, CDC* pOutDC, int x, int y,
                             int icx, int icy )
{
    CBitmap BMP, *tempBMP;
    CBitmap copyBMP, *tempCopyBMP;
    CDC dc, copyDC;
    // создаем совместимые контексты для изображений
    dc.CreateCompatibleDC ( pInDC );
    BMP.CreateBitmap ( icx, icy, 1, 1, NULL );
    tempBMP = dc.SelectObject ( &BMP );
    copyDC.CreateCompatibleDC ( pOutDC );
    copyBMP.CreateCompatibleBitmap ( pOutDC, icx, icy );
    tempCopyBMP = copyDC.SelectObject ( &copyBMP );
    // копируем данные из одного контекста в другой
    copyDC.BitBlt ( 0, 0, icx, icy, &m_DC, 0, 0, SRCCOPY );
    // устанавливаем фон изображения
    pInDC->SetBkColor ( m_ColorMask );
    dc.BitBlt ( m_iX, m_iY, m_iWidth, m_iHeight, pInDC, m_iX, m_iY,
                SRCCOPY );
    // устанавливаем цвет фона и текста для исходного контекста
    pInDC->SetBkColor ( RGB ( 0, 0, 0 ) );
    pInDC->SetTextColor ( RGB ( 255, 255, 255 ) );
    pInDC->BitBlt ( m_iX, m_iY, icx, icy, &dc, m_iX, m_iY, SRCAND );
    // устанавливаем цвет для контекста маски
    copyDC.SetBkColor ( RGB ( 255, 255, 255 ) );
    copyDC.SetTextColor ( RGB ( 0, 0, 0 ) );
    copyDC.BitBlt ( m_iX, m_iY, m_iWidth, m_iHeight, &dc, m_iX, m_iY,
                    SRCAND );
    copyDC.BitBlt ( 0, 0, icx, icy, pInDC, 0, 0, SRCPAINT );
```

```
    // копируем данные маски в выходной контекст
    pOutDC->BitBlt ( x, y, icx, icy, &copyDC, 0, 0, SRCCOPY );
// выбираем полученные объекты изображений
    copyDC.SelectObject ( tempCopyBMP );
    dc.SelectObject ( tempBMP );
}
// функция обратного вызова для обработки событий таймера
void CALLBACK CAVI :: UpdatePlay ( HWND hWnd, UINT, UINT, DWORD )
{
    CAVI* pAVI = ( CAVI* ) CWnd :: FromHandle ( hWnd );
    // если указатель пустой, выходим
    if ( pAVI == NULL ) return;
    // если достигнут конец видеоданных, останавливаем игру
    if ( ++ pAVI->m_uCurFrame >= ( UINT ) pAVI->m_CountFrame )
     {
       // обнуляем номер текущего фрейма
       pAVI->m_uCurFrame = 0;
       pAVI->Stop ();
       return;
     }
    // выводим следующий кадр изображения
    pAVI->Invalidate ();
}
// обработчик сообщения WM_DESTROY
void CAVI :: OnDestroy ()
{
    Stop ();
}
// обработчик сообщения WM_PAINT
void CAVI :: OnPaint ()
{
    // создаем контекст для рисования и выводим очередной кадр
    CPaintDC dc ( this );
    if ( m_pStream ) ShowFrame ( &dc );
}
// функция для загрузки файла в формате AVI
int CAVI :: Load ( LPCTSTR lpszAVIFile )
{
    CClientDC dc ( this );
    RECT rect;
    LONG lLenghtAVI = 0L;
    int x = 0, y = 0;
    int iCenterX = 0, iCenterY = 0;
    int H = 0, W = 0;
```

```
// обнуляем поля структуры AVISTREAMINFO
memset ( &AVIinfo, 0, sizeof ( AVISTREAMINFO ) );
// если видеопоток открыт, закрываем его
if ( m_pStream )
 {
   FreeAVI ();
   InitializeAVI ();
 }
// получаем размеры клиентской области окна
GetClientRect ( &rect );
// вычисляем высоту и ширину клиентской области окна
H = rect.bottom — rect.top;
W = rect.right — rect.left;
// открываем видеопоток
if ( AVIStreamOpenFromFile ( &m_pStream, lpszAVIFile,
     streamtypeVIDEO, 0, OF_READ | OF_SHARE_EXCLUSIVE, NULL ) )
 {
   m_pStream = NULL;
   return -1L; // не удалось открыть поток
 }
// определяем полное число фреймов в потоке
m_CountFrame = AVIStreamLength ( m_pStream );
// подготавливаем обработчик фреймов
m_pFrame = AVIStreamGetFrameOpen ( m_pStream, NULL );
// получаем время конца потока
lLenghtAVI = AVIStreamEndTime ( m_pStream );
// вычисляем время смены кадров
m_uTimer = ( UINT ) ( lLenghtAVI / m_CountFrame );
// получаем информацию о потоке
AVIStreamInfo ( m_pStream, &AVIinfo, sizeof ( AVISTREAMINFO ) );
// вычисляем размер видеоизображения
x = AVIinfo.rcFrame.right — AVIinfo.rcFrame.left;
y = AVIinfo.rcFrame.bottom — AVIinfo.rcFrame.top;
// вычисляем координаты для центровки изображения
iCenterX = ( ( W > x ) ? ( ( W — x ) / 2 ) : 0 );
iCenterY = ( ( H > y ) ? ( ( H — y ) ) / 2 : 0 );
// сохраняем полученные значения
m_iX = iCenterX;
m_iY = iCenterY;
m_iWidth = x;
m_iHeight = y;
// создаем совместимый контекст для вывода
m_DC.CreateCompatibleDC ( &dc );
```

```
    m_BMPBGR.CreateCompatibleBitmap ( &dc, W , H );
    m_BMPTEMP = m_DC.SelectObject ( &m_BMPBGR );
    // закрашиваем клиентскую область стандартным цветом
    m_DC.FillSolidRect ( &rect, ::GetSysColor ( COLOR_WINDOW ) );
}
// функция для установки цвета маски
void CAVI :: SetColorMask ( COLORREF color )
{
    m_ColorMask = color;
}
// функция для запуска процесса воспроизведения
BOOL CAVI :: Play ( )
{
    // если воспроизведение уже идет, выходим
    if ( m_bPlay )
     {
       m_uCurFrame = 0;
       return FALSE;
     }
    // выводим на экран первый кадр
    Invalidate ( );
    // запускаем таймер
    ::SetTimer ( GetSafeHwnd ( ), TIMER_UPDATE, m_uTimer,
                ( TIMERPROC ) UpdatePlay );
    // устанавливаем состояние воспроизведения
    m_bPlay = TRUE;
    return TRUE;
}
// функция для остановки текущего воспроизведения
BOOL CAVI :: Stop ()
{
    // если воспроизведение уже остановлено, выходим
    if ( !m_bPlay ) return FALSE;
    // останавливаем таймер
    ::KillTimer ( GetSafeHwnd (), TIMER_UPDATE );
    // устанавливаем состояние воспроизведения
    m_bPlay = FALSE;
    return TRUE;
}
```

Теперь, когда класс для воспроизведения AVI-файлов готов, создадим тестовое приложение для проверки его работы. Создайте новый проект с поддержкой библиотеки MFC (MFC AppWizard (exe)). Введите название проекта, на-

пример Video. В первом окне мастера выберите диалоговое приложение (**Dialog based**). В следующем окне мастера оставьте установленным только флажок **3D controls**. Далее нажимайте кнопки **Next** и **Finish**, пока мастер не создаст каркас программы. В редакторе ресурсов добавьте на диалоговое окно кнопки **Load File**, **Play**, **Stop** и удалите кнопку **OK**. В качестве окна для вывода изображения используйте элемент управления **Static Text**. Присвойте ему уникальный идентификатор, например, `IDC_VIDEO`. Именно в этой рамке и будет осуществляться вывод видеоданных. Кнопка **Load File** позволит выбрать файл AVI на диске и загрузить его в программу.

Скопируйте файлы класса `CAVI` в папку вашего тестового проекта. В файл объявлений для диалогового окна (например, VideoDlg.h) запишите ссылку на класс `CAVI`, как показано в листинге 5.8.

Листинг 5.8. Файл VideoDlg.h

```
// перед объявлением класса добавьте ссылку на файл CAVI.H
#include "AVI.h"
// далее следует каркас класса CVideoDlg dialog
class CVideoDlg : public CDialog
{
// Construction
public:
      CvideoDlg ( CWnd* pParent = NULL ); // standard constructor
// Dialog Data
      // { { AFX_DATA ( CvideoDlg )
      enum { IDD = IDD_VIDEO_DIALOG };
          // NOTE: the ClassWizard will add data members here
      // } } AFX_DATA
      // ClassWizard generated virtual function overrides
      // { { AFX_VIRTUAL ( CvideoDlg )
      protected:
      // DDX/DDV support
      virtual void DoDataExchange ( CDataExchange* pDX );
      // } } AFX_VIRTUAL
// Implementation
protected:
      HICON m_hIcon;
      // Generated message map functions
      // { { AFX_MSG ( CvideoDlg )
      virtual BOOL OnInitDialog ();
      afx_msg void OnPaint ();
      afx_msg HCURSOR OnQueryDragIcon ();
```

```
      afx_msg void OnPlay ();
       afx_msg void OnStop ();
       afx_msg void OnOpen ();
       // } } AFX_MSG
       DECLARE_MESSAGE_MAP ()
private:
       // объявим наш класс CAVI
       CAVI m_AVI;
};
```

Файл VideoDlg.cpp генерируется мастером автоматически и имеет стандартный для диалогового приложения вид. Полный листинг этого файла можно найти на диске, прилагаемом к книге.

Добавьте в функцию инициализации диалогового окна (в файле VideoDlg.cpp) код, как показано в листинге 5.9.

Листинг 5.9. Функция `OnInitDialog()`

```
BOOL CvideoDlg :: OnInitDialog ()
{
       Cdialog :: OnInitDialog ();
       // Здесь определяем значок для диалогового окна
       SetIcon ( m_hIcon, TRUE ); // Set big icon
       SetIcon ( m_hIcon, FALSE ); // Set small icon
       // добавляем обработку событий через класс CWnd
       m_AVI.SubclassDlgItem ( IDC_VIDEO, this );
       return TRUE;
}
```

После этого следует написать код для обработки нажатий кнопок **Play**, **Stop** и **Load File** так, как это сделано в листинге 5.10.

Листинг 5.10. Обработка нажатий управляющих кнопок

```
// функция для открытия и загрузки файла AVI
void CvideoDlg :: OnOpen ()
{
    CFileDialog dlg ( TRUE, NULL, "*.avi", OFN_HIDEREADONLY |
                OFN_FILEMUSTEXIST, "AVI Files ( *.avi )|*.avi ||", this );
    // выводим на экран диалоговое окно для открытия файлов
    if ( IDOK == dlg.DoModal () )
     {
```

```
      // получаем имя выбранного файла
      m_AVI.Load (dlg.GetPathName () );
      // устанавливаем цвет маски AVI-файла, например черный
      m_AVI.SetColorMask ( RGB ( 0, 0, 0 ) );
     }
}
// обработка нажатия кнопки Play
void CvideoDlg :: OnPlay ()
{
    m_AVI.Play ();
}
// обработка нажатия кнопки Stop
void CvideoDlg :: OnStop ()
{
    m_AVI.Stop ();
}
```

Функция `SetColorMask` позволяет установить цвет маски изображения. Делается это для того, чтобы фон окна вывода совпадал с фоном видеопотока. Если для вас это не принципиально, то данную функцию можно не использовать. Скомпилируйте программу и проверьте ее в работе. Сразу замечу, что просматривать можно только стандартные AVI-файлы. Если же вы захотите работать с любыми AVI-файлами, следует в класс `CAVI` добавить набор функций для декомпрессии видеоданных с использованием различных кодеков, установленных в системе. Как правило, для этого достаточно всего нескольких базовых функций и макрокоманд:

- `ICDecompressOpen`

  Данная макрокоманда открывает декомпрессор, совместимый с указанным форматом;

- `ICDecompress`

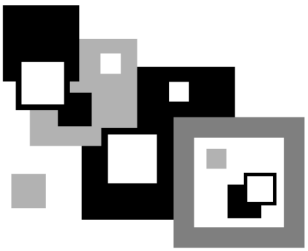
  Данная функция производит декомпрессию одного фрейма;

- `ICClose`

  Эта функция служит для закрытия компрессора и декомпрессора.

Программирование функций компрессии AVI-файлов выходит за рамки данной книги и не будет здесь рассматриваться. Если же вас интересует данная тема, то ознакомьтесь с документацией, предоставляемой фирмой Microsoft.

ГЛАВА 6



Звуковая карта

Появление звуковых карт по праву может считаться началом создания полноценных мультимедийных компьютеров. Если раньше для извлечения звуков использовался системный динамик (спикер), который звучал на уровне пустой консервной банки, то сейчас звуковой модуль представляет собой сложнейшее аналого-цифровое устройство. В состав современных звуковых карт непременно входят следующие компоненты:

- *DSP* (Digital Sound Processor) — *звуковой процессор*;
- *Mixer* — *микшер*;
- интерфейс *MIDI* (Musical Instrument Digital Interface).

Каждый из этих компонентов имеет собственные порты управления и отвечает за конкретные задачи. К сожалению, звуковые карты не имеют жестко закрепленных портов ввода-вывода, что создает определенные проблемы не только программистам, но и настройщикам компьютерного оборудования. В ранних версиях Windows надо вручную устанавливать параметры звуковой платы, исходя из общей конфигурации оборудования. Для этого в файл autoexec.bat добавляется строка следующего вида:

```
SET BLASTER=A220 I5 D1 [ H5 M220 P330 ]
```

Здесь:

- `A` — номер порта ввода-вывода;
- `I` — номер выделенного прерывания;
- D — номер выбранного 8-битного канала *DMA* (Direct Memory Access — прямой доступ к памяти);
- `H` — номер выбранного 16-битного канала DMA;
- `M` — номер порта ввода-вывода для микшера;

- P — номер порта ввода-вывода для MPU-401 (это один из режимов работы MIDI-интерфейса).

Весь рассматриваемый в этой главе материал ориентирован на использование с перечисленными далее следующими типами звуковых карт:

- Sound Blaster версии 1.5 (SB 1.5);
- Sound Blaster версии 2.0 (SB 2.0);
- Sound Blaster версии 2.0 с подключением CD (SB 2.0CD);
- Sound Blaster Pro (SBPRO);
- Sound Blaster 16 (SB 16);
- Sound Blaster 16 с расширенным сигнальным процессором (ASP).

Еще раз повторю, что программирование звуковых карт несколько сложнее других устройств, поскольку нет единого стандарта.

В общем, как и ранее, представим весь материал по звуковым картам в виде двух отдельных тем:

- С использованием аппаратных портов.
- С использованием интерфейса Win32.

6.1. Использование портов

Поскольку современная звуковая плата состоит из нескольких модулей, мы рассмотрим программирование каждого из них в отдельности. Сегодня наиболее популярны у пользователей звуковые платы фирмы Creative. Благодаря современным технологиям, этой фирме первой удалось выпустить на рынок относительно недорогие модели звуковых плат, поддерживающие не только высококачественный звук, но и имеющие потрясающие возможности MIDI для профессиональной работы с музыкой.

Основой звуковых плат является звуковой процессор (DSP), позволяющий записывать и воспроизводить звуковой сигнал с различных источников: цифровых и аналоговых. Как правило, он поддерживает 8- и 16-разрядный (и более) цифровой звук, работает в моно- и стереорежимах, дуплексном режиме (воспроизведение и запись сигнала одновременно). Кроме этого, полностью реализована декомпрессия (модуляция *ADPCM* — Adaptive Differential Pulse Code Modulation) звука в реальном времени.

Встроенный модуль микшера управляет уровнем громкости доступных каналов, позволяет выбирать активный канал для записи и воспроизведения звука. Современные микшеры поддерживают достаточное количество каналов: микрофонные, цифровые, линейные.

Интерфейс MIDI интересен в первую очередь музыкантам, поскольку позволяет писать на компьютере полноценную партитуру для своего произведения, а также подключать внешние устройства (например, синтезатор). Правда, чтобы подключить внешний музыкальный инструмент или какой-либо модуль, понадобится дополнительный переходник. Это необходимо для правильной развязки сигналов по питанию. Думаю, музыкантам это известно и так, а нам это вряд ли пригодится. Замечу только, что интерфейс MIDI использует общий разъем с джойстиком (15 контактов).

Итак, рассмотрим программирование следующих основных модулей звуковой платы:

- цифрового процессора (DSP);
- микшера;
- интерфейса MIDI.

6.1.1. Цифровой процессор

Цифровой процессор или иначе DSP (Digital Sound Processor) выполняет основную часть обработки звукового сигнала. Для ускорения работы DSP может задействовать контроллер прямого доступа к памяти (DMA). DSP используется для следующих операций:

- записи и воспроизведения моно- или стереозвука с разрядностью 8 и 16 бит;
- управления частотой дискретизации;
- в режиме эмуляции Sound Blaster для декомпрессии ADPCM по формулам 4:1, 3:1 и 2:1;
- поддержки расширенной обработки сигнала (ASP) для Sound Blaster 16;
- управления ЦАП спикера;
- управления режимами передачи данных посредством каналов DMA.

Для доступа к DSP используются следующие базовые порты:

- `2x6h` — используется только для записи; позволяет выполнить программный сброс процессора и установить заданные по умолчанию параметры;
- `2xAh` — используется только для чтения; позволяет читать данные из DSP;
- `2xCh` — используется для операций записи и чтения; в режиме записи является командным и позволяет посылать DSP различные управляющие команды или данные; в режиме чтения через этот порт считывается состояние готовности DSP к приему команды;
- `2xEh` — используется только для чтения; позволяет проверить готовность DSP для передачи данных.

Символ x (икс) указывает на возможные варианты адресации портов для реального компьютера: 210h, 220h, 230h, 240h, 250h, 260h и 280h. Например, если для звуковой платы выбран порт 220h, то управлять DSP можно через порт 226h. А теперь разберем работу с этими портами подробнее.

6.1.1.1. Порт *2x6h*

Позволяет сбросить DSP и инициализировать значения по умолчанию. Следует обязательно выполнять эту операцию перед началом работы с процессором. Процедура сброса должна соответствовать определенной последовательности действий:

1. В порт 2x6h записывается число 1 и выдерживается пауза минимум 3 мкс.
2. В порт 2x6h записывается число 0.
3. Проверяется порт 2xEh для подтверждения полученных данных. Если бит 7 будет установлен в 1, данные получены.
4. Опрашивается порт 2xAh. Если значение равно 0AAh, значит, DSP успешно сброшен, иначе можно предположить, что DSP отсутствует. Время опроса должно быть не менее 100 мкс.

Рассмотрим пример кода для инициализации DSP через порт 226h (листинг 6.1).

Листинг 6.1. Инициализация DSP

```
// пример функции для инициализации DSP
bool ResetDSP ( int iBasePort )
{
    DWORD dwResult  = 0; // переменная для хранения результата
    int iTimeWait = 65535;
    // записываем в порт 1
    outPort ( iBasePort + 6, 0x01, 1 );
    // задержка 1 мс
    Sleep ( 1 );
    // записываем в порт 0
    outPort ( iBasePort + 6, 0x00, 1 );
    // проверяем наличие данных и результат операции
    while ( -- iTimeWait > 0 )
     {
       // читаем состояние порта 2xEh
       inPort ( iBasePort + 0x0E, &dwResult, 1 );
       if ( ( dwResult & 0x80 ) == 0x01 )
       {
```

```
          // читаем порт 2xAh
          for ( int i = 65000; i > 0; i -- )
           {
             inPort ( iBasePort + 0x0A, &dwResult, 1 );
             if ( dwResult == 0xAA ) return true; // DSP найден
           }
       }
     }
    // DSP не обнаружен
    return false;
}
```

6.1.1.2. Порт *2xAh*

Используется только для чтения данных из DSP. Перед тем как прочитать данные из этого порта, следует убедиться, что бит 7 в порту состояния (2xEh) установлен в 1. Это гарантирует наличие данных в 2xAh, после чего их можно прочитать. Базовый пример работы с портом 22Ah показан в листинге 6.2.

Листинг 6.2. Чтение данных из порта 22Ah

```
// функция для чтения данных из DSP
unsigned char ReadDSP ( int iBasePort )
{
    DWORD dwResult  = 0; // переменная для хранения результата
    int iTimeWait = 50000;
    while ( -- iTimeWait > 0 )
     {
        // читаем состояние порта 2xEh
        inPort ( iBasePort + 0x0E, &dwResult, 1 );
        if ( ( dwResult & 0x80 ) == 0x01 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return NO_TIME;
     }
    // читаем данные из порта 22Ah
    dwResult = 0;
    inPort ( iBasePort + 0x0A, &dwResult, 1 );
    return (unsigned char) dwResult;
}
```

6.1.1.3. Порт *2xCh*

В режиме записи позволяет записать управляющую команду DSP и дополнительные аргументы, а в режиме чтения проверяет готовность DSP к приему

команды. Если бит 7 сброшен в 0, то значит DSP готов к приему команд или данных. Рассмотрим пример записи команды E1h в порт 22Ch.

Листинг 6.3. Запись данных в порт 22Ch

```
// функция для записи данных в DSP
void WriteDSP ( int iBasePort, unsigned char uValue )
{
    DWORD dwResult  = 0; // переменная для хранения результата
    int iTimeWait = 50000;
    while ( -- iTimeWait > 0 )
     {
        // читаем состояние порта 2xEh
        inPort ( iBasePort + 0x0C, &dwResult, 1 );
        if ( ( dwResult & 0x80 ) == 0x00 ) break;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return;
     }
    // записываем команду в порт 22Ch
    outPort ( iBasePort + 0x0C, uValue, 1 );
}
// вызываем нашу функцию для записи команды E1h в порт
WriteDSP ( 0x220, 0xE1 );
```

Процессор DSP поддерживает набор управляющих команд. Все они передаются через порт 2xCh. Многие из них имеют дополнительные аргументы. Список команд показан в табл. 6.1.

Таблица 6.1. *Команды DSP*

Код команды	Описание
10h	Прямое воспроизведение 8-битного цифрового звука
14h	Воспроизведение 8-битного цифрового звука через канал DMA
16h	Воспроизведение 2-битного звука с использованием метода компрессии звука ADPCM через канал DMA
17h	Воспроизведение 2-битного звука с использованием ADPCM через канал DMA с пустым байтом
1Ch	Воспроизведение 8-битного цифрового звука через канал DMA с автоинициализацией
1Fh	Воспроизведение 2-битного звука с использованием ADPCM через канал DMA с автоинициализацией

Таблица 6.1 *(продолжение)*

Код команды	Описание
20h	Прямое чтение 8-битного цифрового звука
24h	Чтение 8-битного цифрового звука через канал DMA
2Ch	Чтение 8-битного цифрового звука через канал DMA с автоинициализацией
30h	Чтение в режиме MIDI
31h	Чтение в режиме MIDI с использованием прерывания
34h	Режим чтения и записи для UART (поллинг)
35h	Режим чтения и записи для UART с использованием прерывания
36h	Режим чтения и записи для UART (поллинг с корректировкой по времени)
37h	Режим чтения и записи для UART с использованием прерывания и корректировкой по времени
38h	Запись в режиме MIDI
40h	Установка временной константы для передачи звуковых данных
41h	Установка частоты дискретизации для воспроизведения
42h	Установка частоты дискретизации для записи
48h	Установка размера блока данных
74h	Воспроизведение 4-битного звука с использованием ADPCM через канал DMA
75h	Воспроизведение 4-битного звука с использованием ADPCM через канал DMA с пустым байтом
76h	Воспроизведение 3-битного звука с использованием ADPCM через канал DMA
77h	Воспроизведение 3-битного звука с использованием ADPCM через канал DMA с пустым байтом
7Dh	Воспроизведение 4-битного звука с использованием ADPCM через канал DMA с пустым байтом и автоинициализацией
7Fh	Воспроизведение 3-битного звука с использованием ADPCM через канал DMA с пустым байтом и автоинициализацией
80h	Установка паузы для ЦАП ввода-вывода
90h	Ускоренное воспроизведение 8-битного цифрового звука через канал DMA с автоинициализацией
91h	Ускоренная запись 8-битного цифрового звука через канал DMA
98h	Ускоренная запись 8-битного цифрового звука через канал DMA с автоинициализацией
A0h	Установка монорежима для записи
A8h	Установка стереорежима для записи

Таблица 6.1 *(окончание)*

Код команды	Описание
D0h	Приостановить 8-битный ввод-вывод через DMA
D1h	Включить динамик
D3h	Выключить динамик
D4h	Продолжить 8-битный ввод-вывод через DMA
D5h	Приостановить 16-битный ввод-вывод через DMA
D6h	Продолжить 16-битный ввод-вывод через DMA
D8h	Получить состояние динамика
D9h	Завершить 16-битный ввод-вывод через DMA с автоинициализацией
DAh	Завершить 8-битный ввод-вывод через DMA с автоинициализацией
E1h	Получить номер версии DSP

Разберем наиболее часто используемые команды DSP подробнее.

Команда *10h*

Эта команда поддерживает воспроизведение несжатого 8-битного звука. За один раз выводит один байт данных.

Использование:

1. Записать в порт 2xCh значение 10h.
2. Записать в порт байт данных.
3. Подождать определенное время и перейти к первому пункту.

Следует повторять эти действия, пока не будут переданы все байты данных.

Команда *14h*

Эта команда поддерживает воспроизведение 8-битных оцифрованных данных через установленный канал DMA. Она имеет два аргумента (младший и старший байты), определяющих полную длину проигрываемого участка данных минус единица.

Использование:

1. Установить обработчик прерывания.
2. Установить частоту дискретизации (команда 40h).
3. Записать в порт команду D1h.

4. Установить размер блока данных для DMA.
5. Записать в порт значение `14h` и аргументы.
6. Проверить порт `2xEh`.
7. Записать в порт команду `D3h`.

Команда *16h*

Эта команда поддерживает 2-битный звук с использованием ADPCM через канал DMA. В отличие от предыдущей команды, работает со сжатыми данными. Кроме кода команды, имеет два аргумента (младший и старший байты), определяющих полную длину передаваемых данных в байтах минус единица. В качестве первого байта данных должен быть передан пустой байт.

Команда *17h*

Данная команда поддерживает 2-битный звук с использованием ADPCM через канал DMA, но первый передаваемый байт должен быть пустым байтом.

Команда *20h*

Эта команда позволяет прямое чтение 8-битного цифрового звука из АЦП. Говоря проще, данная команда позволяет записать оцифрованный звук и сохранить его на диск.

Использование:

1. Записать в порт значение `20h`.
2. Считать из порта один байт.
3. Подождать определенное время и перейти к первому пункту.

Следует повторять эти действия, пока не будут считаны все данные.

Команда *30h*

Эта команда позволяет прочитать данные из порта MIDI.

Использование:

1. Записать в порт значение `30h`.
2. Опросить DSP на наличие данных MIDI.

Команда *38h*

Эта команда позволяет записать данные в порт MIDI.

Использование:

1. Записать в порт значение `38h`.
2. Записать в порт данные MIDI.

Команда *40h*

Эта команда позволяет установить значение временной константы, которая может быть высчитана так, как показано ниже:

Временная константа =
= 65 536 — 256 000 000 / (число каналов * частота дискретизации).

Для монорежима используется один канал, а для стерео — два канала.

Использование:

1. Записать в порт значение `40h`.
2. Записать в порт значение константы (используется только старший байт).

Команда *41h*

Эта команда позволяет установить частоту дискретизации для воспроизведения звуковых данных. Допустимые значения частоты лежат в диапазоне от 5000 до 45 000 Гц включительно.

Использование:

1. Записать в порт значение `41h`.
2. Записать в порт старший байт частоты.
3. Записать в порт младший байт частоты.

Команда *42h*

Эта команда позволяет установить частоту дискретизации для записи звуковых данных. Допустимые значения частоты лежат в диапазоне от 5000 до 45 000 Гц включительно.

Использование:

1. Записать в порт значение `42h`.
2. Записать в порт старший байт частоты.
3. Записать в порт младший байт частоты.

Команда *48h*

Эта команда позволяет установить размер одного блока для ускоренного режима передачи данных DMA. Измеряется в байтах минус единица.

Использование:

1. Записать в порт значение `48h`.
2. Записать в порт младший байт.
3. Записать в порт старший байт.

Команда *80h*

Эта команда позволяет установить продолжительность паузы. Измеряется в выборках минус единица. С помощью данной команды можно реализовать режим тишины — полного отсутствия уровня сигнала. Команда использует текущее значение частоты дискретизации.

Использование:

1. Записать в порт значение `80h`.
2. Записать в порт младший байт.
3. Записать в порт старший байт.

Команда *A0h*

Данная команда позволяет установить монорежим для записи. По умолчанию используется монорежим.

Команда *A1h*

Эта команда позволяет установить стереорежим для записи.

Команда *D1h*

Данная команда позволяет установить связь между цифровым выходом и входом усилителя. Для завершения команды необходимо максимум 112 мс.

Команда *D3h*

Данная команда позволяет разорвать связь между цифровым выходом и входом усилителя. Для завершения команды необходимо максимум 220 мс.

Команда *D8h*

Данная команда позволяет получить текущее состояние канала связи между цифровым выходом и входом усилителя. Если связи нет, то возвращается `00h`, а иначе значение `FFh`.

Команда *E1h*

Данная команда позволяет получить версию установленного процессора DSP. После выполнения команды будут получены два байта: первый определяет старший (major) номер версии, второй содержит младший номер (minor).

Рассмотрим несколько примеров работы с DSP. Напишем код для включения динамика (листинг 6.4).

Листинг 6.4. Включение динамика

```
// вызываем нашу функцию для записи команды D1h в порт
WriteDSP ( 0x220, 0xD1 );
```

А теперь попробуем установить частоту дискретизации для воспроизведения 44 100 Гц (0AC44h) так, как это сделано в листинге 6.5.

Листинг 6.5. Установка новой частоты дискретизации

```
// используем нашу функцию для записи значений в порт DSP
WriteDSP ( 0x220, 0x41 ); // код команды 41h
WriteDSP ( 0x220, 0xAC ); // старший байт частоты
WriteDSP ( 0x220, 0x44 ); // младший байт частоты
```

Для установки DSP в 16-битный режим воспроизведения и передачи блока данных размером 16 Кбайт можно использовать код из листинга 6.6.

Листинг 6.6. Установка 16-битного режима

```
// используем нашу функцию для записи значений в порт DSP
WriteDSP ( 0x220, 0xB0 ); // воспроизведение 16 бит
WriteDSP ( 0x220, 0x30 ); // стереорежим
WriteDSP ( 0x220, 0xFF ); // младший байт длины блока
WriteDSP ( 0x220, 0x03 ); // старший байт длины блока
```

Длина блока данных в обоих примерах рассчитана умножением числа 16 на 1024 минус единица. Результирующее значение равно 16 383 или 3FFFh в шестнадцатеричном исчислении. На этом завершим работу с процессором DSP и рассмотрим, как программируется микшер звуковой карты.

6.1.2. Микшер

Микшер, как уже было сказано ранее, предназначен для коммутации каналов, а также управления громкостью звука этих каналов. Для поддержки работы с микшером предназначены следующие порты:

- 2x4h — адресный порт; используется только для записи; позволяет выбрать номер управляющего регистра;
- 2x5h — порт данных; используется для чтения и записи; позволяет передавать и получать данные для указанного регистра.

Для работы с микшером необходимо записать в порт 2x4h номер управляющего регистра. После этого можно записывать или считывать данные из пор-

та 2x5h. При изменении отдельных битов управляющего регистра вначале следует прочитать его значение, а после этого изменить нужный разряд и записать значение в порт. Важно соблюдать это правило для совместимости с последующими расширениями. Список управляющих регистров показан в табл. 6.2.

Таблица 6.2. *Список регистров микшера*

Код регистра	Описание							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
00h	Сброс микшера							
04h	Громкость ЦАП в левом канале				Громкость ЦАП в правом канале			
0Ah	Резерв					Громкость микрофона		
22h	Общая громкость в левом канале				Общая громкость в правом канале			
26h	Громкость MIDI в левом канале				Громкость MIDI в правом канале			
28h	Громкость CD в левом канале				Громкость CD в правом канале			
2Eh	Громкость линейного входа в левом канале				Громкость линейного входа в правом канале			
30h	Общая громкость в левом канале					Резерв		
31h	Общая громкость в правом канале					Резерв		
32h	Громкость ЦАП в левом канале					Резерв		
33h	Громкость ЦАП в правом канале					Резерв		
34h	Громкость MIDI в левом канале					Резерв		
35h	Громкость MIDI в правом канале					Резерв		
36h	Громкость CD в левом канале					Резерв		
37h	Громкость CD в правом канале					Резерв		
38h	Громкость линейного входа в левом канале					Резерв		
39h	Громкость линейного входа в правом канале					Резерв		
3Ah	Громкость микрофона					Резерв		
3Bh	Громкость динамика ПК			Резерв				
3Ch	Резерв				Управление выходными каналами			
				ЛЛ	ЛП	АЛ	АП	М
3Dh	Резерв	Управление входным левым каналом						
		МЛ	МП	ЛЛ	ЛП	АЛ	АП	М

Таблица 6.2 *(окончание)*

Код регистра	Описание							
	7	6	5	4	3	2	1	0
3Eh	Резерв	Управление входным правым каналом						
		МЛ	МП	ЛЛ	ЛП	АЛ	АП	М
3Fh	УЛВХК	Резерв						
40h	УПВХК	Резерв						
41h	УЛВК	Резерв						
42h	УПВК	Резерв						
43h	Резерв							АПУ
44h	Уровень высоких частот в левом канале				Резерв			
45h	Уровень высоких частот в правом канале				Резерв			
46h	Уровень низких частот в левом канале				Резерв			
47h	Уровень низких частот в правом канале				Резерв			

Описание таблицы.

- Регистр `00h` предназначен для сброса микшера. После сброса все регистры микшера будут установлены в значения, используемые по умолчанию. Для выполнения сброса достаточно записать любое значение (размером в байт) в этот регистр.
- Регистр `04h` управляет уровнем громкости ЦАП в левом и правом каналах. Биты 7—4 отвечают за левый канал, а биты 3—0 — за правый. Имеется 16 уровней (от 0 до 15) громкости, где 0 соответствует значению –60 дБ, а 15 — 0 дБ с шагом в 4 дБ. По умолчанию уровень равен 12 (12 дБ).
- Регистр `0Ah` управляет уровнем громкости микрофона. Используются только биты 2—0. Имеется 8 уровней (от 0 до 7) громкости, где 0 соответствует значению –48 дБ, а 7 — 0 дБ с шагом в 6 дБ. По умолчанию уровень равен 0 (–48 дБ).
- Регистр `22h` управляет общим уровнем громкости в левом и правом каналах. Биты 7—4 отвечают за левый канал, а биты 3—0 — за правый. Имеется 16 уровней (от 0 до 15) громкости, где 0 соответствует значению –60 дБ, а 15 — 0 дБ с шагом в 4 дБ. По умолчанию уровень равен 12 (12 дБ).

- Регистр `26h` управляет уровнем громкости MIDI в левом и правом каналах. Биты 7—4 отвечают за левый канал, а биты 3—0 — за правый. Имеется 16 уровней (от 0 до 15) громкости, где 0 соответствует значению –60 дБ, а 15 — 0 дБ с шагом в 4 дБ. По умолчанию уровень равен 12 (12 дБ).
- Регистр `28h` управляет уровнем громкости CD аудио в левом и правом каналах. Биты 7—4 отвечают за левый канал, а биты 3—0 — за правый. Имеется 16 уровней (от 0 до 15) громкости, где 0 соответствует значению –60 дБ, а 15 — 0 дБ с шагом в 4 дБ. По умолчанию уровень равен 0 (–60 дБ).
- Регистр `2Eh` управляет уровнем громкости линейного входа в левом и правом каналах. Биты 7—4 отвечают за левый канал, а биты 3—0 — за правый. Имеется 16 уровней (от 0 до 15) громкости, где 0 соответствует значению –60 дБ, а 15 — 0 дБ с шагом в 4 дБ. По умолчанию уровень равен 0 (–60 дБ).
- Регистр `30h` управляет общим уровнем громкости в левом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 24 (–14 дБ).
- Регистр `31h` управляет общим уровнем громкости в правом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 24 (–14 дБ).
- Регистр `32h` управляет уровнем громкости ЦАП в левом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 24 (–14 дБ).
- Регистр `33h` управляет уровнем громкости ЦАП в правом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 24 (–14 дБ).
- Регистр `34h` управляет уровнем громкости MIDI в левом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 24 (–14 дБ).
- Регистр `35h` управляет уровнем громкости MIDI в правом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 24 (–14 дБ).

- Регистр 36h управляет уровнем громкости аудиодиска в левом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 0 (–62 дБ).
- Регистр 37h управляет уровнем громкости аудиодиска в правом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 0 (–62 дБ).
- Регистр 38h управляет уровнем громкости линейного входа в левом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 0 (–62 дБ).
- Регистр 39h управляет уровнем громкости линейного входа в правом канале. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 0 (–62 дБ).
- Регистр 3Ah управляет уровнем громкости микрофона. Используются только биты 7—3. Имеется 32 уровня (от 0 до 31) громкости, где 0 соответствует значению –62 дБ, а 31 — 0 дБ с шагом в 2 дБ. По умолчанию уровень равен 0 (–62 дБ).
- Регистр 3Bh управляет уровнем громкости динамика ПК. Используются только биты 7 и 6. Имеется 4 уровня (от 0 до 3) громкости, где 0 соответствует значению –18 дБ, а 3 — 0 дБ с шагом в 6 дБ. По умолчанию уровень равен 0 (–18 дБ).
- Регистр 3Ch управляет выходными каналами. Используются только биты 4—0: бит 4 — линейный левый канал (ЛЛ), 3 — линейный правый канал (ЛП), 2 — аудио CD левый (АЛ), 1 — аудио CD правый АП), 0 — микрофон (М). Установка бита в 0 позволяет открыть соответствующий канал. По умолчанию значения всех битов равны 1 (все закрыты).
- Регистр 3Dh управляет левыми входными каналами. Используются только биты 6—0: бит 6 — MIDI левый (МЛ), 5 — MIDI правый (МП), 4 — линейный левый (ЛЛ), 3 — линейный правый (ЛП), 2 — аудио CD левый (АЛ), 1 — аудио CD правый (АП), 0 — микрофон (М). Значения по умолчанию для битов 6, 5, 3 и 1 равны 0. Для битов 4, 2 и 0 значения по умолчанию равны 1. Установка бита в 0 позволяет открыть соответствующий канал. В режиме моно являются единственными каналами для ввода данных.
- Регистр 3Eh управляет правыми входными каналами. Используются только биты 6—0: бит 6 — MIDI левый (МЛ), 5 — MIDI правый (МП), 4 — ли-

нейный левый (ЛЛ), 3 — линейный правый (ЛП), 2 — аудио CD левый (АЛ), 1 — аудио CD правый (АП), 0 — микрофон (М). Значения по умолчанию для битов 6, 5, 3 и 1 равны 0. Для битов 4, 2 и 0 значения по умолчанию равны 1. Установка бита в 0 позволяет открыть соответствующий канал.

- Регистр `3Fh` управляет уровнем усиления в левом входном канале (УЛВХК). Используются только биты 7 и 6. Имеется 4 уровня (от 0 до 3) усиления, где 0 соответствует значению 0 дБ, а 3 — 18 дБ с шагом в 6 дБ. По умолчанию уровень равен 0 (0 дБ).
- Регистр `40h` управляет уровнем усиления в правом входном канале (УПВХК). Используются только биты 7 и 6. Имеется 4 уровня (от 0 до 3) усиления, где 0 соответствует значению 0 дБ, а 3 — 18 дБ с шагом в 6 дБ. По умолчанию уровень равен 0 (0 дБ).
- Регистр `41h` управляет уровнем усиления в левом выходном канале (УЛВК). Используются только биты 7 и 6. Имеется 4 уровня (от 0 до 3) усиления, где 0 соответствует значению 0 дБ, а 3 — 18 дБ с шагом в 6 дБ. По умолчанию уровень равен 0 (0 дБ).
- Регистр `42h` управляет уровнем усиления в правом выходном канале (УПВК). Используются только биты 7 и 6. Имеется 4 уровня (от 0 до 3) усиления, где 0 соответствует значению 0 дБ, а 3 — 18 дБ с шагом в 6 дБ. По умолчанию уровень равен 0 (0 дБ).
- Регистр `43h` управляет автоматической подстройкой уровня усиления (АПУ) микрофона. Используется только бит 0. Установка бита в 1 позволяет увеличить уровень на 20 дБ. По умолчанию бит равен 0 (0 дБ).
- Регистр `44h` управляет уровнем высоких частот в левом канале. Используются только биты 7—4. Имеется 16 уровней. Значения от 0 до 7 устанавливают соответственно значения уровня от –14 дБ до 0 дБ с шагом 2 дБ. Значения от 8 до 15 устанавливают соответственно значения уровня от 0 дБ до 14 дБ с шагом 2 дБ. По умолчанию используется значение 8 (0 дБ).
- Регистр `45h` управляет уровнем высоких частот в правом канале. Используются только биты 7—4. Имеется 16 уровней. Значения от 0 до 7 устанавливают соответственно значения уровня от –14 дБ до 0 дБ с шагом 2 дБ. Значения от 8 до 15 устанавливают соответственно значения уровня от 0 дБ до 14 дБ с шагом 2 дБ. По умолчанию используется значение 8 (0 дБ).
- Регистр `46h` управляет уровнем низких частот в левом канале. Используются только биты 7—4. Имеется 16 уровней. Значения от 0 до 7 устанавливают соответственно значения уровня от –14 дБ до 0 дБ с шагом 2 дБ.

Значения от 8 до 15 устанавливают соответственно значения уровня от 0 дБ до 14 дБ с шагом 2 дБ. По умолчанию используется значение 8 (0 дБ).

- Регистр `47h` управляет уровнем низких частот в правом канале. Используются только биты 7—4. Имеется 16 уровней. Значения от 0 до 7 устанавливают соответственно значения уровня от –14 дБ до 0 дБ с шагом 2 дБ. Значения от 8 до 15 устанавливают соответственно значения уровня от 0 дБ до 14 дБ с шагом 2 дБ. По умолчанию используется значение 8 (0 дБ).
- Регистр `80h` позволяет установить номер прерывания для звуковой карты. Используются только биты с 3 по 0: 3 бит — прерывание `IRQ 10`, 2 бит — `IRQ 7`, 1 бит — `IRQ 5`, 0 бит — `IRQ 2`. Следует помнить, что в один момент времени может быть установлен только один номер прерывания. Для этого нужно записать в соответствующий бит значение 1. Хотя этот регистр и управляет общими настройками всей платы, работать с ним нужно через порты микшера.
- Регистр `81h` позволяет установить номер канала DMA для звуковой карты. Как и предыдущий регистр, он управляет глобальными настройками звуковой платы, но адресуется через порты микшера. Формат данного регистра показан в табл. 6.3. Для выбора нужного канала следует установить соответствующий бит в 1. В один момент времени может быть установлен только один из трех 16- и 8-битных каналов DMA. Производители звуковых плат не рекомендуют изменять значения в регистрах `80h` и `81h`, а пользоваться специальными утилитами, входящими в комплект поставки звуковой платы. Кроме того, эти регистры могут использоваться только для плат PnP (Plug and Play).

Таблица 6.3. *Формат регистра 81h*

Бит	7	6	5	4	3	2	1	0
Канал DMA	DMA 7 (16 бит)	DMA 6 (16 бит)	DMA 5 (16 бит)	Резерв	DMA 3 (8 бит)	Резерв	DMA 1 (8 бит)	DMA 0 (8 бит)

- Регистр `82h` предназначен для получения состояния об аппаратном прерывании звуковой платы. Может применяться для работы со всеми модулями звуковой платы: процессором DSP, микшером и интерфейсом MPU-401. Работа с регистром выполняется через порты микшера. Регистр можно только читать. Информационными являются следующие биты: 2 — интерфейс MPU-401, 1— 16-битный канал DMA для ввода и вывода, 0 — 8-битный канал DMA для ввода и вывода. Если считываемый бит установлен в 1, то значит прерывание было вызвано соответствующим компонентом (MPU-401, DMA 16-бит или DMA 8-бит). Для подтверждения преры-

вания следует прочитать один из следующих портов: `2xEh` — для DMA 8-бит или SB-MIDI, `2xFh` — для 16-бит DMA, `3x0h` — для MPU-401.

Как видите, управлять микшером не очень сложно. Рассмотрим простой пример для установки новых значений уровня общей громкости для левого и правого каналов (листинг 6.7).

Листинг 6.7. Установка общего уровня громкости

```
// функция для записи данных в DSP
void SetMasterVolume ( DWORD dwValueL, DWORD dwValueR )
{
    DWORD dwValue = 0;
    // записываем номер регистра в порт 224h
    outPort ( 0x224, 0x30, 1 );
    dwValue = dwValueL << 3; // используются только биты с 7 по 3
    outPort ( 0x225, dwValue, 1 ); // левый канал
    // записываем номер регистра в порт 224h
    outPort ( 0x224, 0x31, 1 );
    dwValue = dwValueR << 3; // используются только биты с 7 по 3
    outPort ( 0x225, dwValue, 1 ); // левый канал
}
// установим общий уровень громкости -22 ДБ для обоих каналов
SetMasterVolume ( 20, 20 );
```

А теперь попробуем определить, каким модулем было вызвано последнее прерывание (листинг 6.8).

Листинг 6.8. Определение источника прерывания

```
// переменная для хранения результата
DWORD dwResult = 0;
// записываем номер регистра 82h в порт 224h
outPort ( 0x224, 0x82, 1 );
// читаем порт 225h
inPort ( 0x225, &dwResult, 1 );
// проверяем, чем вызвано прерывание
switch ( dwResult )
{
  case 1: // DMA 8 бит
   // выполняем какие-либо действия
   break;
  case 2: // DMA 16 бит
```

```
    // выполняем какие-либо действия
    break;
  case 4: // интерфейс MPU-401
    // выполняем какие-либо действия
    break;
}
```

Вся информация об устройстве микшера приведена для модуля CT1745 фирмы Creative. Если читатель программирует более ранние версии микшера, то ему следует полагаться на информацию, представленную в табл. 6.4 и 6.5.

Таблица 6.4. *Список регистров микшера CT1335*

Код регистра	Бит							
	7	6	5	4	3	2	1	0
00h	Сброс микшера							
02h	Резерв				Общий уровень громкости			Резерв
06h	Резерв				Уровень громкости MIDI			Резерв
08h	Резерв				Уровень громкости CD аудио			Резерв
0Ah	Резерв					Уровень громкости ЦАП		Резерв

Приведем краткое описание табл. 6.4.

- Регистр 00h позволяет сбросить микшер. После сброса все регистры микшера будут установлены в значения, используемые по умолчанию. Для выполнения сброса достаточно записать любое значение (размером в байт) в этот регистр.
- Регистр 02h управляет общим уровнем громкости всех каналов. Используются только 3—1 биты. Имеется 8 уровней (от 0 до 7) громкости, где 0 соответствует значению –46 дБ, а 7 — 0 дБ с нелинейным шагом в 4 дБ. По умолчанию уровень равен 4 (–11 дБ).
- Регистр 06h управляет уровнем громкости MIDI. Используются только 3—1 биты. Имеется 8 уровней (от 0 до 7) громкости, где 0 соответствует значению –46 дБ, а 7 — 0 дБ с нелинейным шагом в 4 дБ. По умолчанию уровень равен 4 (–11 дБ).
- Регистр 08h управляет уровнем громкости аудиодиска. Используются только 3—1 биты. Имеется 8 уровней (от 0 до 7) громкости, где 0 соответствует значению –46 дБ, а 7 — 0 дБ с нелинейным шагом в 4 дБ. По умолчанию уровень равен 0 (–46 дБ).

- Регистр `0Ah` управляет уровнем громкости ЦАП. Используются только 2 и 1 биты. Имеется 4 уровня (от 0 до 3) громкости, где 0 соответствует значению –46 дБ, а 3 — 0 дБ с нелинейным шагом в 7 дБ. По умолчанию уровень равен 0 (–46 дБ).

Таблица 6.5. *Список регистров микшера CT1345*

Код регистра	Бит							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
00h	Сброс микшера							
04h	Уровень громкости ЦАП для левого канала			Резерв	Уровень громкости ЦАП для правого канала			Резерв
0Ah	Резерв					M		Резерв
0Ch	Резерв		ВХФ	Резерв	Фильтр НЧ	Входной источник		Резерв
0Eh	Резерв		ВЫХФ	Резерв			Стерео	Резерв
22h	Общий уровень в левом канале			Резерв	Общий уровень в правом канале			Резерв
26h	Уровень MIDI в левом канале			Резерв	Уровень MIDI в правом канале			Резерв
28h	Уровень CD в левом канале			Резерв	Уровень CD в правом канале			Резерв
2Eh	Линейный левый канал			Резерв	Линейный правый канал			Резерв

Приведем краткое описание табл. 6.5.

- Регистр `00h` позволяет сбросить микшер. После сброса все регистры микшера будут установлены в значения, используемые по умолчанию. Для выполнения сброса достаточно записать любое значение (размером в байт) в этот регистр.
- Регистр `04h` управляет уровнем громкости ЦАП для левого (биты 7—5) и правого (биты 3—1) каналов. Имеется 8 уровней (от 0 до 7) громкости, где 0 соответствует значению –46 дБ, а 7 — 0 дБ с нелинейным шагом в 4 дБ. По умолчанию уровень равен 4 (–11 дБ).
- Регистр `0Ah` управляет уровнем громкости микрофона (биты 2 и 1). Имеется 4 уровня (от 0 до 3) громкости, где 0 соответствует значению –46 дБ, а 3 — 0 дБ с нелинейным шагом в 7 дБ. По умолчанию уровень равен 0 (–46 дБ).
- Регистр `0Ch` имеет следующее назначение: бит 5 управляет состоянием входного фильтра (ВХФ): 0 — включен и сигнал проходит через фильтр

нижних частот, 1 — выключен и сигнал идет в обход фильтра нижних частот. По умолчанию бит 5 равен 0. Бит 3 позволяет выбрать частоту для фильтра нижних частот, если бит 5 установлен в 0. Если бит 3 равен 0, будет выбрана частота 3,2 кГц, если 1 — 8,8 кГц. По умолчанию бит 3 равен 0. Биты 2 и 1 позволяют выбрать входной источник: 0 или 2 — микрофон, 1 — аудиодиск, 3 — линейный вход.

- Регистр `0Eh` имеет следующее назначение: бит 5 управляет состоянием выходного фильтра (ВЫХФ): 0 — включен и сигнал проходит через фильтр нижних частот, 1 — выключен и сигнал идет в обход фильтра нижних частот. По умолчанию бит 5 равен 0. Для стереорежима бит 5 следует установить в 1. Бит 1 управляет стереорежимом для выходного канала: 0 — моно, 1 — стерео. По умолчанию используется моно (0) режим.
- Регистр `22h` управляет общим уровнем громкости для левого (биты 7—5) и правого (биты 3—1) каналов. Имеется 8 уровней (от 0 до 7) громкости, где 0 соответствует значению –46 дБ, а 7 — 0 дБ с нелинейным шагом в 4 дБ. По умолчанию уровень равен 4 (–11 дБ).
- Регистр `26h` управляет уровнем громкости MIDI для левого (биты 7—5) и правого (биты 3—1) каналов. Имеется 8 уровней (от 0 до 7) громкости, где 0 соответствует значению –46 дБ, а 7 — 0 дБ с нелинейным шагом в 4 дБ. По умолчанию уровень равен 4 (–11 дБ).
- Регистр `28h` управляет уровнем громкости аудиодиска для левого (биты 7—5) и правого (биты 3—1) каналов. Имеется 8 уровней (от 0 до 7) громкости, где 0 соответствует значению –46 дБ, а 7 — 0 дБ с нелинейным шагом в 4 дБ. По умолчанию уровень равен 0 (–46 дБ).
- Регистр `2Eh` управляет уровнем громкости для левого (биты 7—5) и правого (биты 3—1) линейных каналов. Имеется 8 уровней (от 0 до 7) громкости, где 0 соответствует значению –46 дБ, а 7 — 0 дБ с нелинейным шагом в 4 дБ. По умолчанию уровень равен 0 (–46 дБ).

На этом можно завершить описание устройства микшера.

6.1.3. Интерфейс MIDI

Звуковая плата обычно поддерживает два основных интерфейса MIDI: MPU-401 и SB-MIDI.

В интерфейсе SB-MIDI используются те же порты ввода-вывода, что и для процессора DSP. Передача данных может выполняться в двух режимах: нормальном и UART (Universal Asynchronous Receiver/Transmitter — универсальный асинхронный режим приема и передачи данных). В нормальном режиме сначала записывается команда MIDI, а после нее данные. В режиме UART следует сразу послать DSP одну из команд разрешения работы с

UART (34h, 35h, 36h или 37h). После того как DSP будет переведен в режим UART, можно читать или писать MIDI-данные. После завершения работы следует послать DSP команду сброса, чтобы восстановить нормальный режим его работы. Для обнаружения MIDI-данных на входе, используются два режима: поллинга и прерывания. В режиме поллинга при поступлении данных на вход, бит 7 в порту статуса (2xEh) устанавливается в 1. Если данных нет, этот бит равен 0. Во втором режиме при поступлении данных происходит прерывание. Для перехвата прерывания пишется специальная функция. В любом из этих режимов чтение данных происходит одинаково:

1. Опрашивается порт 2xEh.
2. Если бит 7 установлен в 1, считываем данные из порта 2xAh.
3. Если бит 7 равен 0, переходим к первому пункту.

В интерфейсе MPU-401 (режим UART) все данные передаются без изменений между компьютером и MIDI-устройством. Когда DSP установлен в режим UART, он не реагирует ни на какие команды, кроме сброса (00h). Интерфейс MPU-401 в режиме UART использует аппаратное прерывание и выделенные адреса ввода-вывода. Как правило, могут применяться следующие номера прерываний: 2, 5 (по умолчанию), 7 или 10. Для адресации используются базовые порты 300h или 330h (по умолчанию). При этом порты ввода-вывода могут иметь следующие адреса: 300h и 301h или 330h и 331h.

Рассмотрим назначение портов подробнее:

- Порт 3x1h в режиме чтения является портом состояния. С помощью этого порта можно определить наличие данных в порту данных. Используются только биты 7 и 6. Бит 7 отвечает за готовность к вводу данных: 0 — данные доступны для считывания, 1 — нет данных для чтения. Бит 6 отвечает за готовность данных на выходе: 0 — интерфейс готов к приему данных или кода команды, 1 — интерфейс не готов принять данные или код команды.
- Порт 3x1h в режиме записи служит для посылки кода управляющей команды.
- Порт 3x0h порт данных. Используется для считывания и записи данных.

В листинге 6.9 показано, как следует проверять готовность интерфейса MIDI к приему данных.

Листинг 6.9. Проверка готовности интерфейса MIDI

```
// пишем функцию проверки порта
bool IsReadMIDI ( )
{
    DWORD dwResult  = 0; // переменная для хранения результата
```

```
    int iTimeWait = 50000;
    while ( -- iTimeWait > 0 )
     {
        // читаем состояние порта 301h
        inPort ( 0x301, &dwResult, 1 );
        if ( ( dwResult & 0x40 ) == 0x00 ) return true;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) break;
     }
    return false;
}
// используем нашу функцию для записи данных в порт 300h
if ( IsReadMIDI )
  outPort ( 0x301, dwCommand, 1 ); // записываем команду в порт
if ( IsReadMIDI )
  outPort ( 0x300, dwData, 1 ); // записываем данные в порт
```

Разберем еще пример, позволяющий определить наличие данных для считывания из порта MIDI (листинг 6.10).

Листинг 6.10. Определение наличия в порту MIDI-данных

```
// пишем функцию для проверки наличия данных в порту
bool IsDataMIDI ( )
{
    DWORD dwResult  = 0; // переменная для хранения результата
    int iTimeWait = 50000;
    while ( -- iTimeWait > 0 )
     {
        // читаем состояние порта 301h
        inPort ( 0x301, &dwResult, 1 );
        if ( ( dwResult & 0x80 ) == 0x00 ) return true;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) break;
     }
    return false;
}
// используем нашу функцию для чтения данных из порта 300h
if ( IsDataMIDI )
  inPort ( 0x300, &dwResult, 1 ); // читаем данные
```

В режиме UART поддерживаются только две команды: сброса и включения режима UART. Команда сброса с кодом `FFh` позволяет выполнить сброс ин-

терфейса MPU-401. Если команда успешно выполнена, интерфейс возвратит значение FEh через порт данных (3x0h). Команда сброса может использоваться для проверки наличия интерфейса MPU-401. Например, чтобы выполнить сброс и убедиться в наличии интерфейса, можно применить код, показанный в листинге 6.11.

Листинг 6.11. Проверка поддержки интерфейса MPU-401

```
bool IsMPU_401 ( )
{
    int iTimeWait = 65535;
    // используем нашу функцию для проверки готовности MIDI
    if ( IsReadMIDI )
      outPort ( 0x301, 0xFF, 1 ); // записываем команду сброса в порт
    while ( -- iTimeWait > 0 )
     {
        // проверяем наличие данных
        if ( IsDataMIDI )
           inPort ( 0x300, &dwResult, 1 ); // читаем данные
        if ( dwResult == 0xEF ) return true;
        // закончилось время ожидания
        if ( iTimeWait < 1 ) break;
     }
    return false; // MPU-401 отсутствует
}
```

Вторая команда (3Fh) позволяет включить режим UART для обмена данными. Если команда успешно выполнена, то интерфейс возвратит значение FEh через порт данных (3x0h). Принцип работы с ней такой же, как и с командой сброса.

Для передачи данных в режиме UART на внешнее устройство MIDI нужно сделать следующее:

1. Проверить готовность устройства MIDI.
2. Если бит 6 равен 0, то следует записать данные в порт 3x0h.
3. Если бит 6 равен 1, то необходимо перейти к первому пункту.

Для получения данных от внешнего устройства MIDI необходимо выполнить следующие действия:

1. Проверить наличие данных в порту статуса.
2. Если бит 7 равен 0, то следует прочитать данные из порта 3x0h.
3. Если бит 7 равен 1, то необходимо перейти к первому пункту.

На этом я хотел бы завершить описание работы с составляющими звуковой платы и привести наиболее часто встречающееся расположение базовых адресов (табл. 6.6) и регистров (табл. 6.7).

Таблица 6.6. *Базовые адреса звуковых плат*

Базовый адрес	Диапазон адресов
220h	220h–233h
240h	240h–253h
260h	260h–273h
280h	280h–293h
388h для FM	389h для FM

Таблица 6.7. *Адреса регистров звуковых плат*

Смещение адреса регистра от базового	Описание
0h	Регистр состояния FM (чтение)
0h	Адресный регистр FM (запись)
1h	Регистр данных FM (только запись)
2h	Расширенный регистр состояния FM (чтение)
2h	Расширенный адресный регистр FM (запись)
3h	Расширенный регистр данных FM (только запись)
4h	Адресный регистр микшера (только запись)
5h	Регистр данных микшера (чтение и запись)
6h	Регистр сброса DSP (только запись)
8h	Регистр состояния FM (чтение)
8h	Адресный регистр FM (запись)
9h	Регистр данных FM (только запись)
Ah	Регистр DSP для чтения данных (только чтение)
Ch	Регистр DSP для записи команд и данных (запись)
Ch	Регистр состояния буфера записи (бит 7) DSP (чтение)
Eh	Регистр состояния буфера чтения (бит 7) DSP (только чтение)
10h	Регистр команд или данных для CD-ROM (чтение и запись)
11h	Регистр состояния для CD-ROM (только чтение)

Таблица 6.7 *(окончание)*

Смещение адреса регистра от базового	Описание
12h	Регистр сброса для CD-ROM (только запись)
13h	Регистр включения CD-ROM (только запись)

6.2. Использование Win32 API

Windows предоставляет программисту возможность работы с микшером звуковой платы и с интерфейсом MIDI. Использование MIDI больше относится к программированию звука, поэтому мы его рассмотрим в следующей главе, а вот управление микшером изучим прямо здесь, поскольку это имеет непосредственное отношение к программированию звуковой платы.

В Windows достаточно удобно работать с микшером, поскольку имеется универсальный программный набор соответствующих функций. Нам не нужно разбираться в портах и особенностях той или иной звуковой платы, а достаточно использовать стандартные функции Win32. Рассмотрим эти функции подробнее:

- `mixerGetDevCaps` — позволяет получить информацию об установленном микшере;
- `mixerOpen` — позволяет открыть указанное устройство микшера;
- `mixerClose` — закрывает указанное устройство микшера;
- `mixerGetID` — позволяет получить идентификатор для указанного устройства микшера;
- `mixerGetNumDevs` — определяет количество микшеров, установленных в системе (аппаратных и виртуальных);
- `mixerGetLineInfo` — позволяет получить информацию об указанной линии (канале) микшера;
- `mixerGetLineControls` — позволяет получить различные элементы управления (например, ползунки для отображения уровней громкости), расположенные на указанной линии (аудиоканале) микшера;
- `mixerGetControlDetails` — позволяет получить значения параметров для элементов управления, расположенных на указанной линии (аудиоканале) микшера;
- `mixerSetControlDetails` — позволяет установить новые значения параметров для элементов управления, расположенных на указанной линии (аудиоканале) микшера;

Для того чтобы разобраться, как все это работает, я приведу пример класса для управления микшером. Назовем наш класс тривиально `CMixer`. В листинге 6.12 показан файл CMixer.h. Перед началом работы следует добавить в опции компоновщика ссылку на стандартную библиотеку Winmm.lib. Кроме того, в начало файла поместите ссылку на файл MMsystem.h.

Листинг 6.12. Файл CMixer.h класса `CMixer`

```
// подключаем файл поддержки функций микшера
#include <mmsystem.h>
// максимальное количество микшеров
#define MAX_MIXER 10
// объявляем наш класс
class CMixer
{
public:
    CMixer ( );
    virtual ~CMixer ( );
    // поиск первого микшера
    void InitMixer ( );
    // функция для установки дескриптора родительского окна
    void SetHWND ( HWND hDlg );
    // функция для получения идентификатора элемента управления
    DWORD GetCtrlID ( int iNum );
    // получение идентификатора открытого микшера
    HMIXER GetMixerID ( );
// основные функции работы с микшером
    // управление общим каналом звука
    DWORD GetMinMasterVol ( ); // получить минимальный уровень громкости
    DWORD GetMaxMasterVol ( ); // получить максимальный уровень громкости
    DWORD GetTicMasterVol ( ); // получить значение шага изменения уровня
    DWORD GetMasterValue ( ); // получить текущее значение громкости
    void SetMasterValue (DWORD dwValue ); // установить новое значение
    // управление общим отключением звука
    bool GetMasterMute ( ); // проверить состояние
    void SetMasterMute ( DWORD dwValue ); // установить новое состояние
    // управление фильтром нижних частот (если он есть на звуковой плате)
    DWORD GetMinBass ( ); // получить минимальное значение
    DWORD GetMaxBass ( ); // получить максимальное значение
    DWORD GetStepBass ( ); // получить значение шага изменения уровня
    DWORD GetBassValue ( ); // получить текущее значение
    void SetBassValue ( DWORD dwValue ); // установить новое значение
```

```
    // управление фильтром верхних частот (если он есть на плате)
    DWORD GetMinTreble ( ); // получить минимальное значение
    DWORD GetMaxTreble ( ); // получить максимальное значение
    DWORD GetStepTreble ( ); // получить значение шага изменения уровня
    DWORD GetTrebleValue ( ); // получить текущее значение
    void SetTrebleValue ( DWORD dwValue ); // установить новое значение
private:
HWND m_hDlg; // переменная для хранения дескриптора окна
HMIXER Mixer; // дескриптор микшера
// структуры данных для поддержки микшера
MIXERLINE mxl;
MIXERCONTROL mxc;
MIXERLINECONTROLS mxlc;
// пишем свою структуру для хранения информации о микшере
typedef struct _MixerInfo
{
char Name [MAXPNAMELEN]; // строковое название микшера
unsigned int uID; // идентификатор устройства микшера
WORD manID; // идентификатор производителя микшера
WORD prodID; // идентификатор изделия
DWORD drvVer; // номер версии драйвера
DWORD dwFlags; // дополнительные возможности микшера
DWORD dwLines; // количество доступных линий микшера
} MIXERINFO, *PMIXERINFO;
// структура описания значений параметров для компонентов микшера
typedef struct _Params
{
char name[50]; // имя компонента
DWORD dwID; // идентификатор компонента
DWORD min; // минимальное значение
DWORD max; // максимальное значение
DWORD cur; // текущее значение
DWORD Step; // шаг
} COMPONENTPARAM, *PCOMPONENTPARAM;
// дополнительная структура для описания используемых компонентов
typedef struct _Components
{
    // 0- Master Volume, 1- Mute Master, 2- Bass, 3- Treble
    COMPONENTPARAM comP[4];
} COMP;
// объявляем необходимые структуры
COMP com;
MIXERINFO info;
```

```
// и функции
HMIXER OpenMixer ( HWND hwnd, UINT mixerID ); // открыть микшер
void CloseMixer ( ); // закрыть микшер
}; // конец объявления класса CMixer
```

А теперь посмотрите на файл реализации (CMixer.cpp) класса `CMixer`, представленный в листинге 6.13.

Листинг 6.13. Файла CMixer.cpp класса `CMixer`

```
#include "stdafx.h"
#include "CMixer.h"
// конструктор класса
CMixer :: CMixer ( )
{
  Mixer = NULL;
  m_hDlg = NULL;
  // готовим структуры к использованию
  memset ( &com, 0, sizeof ( COMP ) );
  memset ( &mxc, 0, sizeof ( MIXERCONTROL ) );
  memset ( &mxl, 0, sizeof ( MIXERLINE ) );
  memset ( &mxlc, 0, sizeof ( MIXERLINECONTROLS ) );
  memset ( &info, 0, sizeof ( MIXERINFO ) );
  // если микшер в системе не обнаружен, то выходим
  if ( !mixerGetNumDevs ( ) ) return;
}
// деструктор класса
CMixer :: ~CMixer ( )
{
   // закрываем устройство микшера
   CloseMixer ( );
}
// функция для установки дескриптора родительского окна
void SetHWND ( HWND hDlg );
{
    m_hDlg = hDlg
}
// функция поиска и инициализации микшера
void CMixer :: InitMixer ( )
{
  if ( m_hDlg == NULL ) return;
  // ищем первый завалявшийся микшер
  for ( int i = 0; i < MAX_MIXER; i++ )
```

```
   {
     if ( !OpenMixer ( m_hDlg, i ) )
        continue;
     else
        break;
     if ( i > 9 ) return;
   }
// получаем информацию об общем регуляторе громкости
mxl.cbStruct = sizeof ( MIXERLINE );
mxl.dwComponentType = MIXERLINE_COMPONENTTYPE_DST_SPEAKERS;
mixerGetLineInfo ( ( HMIXEROBJ ) Mixer, &mxl, MIXER_OBJECTF_HMIXER |
                   MIXER_GETLINEINFOF_COMPONENTTYPE );
// получаем параметры для общего регулятора громкости
mxlc.cbStruct = sizeof ( MIXERLINECONTROLS );
mxlc.dwLineID = mxl.dwLineID;
mxlc.dwControlType = MIXERCONTROL_CONTROLTYPE_VOLUME;
mxlc.cControls = 1;
mxlc.cbmxctrl = sizeof ( MIXERCONTROL );
mxlc.pamxctrl = &mxc;
mixerGetLineControls ( ( HMIXEROBJ ) Mixer, &mxlc, MIXER_OBJECTF_HMIXER |
                       MIXER_GETLINECONTROLSF_ONEBYTYPE );
// сохраняем полученные значения в нашу структуру
strcpy ( com.comP[0].name, mxl.szName ); // имя регулятора громкости
com.comP[0].min = mxc.Bounds.dwMinimum; // минимальное значение
com.comP[0].max = mxc.Bounds.dwMaximum; // максимальное значение
com.comP[0].dwID = mxc.dwControlID; // уникальный идентификатор
com.comP[0].Step = mxc.Metrics.cSteps; // шаг
// получаем информацию о переключателе, управляющем общим звуком
memset ( &mxc, 0, sizeof ( MIXERCONTROL ) );
memset ( &mxl, 0, sizeof ( MIXERLINE ) );
memset ( &mxlc, 0, sizeof ( MIXERLINECONTROLS ) );
mxl.cbStruct = sizeof ( MIXERLINE );
mxl.dwComponentType = MIXERLINE_COMPONENTTYPE_DST_SPEAKERS;

mixerGetLineInfo ( ( HMIXEROBJ ) Mixer, &mxl, MIXER_OBJECTF_HMIXER |
                   MIXER_GETLINEINFOF_COMPONENTTYPE );
// получаем параметры для управления переключателем
mxlc.cbStruct = sizeof ( MIXERLINECONTROLS );
mxlc.dwLineID = mxl.dwLineID;
mxlc.dwControlType = MIXERCONTROL_CONTROLTYPE_MUTE;
mxlc.cControls = 1;
mxlc.cbmxctrl = sizeof ( MIXERCONTROL );
mxlc.pamxctrl = &mxc;
```

```
mixerGetLineControls ( ( HMIXEROBJ ) Mixer, &mxlc, MIXER_OBJECTF_HMIXER |
                       MIXER_GETLINECONTROLSF_ONEBYTYPE );
// сохраняем полученные значения в нашу структуру
strcpy ( com.comP[1].name, mxl.szName ); // имя переключателя
com.comP[1].dwID = mxc.dwControlID; // идентификатор переключателя
// получаем информацию о регуляторе нижних частот
memset ( &mxc, 0, sizeof ( MIXERCONTROL ) );
memset ( &mxl, 0, sizeof ( MIXERLINE ) );
memset ( &mxlc, 0, sizeof ( MIXERLINECONTROLS ) );
mxl.cbStruct = sizeof ( MIXERLINE );
mxl.dwComponentType = MIXERLINE_COMPONENTTYPE_DST_SPEAKERS;
mixerGetLineInfo ( ( HMIXEROBJ ) Mixer, &mxl, MIXER_OBJECTF_HMIXER |
                   MIXER_GETLINEINFOF_COMPONENTTYPE );
// получаем параметры для регулятора нижних частот
mxlc.cbStruct = sizeof ( MIXERLINECONTROLS );
mxlc.dwLineID = mxl.dwLineID;
mxlc.dwControlType = MIXERCONTROL_CONTROLTYPE_BASS;
mxlc.cControls = 1;
mxlc.cbmxctrl = sizeof ( MIXERCONTROL );
mxlc.pamxctrl = &mxc;
mixerGetLineControls ( ( HMIXEROBJ ) Mixer, &mxlc, MIXER_OBJECTF_HMIXER |
                       MIXER_GETLINECONTROLSF_ONEBYTYPE );
// сохраняем полученные значения в нашу структуру
strcpy (com.comP[2].name, mxl.szName ); // имя регулятора нижних частот
com.comP[2].min = mxc.Bounds.dwMinimum; // минимальное значение
com.comP[2].max = mxc.Bounds.dwMaximum; // максимальное значение
com.comP[2].dwID = mxc.dwControlID; // уникальный идентификатор
com.comP[2].Step = mxc.Metrics.cSteps; // шаг
// получаем информацию о регуляторе верхних частот
memset ( &mxc, 0, sizeof ( MIXERCONTROL ) );
memset ( &mxl, 0, sizeof ( MIXERLINE ) );
memset ( &mxlc, 0, sizeof ( MIXERLINECONTROLS ) );
mxl.cbStruct = sizeof ( MIXERLINE );
mxl.dwComponentType = MIXERLINE_COMPONENTTYPE_DST_SPEAKERS;
mixerGetLineInfo ( ( HMIXEROBJ ) Mixer, &mxl, MIXER_OBJECTF_HMIXER |
                   MIXER_GETLINEINFOF_COMPONENTTYPE );
// получаем параметры для регулятора верхних частот
mxlc.cbStruct = sizeof ( MIXERLINECONTROLS );
mxlc.dwLineID = mxl.dwLineID;
mxlc.dwControlType = MIXERCONTROL_CONTROLTYPE_TREBLE;
mxlc.cControls = 1;
mxlc.cbmxctrl = sizeof ( MIXERCONTROL );
mxlc.pamxctrl = &mxc;
```

```
mixerGetLineControls ( ( HMIXEROBJ) Mixer, &mxlc, MIXER_OBJECTF_HMIXER |
                        MIXER_GETLINECONTROLSF_ONEBYTYPE );
// сохраняем полученные значения в нашу структуру
strcpy (com.comP[3].name, mxl.szName ); // имя регулятора верхних частот
com.comP[3].min = mxc.Bounds.dwMinimum; // минимальное значение
com.comP[3].max = mxc.Bounds.dwMaximum; // максимальное значение
com.comP[3].dwID = mxc.dwControlID; // уникальный идентификатор
com.comP[3].Step = mxc.Metrics.cSteps; // шаг
} // конец функции инициализации
// функция для закрытия микшера
void CMixer :: CloseMixer ( )
{
    if ( Mixer )
     {
       // если микшер существует, то закрываем его
       mixerClose ( Mixer );
       Mixer = NULL;
      }
}
// функция для получения идентификатора текущего микшера
HMIXER CMixer :: GetMixerID ( )
{
    if ( Mixer == NULL )
      return 0;
    return Mixer;
}
// функция для получения идентификатора для указанного компонента
DWORD CMixer :: GetCtrlID ( int iNum )
{
    if ( Mixer == NULL )
      return 0;
    switch ( iNum )
    {
     case 0: // Master Volume
         return com.comP[0].dwID;
     case 1: // Master Mute
         return com.comP[1].dwID;
     case 2: // Bass
         return com.comP[2].dwID;
     case 3: // Treble
         return com.comP[3].dwID;
    }
    return 0;
}
```

```
// функция для открытия устройства микшера
HMIXER CMixer :: OpenMixer ( HWND hwnd, UINT mixerID )
{
    UINT uID = 0;
    HMIXER mixerTMP;
    MIXERCAPS mixerCaps;
    // получаем информацию для указанного микшера
    if ( mixerGetDevCaps ( mixerID, &mixerCaps, sizeof ( MIXERCAPS ) ) )
     {
       return ( Mixer ); // возвращаем идентификатор предыдущего микшера
     }
    // открываем новое устройство микшера
    if ( mixerOpen ( &mixerTMP, mixerID, (DWORD) hwnd, 0L,
                     CALLBACK_WINDOW ) )
     {
       // если не удалось открыть новый микшер, возвращаем предыдущий
       return ( Mixer );
     }
    // если предыдущий микшер был открыт, закрываем его
    if ( Mixer )
     {
       if ( mixerClose ( Mixer ) )
        {
          // если не удалось закрыть текущий микшер, закрываем новый
          mixerClose ( mixerTMP );
          return ( Mixer );
        }
     }
    // получаем информацию о микшере
    mixerGetID ( ( HMIXEROBJ ) Mixer, &uID, MIXER_OBJECTF_HMIXER );
    strcpy ( info.Name, mixerCaps.szPname );
    info.uID = uID;
    info.manID = mixerCaps.wMid;
    info.prodID = mixerCaps.wPid;
    info.drvVer = mixerCaps.vDriverVersion;
    info.dwFlags = mixerCaps.fdwSupport;
    info.dwLines = mixerCaps.cDestinations;
    // сохраняем идентификатор открытого микшера
    Mixer = mixerTMP;
    // выходим из функции
    return ( mixerTMP );
}
```

```
// получаем текущее значение для общей громкости
DWORD CMixer :: GetMasterValue ( )
{
    MIXERCONTROLDETAILS_UNSIGNED mxcdVolume;
    MIXERCONTROLDETAILS mxcd;
    // получаем текущее значение
    mxcd.cbStruct = sizeof ( MIXERCONTROLDETAILS );
    mxcd.dwControlID = com.comP[0].dwID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof ( MIXERCONTROLDETAILS_UNSIGNED );
    mxcd.paDetails = &mxcdVolume;
    mixerGetControlDetails ( ( HMIXEROBJ ) Mixer, &mxcd,
                MIXER_OBJECTF_HMIXER | MIXER_GETCONTROLDETAILSF_VALUE );
    com.comP[0].cur = mxcdVolume.dwValue;
    // возвращаем текущее значение
    return mxcdVolume.dwValue;
}
// функция для установки текущего значения общей громкости
void CMixer :: SetMasterValue ( DWORD dwValue )
{
    MIXERCONTROLDETAILS_UNSIGNED mxcdVolume = { dwValue };
    MIXERCONTROLDETAILS mxcd;
    // устанавливаем новое значение
    mxcd.cbStruct = sizeof ( MIXERCONTROLDETAILS );
    mxcd.dwControlID = com.comP[0].dwID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof ( MIXERCONTROLDETAILS_UNSIGNED );
    mxcd.paDetails = &mxcdVolume;
    mixerSetControlDetails ( ( HMIXEROBJ ) Mixer, &mxcd,
              MIXER_OBJECTF_HMIXER | MIXER_SETCONTROLDETAILSF_VALUE );
}
// получение минимального значения регулятора общей громкости
DWORD CMixer :: GetMinMasterVol ( )
{
    if ( Mixer == NULL )
     return -1;
    return com.comP[0].min;
}
// получение максимального значения регулятора общей громкости
DWORD CMixer :: GetMaxMasterVol ( )
{
    if ( Mixer == NULL )
```

```
     return -1;
    return com.comP[0].max;
}
// получить шаг изменения общей громкости
DWORD CMixer :: GetStepMasterVol ( )
{
    if ( Mixer == NULL )
     return -1;
    return com.comP[0].Step;
}
// получение текущего состояния переключателя звука
bool CMixer :: GetMasterMute ( )
{
    MIXERCONTROLDETAILS_BOOLEAN mxcdMute;
    MIXERCONTROLDETAILS mxcd;
    // получаем текущее значение
    mxcd.cbStruct = sizeof ( MIXERCONTROLDETAILS );
    mxcd.dwControlID = com.comP[1].dwID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof ( MIXERCONTROLDETAILS_BOOLEAN );
    mxcd.paDetails = &mxcdMute;
    mixerGetControlDetails ( ( HMIXEROBJ ) Mixer, &mxcd,
                MIXER_OBJECTF_HMIXER | MIXER_GETCONTROLDETAILSF_VALUE );
    com.comP[1].cur = mxcdMute.fValue;
    // возвращаем текущее значение
    return ( bool ) mxcdMute.fValue;
}
// установка состояния переключателя звука
void CMixer :: SetMasterMute ( DWORD dwValue )
{
    MIXERCONTROLDETAILS_BOOLEAN mxcdMute = { dwValue };
    MIXERCONTROLDETAILS mxcd;
    // устанавливаем новое значение
    mxcd.cbStruct = sizeof ( MIXERCONTROLDETAILS );
    mxcd.dwControlID = com.comP[1].dwID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof ( MIXERCONTROLDETAILS_BOOLEAN );
    mxcd.paDetails = &mxcdMute;
    mixerSetControlDetails (  ( HMIXEROBJ ) Mixer, &mxcd,
             MIXER_OBJECTF_HMIXER | MIXER_SETCONTROLDETAILSF_VALUE );
}
```

```
// получение текущего значения для регулятора нижних частот
DWORD CMixer :: GetBassValue ( )
{
    MIXERCONTROLDETAILS_UNSIGNED mxcdValue;
    MIXERCONTROLDETAILS mxcd;
    // получаем текущее значение
    mxcd.cbStruct = sizeof ( MIXERCONTROLDETAILS );
    mxcd.dwControlID = com.comP[2].dwID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof ( MIXERCONTROLDETAILS_UNSIGNED );
    mxcd.paDetails = &mxcdValue;
    mixerGetControlDetails ( ( HMIXEROBJ ) Mixer, &mxcd,
             MIXER_OBJECTF_HMIXER | MIXER_GETCONTROLDETAILSF_VALUE );
    com.comP[2].cur = mxcdValue.dwValue;
    // возвращаем текущее значение
    return mxcdValue.dwValue;
}
// установка нового значения для регулятора нижних частот
void CMixer :: SetBassValue ( DWORD dwValue )
{
    MIXERCONTROLDETAILS_UNSIGNED mxcdVol = { dwValue };
    MIXERCONTROLDETAILS mxcd;
    // устанавливаем новое значение
    mxcd.cbStruct = sizeof ( MIXERCONTROLDETAILS );
    mxcd.dwControlID = com.comP[2].dwID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof ( MIXERCONTROLDETAILS_UNSIGNED );
    mxcd.paDetails = &mxcdVol;
    mixerSetControlDetails ( ( HMIXEROBJ ) Mixer, &mxcd,
           MIXER_OBJECTF_HMIXER | MIXER_SETCONTROLDETAILSF_VALUE );
}
// получение минимального значения регулятора нижних частот
DWORD CMixer :: GetMinBass ( )
{
    if ( Mixer == NULL )
     return -1;
    return com.comP[2].min;
}
// получение максимального значения регулятора нижних частот
DWORD CMixer :: GetMaxBass ( )
{
    if ( Mixer == NULL )
```

```
     return -1;
    return com.comP[2].max;
}
// получить шаг изменения для регулятора нижних частот
DWORD CMixer :: GetStepBass ( )
{
    if ( Mixer == NULL )
     return -1;
    return com.comP[2].Step;
}
// получение текущего значения для регулятора верхних частот
DWORD CMixer :: GetTrebleValue ( )
{
    MIXERCONTROLDETAILS_UNSIGNED mxcdValue;
    MIXERCONTROLDETAILS mxcd;
    // получаем текущее значение
    mxcd.cbStruct = sizeof ( MIXERCONTROLDETAILS );
    mxcd.dwControlID = com.comP[3].dwID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof ( MIXERCONTROLDETAILS_UNSIGNED );
    mxcd.paDetails = &mxcdValue;
    mixerGetControlDetails ( ( HMIXEROBJ ) Mixer, &mxcd,
                MIXER_OBJECTF_HMIXER | MIXER_GETCONTROLDETAILSF_VALUE);
    com.comP[3].cur = mxcdValue.dwValue;
    // возвращаем текущее значение
    return mxcdValue.dwValue;
}
// установка нового значения для регулятора верхних частот
void CMixer :: SetTrebleValue ( DWORD dwValue )
{
    MIXERCONTROLDETAILS_UNSIGNED mxcdVol = { dwValue };
    MIXERCONTROLDETAILS mxcd;
    // устанавливаем новое значение
    mxcd.cbStruct = sizeof ( MIXERCONTROLDETAILS );
    mxcd.dwControlID = com.comP[3].dwID;
    mxcd.cChannels = 1;
    mxcd.cMultipleItems = 0;
    mxcd.cbDetails = sizeof ( MIXERCONTROLDETAILS_UNSIGNED );
    mxcd.paDetails = &mxcdVol;
    mixerSetControlDetails ( ( HMIXEROBJ ) Mixer, &mxcd,
          MIXER_OBJECTF_HMIXER | MIXER_SETCONTROLDETAILSF_VALUE );
}
```

```
// получение минимального значения регулятора верхних частот
DWORD CMixer :: GetMinMasterVol ( )
{
    if ( Mixer == NULL )
     return -1;
    return com.comP[0].min;
}
// получение максимального значения регулятора верхних частот
DWORD CMixer :: GetMaxMasterVol ( )
{
    if ( Mixer == NULL )
     return -1;
    return com.comP[0].max;
}
// получить шаг изменения для регулятора верхних частот
DWORD CMixer :: GetStepMasterVol ( )
{
    if ( Mixer == NULL )
     return -1;
    return com.comP[0].Step;
}
```

Теперь, когда наш класс управления микшером готов, можно добавить его в свою программу. Как вы, наверное, заметили, мы обработали только четыре управляющих компонента микшера: регулятор общей громкости, переключатель для отключения и включения звука, а также регуляторы низких и высоких частот. Регулировка частот поддерживается не всеми звуковыми платами. Данный класс прекрасно будет работать с семейством звуковых плат от Creative "SB Live!". На платах других производителей должна быть поддержка управления низкими и высокими частотами.

Кроме того, разобравшись с работой класса, читатель сможет самостоятельно добавить обработку других стандартных компонентов микшера: регулировку громкости MIDI, CD аудио, линейного канала. Дополнительную информацию можно получить из документации фирмы Microsoft (на сайте **www.msdn.com**) по программированию звука в Windows, поскольку описать в небольшой главе все возможности просто нереально. Главное, чтобы вы поняли принципы работы с микшером и тогда сможете самостоятельно разобраться с остальными возможностями.

В завершение темы я приведу примерный код, использующий наш класс `CMixer` (листинг 6.14). Создайте в редакторе ресурсов диалоговое окно и добавьте на него три ползунка и один переключатель. Присвойте им следующие идентификаторы: `IDC_VOLUME` (регулятор общей громкости), `IDC_MUTE` (пере-

ключатель звука), IDC_BASS (регулятор низких частот), IDC_TREBLE (регулятор высоких частот).

Листинг 6.14. Использование класса CMixer

```
// Объявляем наш класс
CMixer mixer;
// функция обработки сообщений диалогового окна
BOOL CALLBACK MixerProc ( HWND hwndDlg, UINT message, WPARAM wParam,
                          LPARAM lParam );
// реализация диалоговой функции
BOOL CALLBACK MixerProc (HWND hwndDlg, UINT message, WPARAM wParam,
                         LPARAM lParam )
{
// получаем идентификаторы элементов управления
static HWND hVolume = NULL, hBass = NULL, hTreble = NULL, hMute = NULL;
hVolume = GetDlgItem ( hwndDlg, IDC_VOLUME );
hMute = GetDlgItem ( hwndDlg, IDC_MUTE );
hBass = GetDlgItem ( hwndDlg, IDC_BASS );
hTreble = GetDlgItem ( hwndDlg, IDC_TREBLE );
// идентификаторы компонентов микшера
static DWORD volID = 0, muteID = 0, bassID = 0, trebleID = 0;
 // обрабатываем сообщения
 switch ( message )
 {
  case WM_INITDIALOG:
   {
    DWORD min = 0, max = 0, cur = 0, Step = 0;
    // устанавливаем идентификатор окна
    mixer.SetHWND ( hwndDlg );
    // инициализируем микшер
    mixer.InitMixer ( );
    // получаем идентификаторы компонентов управления микшера
    volID = mixer.GetCtrlID ( 0 );
    muteID = mixer.GetCtrlID ( 1 );
    bassID = mixer.GetCtrlID ( 2 );
    trebleID = mixer.GetCtrlID ( 3 );
    // получаем значения параметров микшера
    min = mixer.GetMinMasterVol ( );
    max = mixer.GetMaxMasterVol ( );
    cur = mixer.GetMasterValue ( );
    Step = mixer.GetStepMasterVol ( );
    if ( Step == 0 ) Step = 5000;
```

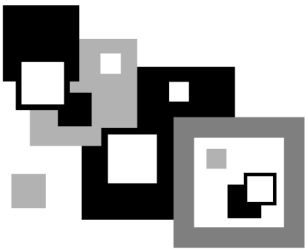
```
if ( max == 0 )
 {
   EnableWindow ( hVolume, false );
 }
else
 {
   SendMessage ( hVolume, TBM_SETRANGEMIN, ( WPARAM ) 0,
               ( LPARAM ) min );
   SendMessage ( hVolume, TBM_SETRANGEMAX, ( WPARAM ) 0,
                   ( LPARAM ) max );
   SendMessage ( hVolume, TBM_SETTICFREQ, ( WPARAM ) max / Step,
               ( LPARAM ) 0 );
   SendMessage ( hVolume, TBM_SETPOS, ( WPARAM ) 1,
               ( LPARAM ) ( LONG ) cur );
 }
min = 0; max = 0; cur = 0;
min = mixer.GetMinBass ( );
max = mixer.GetMaxBass ( );
cur = mixer.GetBassValue ( );
Step = mixer.GetStepBass ( );
if ( Step == 0 ) Step = 5000;
if ( max == 0 )
 {
   EnableWindow ( hBass, false );
 }
else
 {
   SendMessage ( hBass, TBM_SETRANGEMIN, ( WPARAM ) 0,
               ( LPARAM ) min);
   SendMessage ( hBass, TBM_SETRANGEMAX, ( WPARAM ) 0,
               ( LPARAM ) max);
   SendMessage ( hBass, TBM_SETTICFREQ, ( WPARAM ) max / Step,
               ( LPARAM ) 0 );
   SendMessage ( hBass, TBM_SETPOS, ( WPARAM ) 1,
               ( LPARAM ) ( LONG ) cur );
 }
min = 0; max = 0; cur = 0;
min = mixer.GetMinTreble ( );
max = mixer.GetMaxTreble ( );
cur = mixer.GetTrebleValue ( );
Step = mixer.GetStepTreble ( );
if ( Step == 0 ) Step = 5000;
if ( max == 0 )
```

```
    {
      EnableWindow ( hTreble, false );
    }
   else
    {
      SendMessage ( hTreble, TBM_SETRANGEMIN, ( WPARAM ) 0,
                  ( LPARAM ) min );
      SendMessage ( hTreble, TBM_SETRANGEMAX, (WPARAM)0,
                  ( LPARAM ) max );
      SendMessage ( hTreble, TBM_SETTICFREQ, ( WPARAM ) max / Step,
                  ( LPARAM ) 0 );
      SendMessage ( hTreble, TBM_SETPOS, (WPARAM)1,
                  ( LPARAM ) ( LONG ) cur );
    }
   if ( muteID == 0 )
    {
      EnableWindow ( hMute, false );
    }
   else
    {
      cur = mixer.GetMasterMute ( );
      if ( cur )
       {
         CheckDlgButton ( hwndDlg, IDC_MUTE, true );
         EnableWindow ( hVolume, false );
         EnableWindow ( hBass, false );
         EnableWindow ( hTreble, false );
       }
    }
  }
  return true;
 case WM_COMMAND:
  switch ( LOWORD ( wParam ) )
   {
    case IDCANCEL:
    case IDOK:
     EndDialog ( hwndDlg, wParam );
     return TRUE;
    // обрабатываем выключатель звука
    case IDC_MUTE:
     if ( HIWORD ( wParam ) == BN_CLICKED )
      {
       if ( IsDlgButtonChecked ( hwndDlg, IDC_VOL_MUTE )
                                == BST_CHECKED )
```

```
        {
          mixer.SetMasterMute ( 1 );
          EnableWindow ( hVolume, false );
          EnableWindow ( hBass, false );
          EnableWindow ( hTreble, false );
        }
       else
        {
          mixer.SetMasterMute ( 0 );
          EnableWindow ( hVolume, true );
          EnableWindow ( hBass, true );
          EnableWindow ( hTreble, true );
        }
       }
      break;
    }
  case WM_HSCROLL:
   {
    // обрабатываем регуляторы
    DWORD pos = 0;
    if ( ( HWND ) lParam == hVolume )
     {
       // получаем текущее положение ползунка
       pos = SendMessage ( hVolume, TBM_GETPOS, 0, 0 );
       // устанавливаем новое значение
       mixer.SetMasterValue ( pos );
     }
    else if ( ( HWND ) lParam == hBass )
     {
       pos = SendMessage ( hBass, TBM_GETPOS, 0, 0 );
       mixer..SetBassValue ( pos );
     }
    else if ( ( HWND ) lParam == hTreble )
     {
       pos = SendMessage ( hTreble, TBM_GETPOS, 0, 0 );
       mixer.SetTrebleValue ( pos );
     }
   }
   break;
 }
 return false;
}
```

Как видите, нет ничего сложного в использовании класса `CMixer`. На этой оптимистической ноте я хотел бы завершить описание программирования звуковой платы и перейти к следующей главе.

ГЛАВА 7



Работа со звуком

В этой главе я хочу познакомить читателя с программированием звука в операционных системах Windows. К сожалению, информации такого рода встречается очень и очень мало. Все так увлеклись базами данных и их составляющими, что совсем забыли о том, что современный компьютер позволяет полностью заменить высококачественный музыкальный центр или звуковую студию, принося гораздо больше пользы, а главное удовольствия обычным пользователям. Трудно встретить человека, не слушающего музыку. Многие не только слушают, но и сами создают музыкальные произведения. А сколько пользы приносят обучающие компьютерные программы для маленьких детей, развивая не только слух и чувство ритма, но и в игровой форме знакомя с музыкальной азбукой. Что ни говори, а без звука операционные системы Windows потеряли бы половину своей привлекательности и удобства. Без звука трудно себе представить современную игровую или производственную (отслеживающую какие-либо задачи в реальном времени) программу, поэтому я и решил в доступной форме представить читателю различные способы программирования звука в Windows. Мы рассмотрим несколько базовых тем:

1. Создание полноценного плеера аудиодисков средствами MCI (Media Control Interface).
2. Программирование MIDI в Windows.
3. Доступ к файлам в формате MP3.

7.1. Создание плеера аудиодисков

Для того чтобы создать полноценный плеер аудиодисков, нам понадобится всего одна функция MCI. Скажете невозможно? Я бы с вами согласился, если бы не был убежден в обратном. Называется эта функция `mciSendCommand` и

предназначена для передачи различных предопределенных сообщений устройству. В качестве устройства могут быть использованы следующие варианты: CD аудио, устройство цифрового воспроизведения звука или видео, MIDI, сканер, видеокассетный магнитофон, проигрыватель видеодисков и стандартное устройство воспроизведения звука. Функция `mciSendCommand` имеет четыре аргумента:

- `IDDevice` — идентификатор открываемого устройства. Данный параметр не должен использоваться с сообщением `MCI_OPEN`.
- `uMsg` — специальное командное сообщение. Наиболее важные для нас сообщения помещены в табл. 7.1.
- `fdwCommand` — дополнительные флаги команды. Возможные значения перечислены в табл. 7.2.
- `dwParam` — указатель на структуру, определяющую параметры команды.

Таблица 7.1. *Командные сообщения*

Имя сообщения	Описание
`MCI_OPEN`	Позволяет инициализировать любое устройство (файл) MCI
`MCI_CLOSE`	Закрывает любое устройство (файл) MCI
`MCI_GETDEVCAPS`	Позволяет получить информацию о любом устройстве MCI
`MCI_INFO`	Позволяет получить информацию о любом устройстве MCI в виде строки
`MCI_PAUSE`	Приостанавливает текущее воспроизведение
`MCI_RESUME`	Возобновляет воспроизведение, приостановленное `MCI_PAUSE`
`MCI_PLAY`	Позволяет начать процесс воспроизведения
`MCI_SEEK`	Позволяет изменить текущую позицию воспроизведения
`MCI_SET`	Позволяет настроить доступные параметры устройства
`MCI_STATUS`	Позволяет получить текущее состояние устройства
`MCI_STOP`	Останавливает процесс воспроизведения

Таблица 7.2. *Дополнительные флаги*

Имя флага	Описание
`MCI_OPEN_TYPE_ID`	Определяет, что младшее слово параметра `lpstrDeviceType` содержит стандартный идентификатор устройства, а старшее — порядковый индекс

***Таблица 7.2** (окончание)*

Имя флага	Описание
MCI_OPEN_TYPE	Тип устройства будет включен в параметр lpstrDeviceType
MCI_NOTIFY	Позволяет определить обработку уведомляющих сообщений для родительского окна
MCI_OPEN_SHAREABLE	Устройство (файл) будет открыто как общий ресурс
MCI_WAIT	Заставляет устройство возвращать управление программе только после завершения операции
MCI_ALL_DEVICE_ID	Указывает на то, что команда будет передана всем доступным устройствам MCI
MCI_STATUS_MODE	Позволяет установить параметр dwReturn (тип возвращаемого режима работы)
MCI_STATUS_ITEM	Позволяет установить тип возвращаемого параметра состояния

Кроме рассмотренной функции нам понадобятся несколько структур MCI: MCI_OPEN_PARMS, MCI_GENERIC_PARMS, MCI_STATUS_PARMS, MCI_SET_PARMS, MCI_PLAY_PARMS, MCI_SEEK_PARMS и MCI_GETDEVCAPS_PARMS. Каждая из них используется для обработки определенного командного сообщения.

Прежде чем писать код самого плеера, создадим три вспомогательных класса: MCI, MSF и TMSF. Первый класс будет отвечать за инициализацию устройства MCI, а два других помогут осуществить удобное форматирование параметров времени. Для удобства поместим все три класса в общий файл. В листинге 7.1 показан заголовочный файл MCI.h, а в листинге 7.2 — файл реализации MCI.cpp. Не забудьте добавить в опции компоновщика ссылку на библиотеку Winmm.lib.

Листинг 7.1. Файл MCI.h

```
#include <mmsystem.h>
// объявляем класс MSF
class MSF
{
public:
    // конструктор по умолчанию
    MSF ( ) { m_MSF = 0; }
   // дополнительный конструктор с инициализацией
   MSF ( DWORD dwMSF ) { m_MSF = dwMSF; }
   // пустой деструктор
   ~MSF ( ) { }
```

```
   // общедоступные функции для форматирования времени
   BYTE GetMin ( ); // получает составляющую минут
   BYTE GetSec ( ); // получает составляющую секунд
   BYTE GetFrm ( ); // получает составляющую фреймов
private:
   // переменная для хранения времени
   DWORD m_MSF;
}; // конец класса MSF
// объявляем класс TMSF
class TMSF
{
public:
// первый конструктор
   TMSF ( ) { m_TMSF = 0; }
   // второй конструктор
   TMSF ( BYTE Track, BYTE Min, BYTE Sec, BYTE Frm )
    {
      // пакуем номер и время трека в DWORD
      m_TMSF = MCI_MAKE_TMSF ( Track, Min, Sec, Frm );
    }
   // пустой деструктор
   ~TMSF ( ) { }
   // общедоступные функции для форматирования времени и номера трека
   BYTE GetTrack ( ); // возвращает номер трека
   BYTE GetMin ( ); // получает составляющую минут
   BYTE GetSec ( ); // получает составляющую секунд
   BYTE GetFrm ( ); // получает составляющую фреймов
   // определяем оператор для текущего значения
   operator DWORD ( ) { return m_TMSF; }
private:
   // переменная для хранения текущего значения
   DWORD m_TMSF;
}; // конец класса TMSF
// объявляем класс MCI
class MCI
{
public:
   // конструктор по умолчанию и деструктор класса
   MCI ( );
   ~MCI ( );
   // общедоступные функции
   DWORD OpenMCI ( DWORD dwDevType ); // открывает устройство MCI
   DWORD Close ( ); // закрывает устройство MCI
```

```
   DWORD GetMode ( ); // получает текущий режим работы устройства MCI
   DWORD GetStatus ( DWORD dwFlag ); // получает текущее состояние
   void SetWindow ( HWND hWnd ); // устанавливает родительское окно
   MCIERROR GetError ( ); // получить последнюю ошибку MCI
protected:
   // переменная для хранения идентификатора устройства MCI
   MCIDEVICEID m_MCIID;
   HWND m_hWnd; // дескриптор родительского окна
   DWORD ExecCommand ( unsigned int uCommand, DWORD dwFlag,
                       DWORD dwParam ); // выполняет команду MCI
private:
   // переменная для хранения последней ошибки MCI
   MCIERROR m_dwError;
   void FreeMCI ( ); // закрывает все устройства MCI, освобождая ресурсы
}; // конец класса MCI
```

Листинг 7.2. Файл MCI.cpp

```
#include "stdafx.h"
#include "mci.h"
// реализация класса MSF
BYTE MSF :: GetMin ( )
{
   return MCI_MSF_MINUTE ( m_MSF );
}
BYTE MSF :: GetSec ( )
{
   return MCI_MSF_SECOND ( m_MSF );
}
BYTE MSF :: GetFrm ( )
{
   return MCI_MSF_FRAME ( m_MSF );
}
// окончание реализация класса MSF
// реализация класса TMSF
BYTE TMSF :: GetTrack ( )
{
   return MCI_TMSF_TRACK ( m_TMSF );
}
BYTE TMSF :: GetMin ( )
{
   return MCI_TMSF_MINUTE ( m_TMSF );
}
```

```
BYTE TMSF :: GetSec ( )
{
   return MCI_TMSF_SECOND ( m_TMSF );
}
BYTE TMSF :: GetFrm ( )
{
   return MCI_TMSF_FRAME ( m_TMSF );
}
// окончание реализация класса TMSF
// реализация класса MCI
MCI :: MCI ( ) // конструктор класса
{
   m_MCIID = NULL;
   m_hWnd = NULL;
}
// деструктор класса
MCI :: ~MCI ( )
{
   if ( m_MCIID )
    {
      FreeMCI ( );
      m_MCIID = NULL;
    }
}
// функция для открытия устройства MCI
DWORD MCI :: OpenMCI ( DWORD dwDevType )
{
   MCI_OPEN_PARMS mciOpenParms;
   DWORD dwResult;
   // определяет тип устройства
   mciOpenParms.lpstrDeviceType = (LPCSTR) dwDevType;
   DWORD dwFlags = MCI_OPEN_TYPE_ID | MCI_OPEN_TYPE | MCI_WAIT;
   // открываем устройство
   dwResult = ExecCommand ( MCI_OPEN, dwFlags,
                            ( DWORD ) ( LPVOID ) &mciOpenParms );
   if ( dwResult == 0 )
    {
       // сохраняем идентификатор устройства
       m_MCIID = mciOpenParms.wDeviceID;
    }
   return dwResult;
}
```

```
// функция для закрытия устройства MCI
DWORD MCI :: Close ( )
{
   MCI_GENERIC_PARMS mciGenericParms;
   mciGenericParms.dwCallback = (DWORD) m_hWnd;
   return ExecCommand ( MCI_CLOSE, 0, ( DWORD ) &mciGenericParms );
}
// возвращает текущий режим работы
DWORD MCI :: GetMode ( )
{
   return GetStatus ( MCI_STATUS_MODE );
}
// возвращает состояние для указанного режима
DWORD MCI :: GetStatus ( DWORD dwFlag )
{
   MCI_STATUS_PARMS mciStatusParms;
   mciStatusParms.dwCallback = ( DWORD ) m_hWnd;
   mciStatusParms.dwReturn = 0;
   mciStatusParms.dwItem = dwFlag;
   ExecCommand ( MCI_STATUS, MCI_STATUS_ITEM,
               ( DWORD ) &mciStatusParms );
   return mciStatusParms.dwReturn;
}
// устанавливает окно, которое будет получать уведомления MCI
void MCI :: SetWindow ( HWND hWnd )
{
   m_hWnd = hWnd;
}
// функция для выполнения указанной команды MCI
DWORD MCI :: ExecCommand ( unsigned int uCommand, DWORD dwFlag,
                           DWORD dwParam )
{
   DWORD dwResult;
   if ( dwResult = mciSendCommand ( m_MCIID, uCommand, dwFlag,
                                    dwParam ) )
    {
      m_dwError = dwResult;
    }
   return dwResult;
}
// функция возвращает последнюю ошибку MCI
MCIERROR MCI :: GetError ( ) const
```

```
{
   return m_dwError;
}
// функция закрывает все устройства MCI и освобождает ресурсы
void MCI :: FreeMCI ( )
{
   mciSendCommand ( MCI_ALL_DEVICE_ID, MCI_CLOSE, MCI_WAIT, NULL );
}
```

Классы `MSF` и `TMSF` помогут нам работать с представлением времени. Для аудиодиска данные можно представить в удобном формате: минута, секунда и фрейм. Дополнительно для операций воспроизведения понадобится еще указать номер трека. Наши классы `MSF` и `TMSF` полностью решают эту задачу. Класс `MCI` является базовым и управляет глобальной работой MCI. С помощью него мы открываем устройство MCI, назначаем дескриптор окна, для которого будут посылаться уведомляющие сообщения MCI, а также закрываем устройство и все используемые системные ресурсы. Этот класс предоставляет универсальную функцию `ExecCommand`, с помощью которой выполняется передача всех управляющих команд устройству.

Теперь у нас все готово для создания класса аудиоплеера. Назовем его `CDAudio` и сделаем производным от MCI. Добавим необходимые функции для управления плеером. Файл CDAudio.h показан в листинге 7.3.

Листинг 7.3. Файл CDAudio.h

```
#include "mci.h"
// объявляем наш класс
class CDAudio : public MCI
{
public:
   // определяем конструктор по умолчанию
   CDAudio ( );
   // определяем пустой деструктор
   ~ CDAudio ( );
   // определяем общедоступные функции класса
   DWORD Open ( ); // открываем устройство CD Audio
   // функции для управления выдвижного лотка CD-ROM
   void OpenTray ( ); // открыть
   void CloseTray ( ); // закрыть
   // функции для управления воспроизведением
   void PlayCD ( int TrackStart, int MinStart, int SecStart,
   int TrackEnd, int MinEnd, int SecEnd ); // играть от и до
```

```
   void PlayStartEndCD ( DWORD dwStart, DWORD dwEnd );
   void PauseCD ( ); // приостановить воспроизведение
   void ResumeCD ( ); // продолжить воспроизведение
   // перемещение по диску
   void SeekToStart ( ); // в начало диска
   void SeekToEnd ( ); // в конец диска
   // проверка состояния
   bool IsDisk ( ); // есть ли диск
   bool IsReady ( ); // готово ли устройство
   // управление форматом представления данных
   void GetFormat ( ); // получить текущий формат
   void SetFormat ( DWORD dwFormat ); // установить новый формат
   // получение различной информации
   DWORD GetTrackType ( DWORD dwTrack ); // тип трека
   DWORD GetTrackLen ( DWORD dwTrack ); // длина трека
   DWORD GetTrackPosition ( DWORD dwTrack ); // позиция трека
   DWORD GetCountTracks ( ); // количество треков на диске
   DWORD GetTotalLenCD ( ); // полный размер диска
   DWORD GetCurPosition ( ); // текущая позиция
   DWORD GetCurTrack ( ); // текущий трек
private:
   // сервисные функции
   DWORD _getTrackInfo ( DWORD dwTrack, DWORD dwFlag ); // информация
   DWORD _seek ( DWORD dwPos, DWORD dwFlag ); // перемещение по диску
}; // окончание класса CDAudio
```

Как вы видите, в классе определены наиболее важные функции управления нашим устройством. На их основе можно реализовать практически любые возможности для аудиоплеера. Файл реализации (CDAudio.cpp) класса `CDAudio` представлен в листинге 7.4.

Листинг 7.4. Файл CDAudio.cpp

```
#include "stdafx.h"
#include "CDAudio.h"
// конструктор класса
CDAudio :: CDAudio ( );
{
}
// сервисные функции
DWORD CDAudio :: _getTrackInfo ( DWORD dwTrack, DWORD dwFlag )
{
   MCI_STATUS_PARMS mciStatus;
   mciStatus.dwTrack = dwTrack;
```

```
   mciStatus.dwItem = dwFlag;
   mciStatus.dwReturn = 0;
   mciStatus.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_STATUS, MCI_TRACK | MCI_STATUS_ITEM,
               ( DWORD ) &mciStatus );
   return mciStatus.dwReturn;
}
DWORD CDAudio :: _seek ( DWORD dwPos, DWORD dwFlag )
{
   MCI_SEEK_PARMS mciSeek;
   mciSeek.dwTo = dwPos;
   dwFlag |= MCI_NOTIFY;
   mciSeek.dwCallback = ( DWORD ) m_hWnd;
   return ExecCommand ( MCI_SEEK, dwFlag, ( DWORD ) &mciSeek );
}
// общие функции управления и поддержки
DWORD CDAudio :: Open ( )
{
   // открываем устройство CD аудио
   return OpenMCI ( MCI_DEVTYPE_CD_AUDIO );
}
void CDAudio :: OpenTray ( )
{
   MCI_SET_PARMS mciSet;
   mciSet.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_SET, MCI_SET_DOOR_OPEN, ( DWORD ) &mciSet );
}
void CDAudio :: CloseTray ( )
{
   MCI_SET_PARMS mciSet;
   mciSet.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_SET, MCI_SET_DOOR_CLOSED, ( DWORD ) &mciSet );
}
void CDAudio :: PlayCD ( int TrackStart, int MinStart, int SecStart,
                         int TrackEnd, int MinEnd, int SecEnd )
{
   MCI_PLAY_PARMS mciPlay;
   mciPlay.dwFrom = MCI_MAKE_TMSF ( TrackStart, MinStart, SecStart, 0 );
   mciPlay.dwTo = MCI_MAKE_TMSF ( TrackEnd, MinEnd, SecEnd, 0 );
   DWORD dwFlag = MCI_NOTIFY | MCI_FROM | MCI_TO;
   mciPlay.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_PLAY , dwFlag, ( DWORD ) ( LPVOID ) &mciPlay );
}
```

```
void CDAudio :: PlayStartEndCD ( DWORD dwStart, DWORD dwEnd )
{
   MCI_PLAY_PARMS mciPlay;
   mciPlay.dwFrom = dwStart;
   mciPlay.dwTo = dwEnd;
   DWORD dwFlag = MCI_NOTIFY | MCI_FROM | MCI_TO;
   mciPlayParms.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_PLAY , dwFlag, ( DWORD ) ( LPVOID ) &mciPlay );
}
void CDAudio :: PauseCD ( )
{
   MCI_GENERIC_PARMS mciGeneric;
   mciGeneric.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_PAUSE, 0, ( DWORD ) &mciGeneric );
}
void CDAudio :: ResumeCD ( )
{
MCI_GENERIC_PARMS mciGeneric;
   mciGeneric.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_RESUME, 0, ( DWORD ) &mciGeneric );
}
void CDAudio :: SeekToStart ( )
{
   return _seek( 0, MCI_SEEK_TO_START );
}
void CDAudio :: SeekToEnd ( )
{
   return _seek( 0, MCI_SEEK_TO_END );
}
bool CDAudio :: IsDisk ( )
{
   return GetStatus ( MCI_STATUS_MEDIA_PRESENT );
}
bool CDAudio :: IsReady ( )
{
   return GetStatus ( MCI_STATUS_READY );
}
void CDAudio :: GetFormat ( )
{
   return GetStatus ( MCI_STATUS_TIME_FORMAT );
}
void CDAudio :: SetFormat ( DWORD dwFormat )
```

```
{
   MCI_SET_PARMS mciSet;
   mciSet.dwTimeFormat = dwFormat;
   ExecCommand ( MCI_SET, MCI_SET_TIME_FORMAT,
               ( DWORD ) ( LPVOID ) &mciSet );
}
DWORD CDAudio :: GetTrackType ( DWORD dwTrack )
{
   return _getTrackInfo ( dwTrack, MCI_CDA_STATUS_TYPE_TRACK );
}
DWORD CDAudio :: GetTrackLen ( DWORD dwTrack )
{
   return _getTrackInfo ( dwTrack, MCI_STATUS_LENGTH );
}
DWORD CDAudio :: GetTrackPosition ( DWORD dwTrack )
{
   return _getTrackInfo ( dwTrack, MCI_STATUS_POSITION );
}
DWORD CDAudio :: GetCountTracks ( )
{
   return GetStatus ( MCI_STATUS_NUMBER_OF_TRACKS );
}
DWORD CDAudio :: GetTotalLenCD ( )
{
   return GetStatus ( MCI_STATUS_LENGTH );
}
DWORD CDAudio :: GetCurPosition ( )
{
   return GetStatus ( MCI_STATUS_POSITION );
}
DWORD CDAudio :: GetCurTrack ( )
{
   return GetStatus ( MCI_STATUS_CURRENT_TRACK );
}
```

На этом наш класс `CDAudio` можно считать полностью законченным. Вам осталось только создать красивый интерфейс и подключить к нему `CDAudio`. Я не буду приводить полный код программы воспроизведения аудиодисков, а покажу только наиболее интересные моменты в создании плеера. Перед началом работы необходимо инициализировать класс `CDAudio`, как показано в листинге 7.5. Сразу хочу заметить, что в примерах подразумевается поддержка библиотеки MFC.

Листинг 7.5. Инициализация класса `CDAudio` и получение информации о треках

```
// предположим, что класс плеера объявлен как CDAudio cd;
// функция инициализации класса CDAudio
bool CMyPlayer :: Init ( )
{
   if ( cd.Open ( ) )
    return true;
   else
   {
    cd.SetWindow ( hMainWnd ); // указываем главное окно программы
    if ( !cd.IsReady ( ) )
    {
      // устройство не готово
    }
    // устанавливаем формат представления данных
    cd.SetFormat ( MCI_FORMAT_TMSF );
    // загружаем информацию о треках в список List View
    LoadTracks ( );
   }
   return false;
}
// функция для загрузки треков в окно программы
// предположим, что наш List View назван m_TrackList
void CMyPlayer :: LoadTracks ( )
{
   CString info;
   MSF msf;
   char Text[30];
   DWORD dwTracks = 0;
   DWORD dwSize = 0;
   // предварительно очищаем список
   m_TrackList.DeleteAllItems ( );
   // определяем общее число треков на диске
   dwTracks = cd.GetCountTracks ( );
   for ( UINT i = 1; i <= dwTracks; i++ )
   {
      // получаем длину первого трека
      msf = cd.GetTrackLength ( i );
      // форматируем строку в удобную для нас форму
      info.Format ( "%02u : %02u", msf.GetMin ( ), msf.GetSec ( ) );
      // создаем имя для трека
      sprintf ( Text, "  Трек %u", i );
```

```
      // выводим информацию в List View
      m_TrackList.SetItemText ( i — 1, 0, Text );
      m_TrackList.SetItemText ( i — 1, 1, info );
      // считаем размер трека в мегабайтах
      dwSize = ( ( msf.GetMin ( ) * 60 ) + msf.GetSec ( ) );
       dwSize *= 75; // число секторов в одной секунде
       dwSize *= 2352; // размер в байтах одного сектора
       sprintf ( Text, "%ld", dwSize / 1048576 );
       m_TrackList.SetItemText ( i — 1, 2, Text );
   }
}
```

Например, чтобы открыть лоток устройства, можно написать функцию, как показано в листинге 7.6.

Листинг 7.6. Пример функции для открывания лотка CD-ROM

```
void CMyPlayer :: Door ( bool bOpen )
{
   if ( bOpen ) // открыть
    cd.OpenTray ( );
   else
    cd.CloseTray ( );
}
```

Реализовать функцию паузы можно так, как это сделано в листинге 7.7. Сделана она таким образом, что позволит включить паузу при первом вызове и продолжить воспроизведение при втором вызове.

Листинг 7.7. Пример функции для приостановки воспроизведения

```
void CMyPlayer :: Pause ( )
{
   if ( cd.GetMode () == MCI_MODE_PLAY ) // идет воспроизведение
   {
     // устанавливаем паузу и останавливаем таймер воспроизведения
     cd.PauseCD ( );
     KillTimer ( MY_TIMER );
   }
   else
   {
      // продолжаем воспроизведение и запускаем таймер
      cd.ResumeCD ( );
```

```
      SetTimer ( MY_TIMER, 1000, NULL );
   }
}
```

Использовать таймер в программе необходимо, если вы хотите отслеживать время воспроизведения и отображать это в своей программе. Для воспроизведения трека можно применить функцию из листинга 7.8.

Листинг 7.8. Пример функции воспроизведения

```
void CMyPlayer :: Play ( BYTE Track )
{
   TMSF m_Start;
   TMSF m_End;
   // определяем длину трека
   MSF msf = cd.GetTrackLen ( Track );
   m_Start = TMSF ( Track, 0, 0, 0 ); // начальная позиция
   m_End = TMSF ( Track, msf.GetMin ( ), msf.GetSec ( ),
                 msf.GetFrm ( ) ); // конечная позиция
   cd.PlayStartEndCD ( m_Start, m_End ); // воспроизведение
   SetTimer ( MY_TIMER , 1000, NULL ); // включаем таймер
}
```

При завершении работы программы следует правильно закрыть все ресурсы. Как это делается, показано в листинге 7.9.

Листинг 7.9. Завершение работы программы

```
void CMyPlayer :: OnCancel ( )
{
   if ( MessageBox ( "Выйти из программы ?" ), "Завершение работы",
        MB_ICONQUESTION | MB_OKCANCEL | MB_DEFBUTTON1 ) == IDOK )
 {
   if (cd.IsReady ( ) )
   {
     cd.StopCD ( ); // останавливаем, если нужно воспроизведение
     cd.Close ( ); // закрываем устройство MCI
     KillTimer ( MY_TIMER ); // включаем таймер
   }
   CDialog :: OnCancel ( );
 }
}
```

И еще я хотел бы рассказать, как организовать быстрое перемещение по музыкальному треку во время воспроизведения. Предположим, что вы добавили ползунок в редакторе ресурсов, назвали его IDC_POSITION и определили для него переменную (имеется в виду MFC) m_Position (класс CSliderCtrl из библиотеки MFC). После этого добавьте обработку сообщения WM_HSCROLL и заполните его так, как это сделано в листинге 7.10. Переменную dwTrackLenght следует определить заранее в классе так, чтобы она определяла полный размер текущего трека, а переменная m_CurTrack должна хранить номер текущего трека.

Листинг 7.10. Обработка сообщения WM_SCROLL

```
void CMyPlayer :: OnHScroll ( UINT nSBCode, UINT nPos,
                              CScrollBar* pScrollBar )
{
   if ( pScrollBar->GetDlgCtrlID ( ) == IDC_POSITION )
   {
     // получаем текущую позицию
     DWORD dwCurPosition = m_Position.GetPos ( );
     // вычисляем новую позицию
     DWORD dwTimePosition = dwTrackLenght / 1000;
     cd.PlayTrackCD ( m_CurTrack, dwCurPosition / 60, dwCurPosition % 60,
                  m_CurTrack, dwTimePosition / 60, dwTimePosition % 60 );
   }
   Cdialog :: OnHScroll ( nSBCode, nPos, pScrollBar );
}
```

На этом можно завершить описание работы с MCI, для программирования устройства компакт-дисков.

7.2. Программирование MIDI

Интерфейс MIDI (Musical Instruments Digital Interface) представляет собой цифровой протокол, основанный на передаче так называемых событий MIDI (MIDI event). С помощью этих событий можно управлять различными устройствами (например, синтезатором), подключенными к игровому порту звуковой карты. Еще раз напомню, что для корректной работы внешнего устройства необходим дополнительный переходник. По протоколу интерфейса MIDI можно подключать последовательно несколько устройств. Для дополнительной поддержки MIDI в 1988 году был разработан специальный файловый формат (Standart MIDI File Format), который четко оговаривает хранение событий MIDI и синхронизирующей информации в одном файле. Поскольку

файл содержит не саму музыку, а только управляющие команды для звукового процессора, размер файла получился очень небольшим. Однако в этом заключается и основной недостаток данного формата: на разных по возможностям звуковых платах файл будет звучать неодинаково. Для того чтобы хоть немного исправить эту проблему, был принят дополнительный стандарт General MIDI, определяющий звучание для 175 инструментов. Теперь любой файл звучит одинаково (по составу инструментов) на любой звуковой плате, однако качество звука все равно может сильно отличаться. Не буду вдаваться в подробности данной проблемы, поскольку книга все-таки рассматривает программирование, а не создание музыки.

Исходя из вышеизложенного, хочу представить вашему вниманию простой и удобный способ воспроизведения MIDI-файлов. Если вы планируете добавить в программу только возможность прослушивания таких файлов, то этот вариант будет наиболее оптимальным.

Итак, создадим новый класс и назовем его `MIDI`. Заполните определение класса, согласно листингу 7.11.

Листинг 7.11. Файл MIDI.h класса `MIDI`

```
#include "Mmsystem.h"
// объявление класса MIDI
class MIDI
{
public:
    // конструктор по умолчанию
    MIDI ( );
    // пустой деструктор
    ~MIDI ( ) { }
   // общие функции
   void Open ( char* szFileMidi ); // загружает файл MIDI, открывает MCI
   void Play ( ); // воспроизводит MIDI-файл
   void Stop ( ); // останавливает воспроизведение
   void Close ( ); // закрывает устройство MCI
private:
// буфер для хранения текстовой строки
char szBuffer[350];
// указатель на загруженный файл
bool bLoad;
}; // окончание класса MIDI
```

Мы определили три функции: для открытия файла, воспроизведения и остановки воспроизведения. Теперь напишем файл реализации класса `MIDI` так, как показано в листинге 7.12.

Листинг 7.12. Файл MIDI.cpp класса `MIDI`

```
#include "stdafx.h"
#include "MIDI.h"
// конструктор по умолчанию
MIDI :: MIDI ( )
{
   bLoad = false;
}
// функция Open
MIDI :: Open ( char* szFileMidi )
{
   // сохраняем в классе указанный файл
   if ( szFileMidi )
   {
     // форматируем командную строку
     sprintf ( szBuffer, "open \"%s\" type sequencer alias MIDIPLAYER",
             szFileMidi );
     // и передаем ее функции MCI
     mciSendString ( szBuffer, 0, 0, 0 );
     bLoad = true;
   }
}
// функция Play
MIDI :: Play ( )
{
   if ( !bLoad ) return;
   // воспроизводим файл MIDI
   mciSendString ( "play MIDIPLAYER from 0", 0, 0, 0 );
}
// функция Stop
MIDI :: Stop ( )
{
   if ( !bLoad ) return;
   // останавливаем воспроизведение файла MIDI
   mciSendString ( "stop MIDIPLAYER", 0, 0, 0 );
}
// функция Close
MIDI :: Close ( )
{
   bLoad = false;
   // закрываем устройство MCI
   mciSendString ( "close all", 0, 0, 0 );
}
```

Вот и все. Класс полностью готов к работе. С помощью него вы сможете прослушать следующие типы файлов: mid, midi и rmi. В качестве основной функции выступает стандартная функция MCI `mciSendString`. Она выполняет те же задачи, что и рассмотренная ранее `mciSendCommand`. Главным отличием является представление аргументов: все управляющие команды записываются в виде текстовой строки.

Написать плеер для воспроизведения MIDI-файлов можно и на базе рассмотренного ранее класса `MCI`. Создадим еще один класс `CMidi`, производный от класса `MCI`, и определим набор функций, как показано в листинге 7.13.

Листинг 7.13. Файл CMidi.h

```
#include "mci.h"
// объявляем класс
class CMidi : public MCI
{
public:
// конструктор по умолчанию и деструктор класса
   CMidi ( );
   ~ CMidi ( ) { }
   // общедоступные функции управления воспроизведением
   void OpenMIDI ( LPCSTR lpszMidiFile ); // открывает устройство MIDI
   void PlayMIDI ( ); // воспроизведение файла MIDI
   void PauseMIDI ( ); // пауза
   void ResumeMIDI ( ); // продолжение воспроизведения после паузы
   void StopMIDI ( );  // остановка воспроизведения
   void SeekToStart ( ); // перейти в начало файла
   void SeekToEnd ( ); // перейти в конец файла
   // функции для получения информации
   bool IsReady ( ); // готово ли устройство
   DWORD GetMidiLen ( ); // длина композиции MIDI
   DWORD GetCurPosition ( ); // текущая позиция воспроизведения
   // дополнительные функции
   DWORD GetFormat ( ); // получить текущий формат времени
   void SetFormatMs ( ); // миллисекунды
   void SetFormatSMPTE_24 ( ); // 24 кадра
   void SetFormatSMPTE_25 ( ); // 25 кадров
   void SetFormatSMPTE_30 ( ); // 30 кадров
   void SetFormatSMPTE_30DROP ( ); // 30 кадров
   DWORD GetMidiTempo ( ); // получить текущий темп
   void SetMidiTempo ( DWORD dwTempo ); // установить новый темп
   void SetMidiCurPort ( ); // выбрать MIDI-порт по умолчанию
```

```
   void SetMidiMapperPort ( ); // выбрать MIDI mapper порт
   void SaveMidi ( LPCSTR lpszMidiFile ); // сохранить в файл
private:
   void _seek ( DWORD dwEnd, DWORD dwFlag ); // установка позиции
}; // окончание класса Cmidi
```

Теперь напишем реализацию функций для класса `CMidi` (листинг 7.14).

Листинг 7.14. Файл CMidi.cpp

```
#include "stdafx.h"
#include "CMidi.h"
// реализация класса CMidi
CMidi :: CMidi ( ) // конструктор класса
{
}
void CMidi :: OpenMIDI ( LPCSTR lpszMidiFile )
{
   MCI_OPEN_PARMS mciOpen;
   mciOpen.lpstrElementName = lpszMidiFile;
   mciOpen.lpstrDeviceType = ( LPCSTR ) MCI_DEVTYPE_SEQUENCER;
   DWORD dwFlag = MCI_WAIT | MCI_OPEN_ELEMENT |
                  MCI_OPEN_TYPE|MCI_OPEN_TYPE_ID;
   ExecCommand ( MCI_OPEN, dwFlag, ( DWORD ) ( LPVOID ) &mciOpen );
   // сохраняем идентификатор открытого устройства
   m_MCIID = mciOpen.wDeviceID;
}
void CMidi :: PlayMIDI ( )
{
   MCI_PLAY_PARMS mciPlay;
   mciPlay.dwCallback = ( DWORD ) m_hWnd;
   DWORD dwFlag |= MCI_NOTIFY;
   ExecCommand ( MCI_PLAY, dwFlag, ( DWORD ) ( LPVOID ) &mciPlay );
}
void CMidi :: PauseMIDI ( )
{
   MCI_GENERIC_PARMS mciGeneric;
   mciGeneric.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_PAUSE, 0, ( DWORD ) &mciGeneric );
}
void CMidi :: ResumeMIDI ( )
{
   MCI_GENERIC_PARMS mciGeneric;
```

```
   mciGeneric.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_RESUME, 0, ( DWORD ) &mciGeneric );
}
void CMidi :: StopMIDI ( )
{
   MCI_GENERIC_PARMS mciGeneric;
   mciGeneric.dwCallback = ( DWORD ) m_hWnd;
   ExecCommand ( MCI_STOP, 0, ( DWORD ) &mciGeneric );
}
void CMidi :: SeekToStart ( )
{
   return _seek ( 0, MCI_SEEK_TO_START );
}
void CMidi :: SeekToEnd ( )
{
   return _seek ( 0, MCI_SEEK_TO_END );
}
void CMidi :: _seek ( DWORD dwEnd, DWORD dwFlag )
{
   MCI_SEEK_PARMS mciSeek;
   mciSeek.dwCallback = ( DWORD ) m_hWnd;
   mciSeek.dwTo = dwEnd;
   dwFlag |= MCI_NOTIFY;
   ExecCommand ( MCI_SEEK, dwFlag, ( DWORD ) &mciSeek );
}
bool CMidi :: IsReady ( )
{
   return GetStatus ( MCI_STATUS_READY );
}
DWORD CMidi :: GetMidiLen ( )
{
   return GetStatus ( MCI_STATUS_LENGTH );
}
DWORD CMidi :: GetCurPosition ( )
{
   return GetStatus ( MCI_STATUS_POSITION );
}
DWORD CMidi :: GetFormat ( )
{
   return GetStatus ( MCI_STATUS_TIME_FORMAT );
}
void CMidi :: SetFormatMs ( )
{
   MCI_SET_PARMS mciSet;
```

```
   mciSet.dwTimeFormat = MCI_FORMAT_MILLISECONDS;
   ExecCommand ( MCI_SET, MCI_SET_TIME_FORMAT,
               ( DWORD ) ( LPVOID ) &mciSet );
}
void CMidi :: SetFormatSMPTE_24 ( )
{
   MCI_SET_PARMS mciSet;
   mciSet.dwTimeFormat = MCI_FORMAT_SMPTE_24;
   ExecCommand ( MCI_SET, MCI_SET_TIME_FORMAT,
               ( DWORD ) ( LPVOID ) &mciSet );
}
void CMidi :: SetFormatSMPTE_25 ( )
{
   MCI_SET_PARMS mciSet;
   mciSet.dwTimeFormat = MCI_FORMAT_SMPTE_25;
   ExecCommand ( MCI_SET, MCI_SET_TIME_FORMAT,
               ( DWORD ) ( LPVOID ) &mciSet );
}
void CMidi :: SetFormatSMPTE_30 ( )
{
   MCI_SET_PARMS mciSet;
   mciSet.dwTimeFormat = MCI_FORMAT_SMPTE_30;
   ExecCommand ( MCI_SET, MCI_SET_TIME_FORMAT,
               ( DWORD ) ( LPVOID ) &mciSet );
}
void CMidi :: SetFormatSMPTE_30DROP ( )
{
   MCI_SET_PARMS mciSet;
   mciSet.dwTimeFormat = MCI_FORMAT_SMPTE_30DROP;
   ExecCommand ( MCI_SET, MCI_SET_TIME_FORMAT,
               ( DWORD ) ( LPVOID ) &mciSet );
}
DWORD CMidi :: GetMidiTempo ( )
{
   return GetStatus ( MCI_SEQ_STATUS_TEMPO );
}
void CMidi :: SetMidiTempo ( DWORD dwTempo )
{
   return GetStatus ( MCI_SEQ_STATUS_TEMPO );
}
void CMidi :: SetMidiCurPort ( )
{
   MCI_SEQ_SET_PARMS mciSeqSet;
   mciSeqSet.dwPort = MCI_SEQ_NONE;
```

```
   ExecCommand ( MCI_SET, MCI_SEQ_SET_PORT,
                ( DWORD ) ( LPVOID ) &mciSeqSet );
}
void CMidi :: SetMidiMapperPort ( )
{
   MCI_SEQ_SET_PARMS mciSeqSet;
   mciSeqSet.dwPort = MIDI_MAPPER;
   ExecCommand ( MCI_SET, MCI_SEQ_SET_PORT,
                ( DWORD ) ( LPVOID ) &mciSeqSet );
}
void CMidi :: SaveMidi ( LPCSTR lpszMidiFile )
{
   MCI_SAVE_PARMS mciSave;
   mciSave.lpfilename = lpszMidiFile;
   ExecCommand ( MCI_SAVE, MCI_SAVE_FILE, ( DWORD ) &mciSave );
}
```

Вот у нас и получился полноценный класс для воспроизведения MIDI-файлов. На этом работу с MIDI будем считать завершенной.

7.3. Доступ к файлам в формате MP3

Говорить о данном формате не имеет смысла, поскольку с ним знаком практически каждый. Как реализовать поддержку кодирования и декодирования файлов MP3 в программе, говорить не буду, поскольку эту информацию можно найти в Интернете. Мы поговорим о том, как воспроизвести файл MP3, а также как модифицировать информационный заголовок в таком файле.

Для воспроизведения MP3 воспользуемся уже знакомым нам интерфейсом MCI. Напишем новый класс плеера и назовем его `CMp3`. В опции компоновщика следует добавить ссылку на библиотеку Vfw32.lib. Заголовочный файл нашего класса представлен в листинге 7.15.

Листинг 7.15. Файл CMp3.h

```
#include <vfw.h>
// объявление класса CMp3
class CMp3
{
public:
   CMp3 ( ); // конструктор по умолчанию
   ~CMp3 ( ); // деструктор
```

```
   // общие функции управления
   void Open ( ); // открыть файл MP3
   void Play ( ); // воспроизведение файла
   void Pause ( ); // пауза
   void Stop ( ); // стоп
   // установить параметры родительского окна
   void SetWinParam ( HWND hWnd, HINSTANCE hInst );
private:
    HWND m_hWnd; // дескриптор родительского окна
    HINSTANCE m_hInst; // дескриптор вызывающего приложения
    HWND m_MP3; // дескриптор устройства
    bool m_bPause; // статус воспроизведения
};
```

Теперь напишем реализацию класса, согласно листингу 7.16.

Листинг 7.16. Файл CMp3.cpp

```
#include "stdafx.h"
#include "CMp3.h"
#include "vfw.h"
// реализация класса CMp3
CMp3 :: CMp3 ( )
{
   m_hWnd = NULL;
   m_MP3 = NULL;
   m_hInst = NULL;
   m_bPause = false;
}
CMp3 :: ~CMp3 ( )
{
   // освобождаем ресурсы
   if ( m_MP3 ) MCIWndDestroy ( m_MP3 );
}
void CMp3 :: SetWinParam ( HWND hWnd, HINSTANCE hInst )
{
  if ( hWnd && hInst )
  {
    m_hWnd = hWnd;
    m_hInst = hInst;
  }
}
```

```
void CMp3 :: Open ( )
{
   if ( ( !m_hWnd ) && ( !m_hInst ) ) return;
   CFileDialog file ( TRUE, NULL, NULL, OFN_HIDEREADONLY,
                      "MP3 Files (*.mp3)|*.mp3||" );
   if ( file.DoModal ( ) == IDOK )
   {
     // открываем устройство MCI
     m_MP3 = MCIWndCreate ( m_hWnd, m_hInst,WS_CHILD
            | MCIWNDF_NOMENU | MCIWNDF_NOPLAYBAR, file.m_ofn.lpstrFile );
   }
}
void CMp3 :: Play ( )
{
   if( !m_MP3 ) return;
   MCIWndHome ( m_MP3 ); // исходное положение файла
   MCIWndPlay ( m_MP3 ); // начинаем воспроизведение
}
void CMp3 :: Pause ( )
{
   if ( !m_MP3 ) return;
   if ( m_bPause )
   {
     MCIWndResume ( m_MP3 );
     m_bPause = false;
   }
   else
   {
     MCIWndPause ( m_MP3 );
     m_bPause = true;
   }
}
void CMp3 :: Stop ( )
{
   if( !m_MP3 ) return;
   MCIWndStop ( m_MP3 );
   m_bPause = false;
}
```

Вот у нас и получился простейший класс для воспроизведения файлов в формате MP3. Все достаточно наглядно и не требует дополнительных комментариев. Хочу сказать только два слова о назначении функции `SetWinParam`. С помощью данной функции мы устанавливаем для нашего класса дескрип-

торы родительского окна и приложения, поскольку интерфейс MCI требует этого. После объявления класса `CMp3` следует сразу вызвать функцию `SetWinParam`, а только потом пользоваться управляющими функциями.

А теперь рассмотрим, как можно прочитать и модифицировать информационный заголовок файла MP3. Для этого потребуется написать класс `CMp3Info`, но прежде познакомимся с заголовком файла MP3. Формат заголовка показан в табл. 7.3.

Таблица 7.3. *Заголовок файла MP3*

Байт	Биты	Описание
0	8	Биты синхронизации установлены в 1
1	3	Биты синхронизации установлены в 1
	2	Номер версии
	2	Индекс уровня (1 — Layer 3, 2 — Layer 2, 3 — Layer 1)
	1	Защита контрольной суммой (CRC)
2	4	Скорость передачи (битов в секунду)
	2	Частота дискретизации
	1	При установке в 1 используется добавочный слот
	1	Частный
3	2	Режим (0 — стерео, 1 — объединенное стерео, 2 — два канала, 3 — один канал)
	2	Расширенный режим
	1	Авторское право (1 — есть)
	1	Оригинал (1 — оригинальная композиция)
	2	Предыскажение

Кроме перечисленного в таблице, файл может содержать дополнительные сведения (128 байт): полное имя файла, данные об исполнителе, название альбома, стиль композиции и дату создания.

Рассмотрим первый файл класса (CMp3Info.h), представленный в листинге 7.17.

Листинг 7.17. Файл CMp3Info.h класса `CMp3Info`

```
// сначала определим список музыкальных стилей
static char* Genres [] = { "Blues", "Classic Rock", "Country", "Dance",
"Disco", "Funk", "Grunge", "Hip-Hop", "Jazz", "Metal", "New Age",
```

```
"Oldies", "Other", "Pop", "R&B", "Rap", "Reggae", "Rock", "Techno",
"Industrial", "Alternative", "Ska", "Death Metal", "Pranks",
"Soundtrack", "Euro-Techno", "Ambient", "Trip Hop", "Vocal", "Jazz+Funk",
"Fusion", "Trance", "Classical", "Instrumental", "Acid", "House", "Game",
"Sound Clip", "Gospel", "Noise", "Alternative Rock", "Bass", "Soul",
"Punk", "Space", "Meditative", "Instrumental Pop", "Instrumental Rock",
"Ethnic", "Gothic", "Darkwave", "Techno-Industrial", "Electronic",
"Pop-Folk", "Eurodance", "Dream", "Southern Rock", "Comedy", "Cult",
"Gangsta Rap", "Top 40", "Christian Rap", "Pop/Punk", "Jungle",
"Native American", "Cabaret", "New Wave", "Psychedelic", "Rave",
"Showtunes", "Trailer", "Lo-Fi", "Tribal", "Acid Punk", "Acid Jazz",
"Polka", "Retro", "Musical", "Rock & Roll", "Hard Rock", "Folk",
"Folk/Rock", "National Folk", "Swing", "Fast-Fusion", "Bebob", "Latin",
"Revival", "Celtic", "Blue Grass", "Avantegarde", "Gothic Rock",
"Progressive Rock", "Psychedelic Rock", "Symphonic Rock", "Slow Rock",
"Big Band", "Chorus", "Easy Listening", "Acoustic", "Humour", "Speech",
"Chanson", "Opera", "Chamber Music", "Sonata", "Symphony", "Booty Bass",
"Primus", "Porn Groove", "Satire", "Slow Jam", "Club", "Tango", "Samba",
"Folklore", "Ballad", "power Ballad", "Rhythmic Soul", "Freestyle",
"Duet", "Punk Rock", "Drum Solo", "A Capella", "Euro-House",
"Dance Hall", "Goa", "Drum & Bass", "Club-House", "Hardcore", "Terror",
"Indie", "Brit Pop", "Negerpunk", "Polsk Punk", "Beat",
"Christian Gangsta Rap", "Heavy Metal", "Black Metal", "Crossover",
"Contemporary Christian", "Christian Rock", "Merengue", "Salsa",
"Trash Metal", "Anime", "JPop", "Synth Pop" };
// количество стилей
#define GENRES_COUNT ( ( int ) ( ( sizeof Genres ) / ( sizeof
                      Genres[0] ) ) )
// частота дискретизации
static int iSampleRate[3][3] =
{
  { 32000, 16000,  8000 }, { 22050, 24000, 16000 },
  { 44100, 48000, 32000 }
};
// скорость передачи данных
static int iBitRate [6] [16] =
{
{ 0, 8, 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 128, 144, 160, 0 },
{ 0, 8, 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 128, 144, 160, 0 },
{ 0, 32, 48, 56, 64, 80, 96, 112, 128, 144, 160, 176, 192, 224, 256, 0 },
{ 0, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320, 0 },
{ 0, 32, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320, 384, 0 },
{ 0, 32, 64, 96, 128, 160, 192, 224, 256, 288, 320, 352, 384, 416, 448,
0 },
};
```

```
class CMp3Info
{
public:
   CMp3Info ( );
   virtual ~CMp3Info ( );
   // функция для загрузки файла MP3
   void Open ( char* szMp3File );
   // сохранить информацию в файл MP3
   void Save ( char* szMp3File );
   // функции для получения текущих значений
   int GetVersion ( ) const; // номер версии
   int GetLayer ( ) const; // номер уровня
   bool IsCRC ( ) const; // защита CRC
   int GetBitrate ( ) const; // скорость передачи данных
   int GetSampleRate ( ) const; // частота дискретизации
   bool IsPrivate ( ) const; // частный
   int GetMode ( ) const; // режим
   bool IsCopyright ( ) const; // авторское право
   bool IsOriginal ( ) const; // оригинал
   DWORD GetSize ( ) const; // размер файла
   void GetTitle ( char* buffer ) const; // название композиции
   void GetArtist ( char* buffer ) const; // имя исполнителя
   void  GetAlbum ( char* buffer ) const; // название альбома
   void GetYear ( char* buffer ) const; // год записи альбома
   void GetComment ( char* buffer ) const; // комментарии
   void GetGenre ( char* buffer ) const; // стиль музыки
   // функции для установки параметров
   void SetTitle ( char* pszTitle );
   void  SetArtist ( char* pszArtist );
   void  SetAlbum ( char* pszAlbum );
   void SetYear ( char* pszYear );
   void SetComment ( char* pszComment );
   void SetGenre ( int iGenre );
private:
   // номер версии формата
   enum MPEG_VERSION
   {
     MPEG_25, // 2.5
     MPEG_0, // резерв
     MPEG_2, // 2.0
     MPEG_1 // 1.0
   };
```

```
// номер уровня пересмотра
enum MPEG_LAYER
{
  LAYER_0, // резерв
  LAYER_3, // Layer 3
  LAYER_2, // Layer 2
  LAYER_1 // Layer 1
};
// тип режима
enum MPEG_MODE
{
  STEREO, // Stereo
  JOINT_STEREO, // Joint Stereo
  DUAL_CHANNEL, // Dual Channel
  SINGLE_CHANNEL // Single Channel
};
// структура для хранения заголовка файла MP3
struct MP3HEADER
{
    int iSync;
    int iVersion;
    int iLayer;
    int iCRC;
    int iBitrate;
    int iSampleRate;
    int iPad;
    int iPrivate;
    int iMode;
    int iModeExt;
    int iCopy;
    int iOriginal;
    int Emp;
    DWORD dwSize;
};
MP3HEADER hdr_Mp3;
char szTitle[30]; // название композиции
char szArtist[30]; // имя исполнителя
char szAlbum[30]; // название альбома
char szYear[4]; // год записи альбома
char szComment[30]; // комментарии
int m_iGenre; // стиль музыки
// функция для получения указанного параметра
unsigned int _getValue ( DWORD dvValue, unsigned int start,
                         unsigned int len );
```

```
   char* _rightTrim ( char* str ); // удаление пробелов справа в строке
}; // окончание класса CMp3Info
```

А теперь реализуем наш класс, как показано в листинге 7.18.

Листинг 7.18. Файл CMp3Info.cpp класса `CMp3Info`

```
#include "stdafx.h"
#include "Mp3Info.h"
CMp3Info :: CMp3Info ( )
{
  iGenre = 0;
}
CMp3Info :: ~CMp3Info ( )
{
}
unsigned int CMp3Info :: getValue ( DWORD dvValue, unsigned int start,
                                    unsigned int len )
{
   return ( dvValue >> ( start — 1 ) ) & ( ( 1 << len ) — 1 );
}
char* CMp3Info :: _rightTrim ( char* str )
{
   int i = strlen ( str );
   while ( ( --i ) > 0 && isspace ( str [i] ) )
   {
     str[i] = '\0';
   }
   return ( str );
}
void CMp3Info :: Open ( char* szMp3File )
{
   HANDLE hFile = NULL;
   // открываем MP3-файл
   if ( ( hFile = CreateFile ( szMp3File, GENERIC_READ, FILE_SHARE_READ,
          NULL, OPEN_EXISTING, 0 , NULL ) ) != INVALID_HANDLE_VALUE )
    {
     DWORD dwInfo = 0, dwReadBytes = 0, dwFileSize = 0;
     char szTemp[40];
     // получаем размер файла
     dwFileSize = GetFileSize ( hFile, NULL );
     hdr_Mp3.dwSize = dwFileSize;
```

```
// устанавливаем маркер чтения на начало файла
SetFilePointer ( hFile, 0, 0, FILE_BEGIN );
// читаем заголовок файла размером 4 байта ( используются 32 бита )
ReadFile ( hFile, &szTemp, 4, &dwReadBytes, NULL );
// сохраняем из буфера нужные данные
dwInfo = ( DWORD ) ( ( ( szTemp[0] & 255 ) << 24 ) |
( ( szTemp[1] & 255 ) << 16 ) | ( ( szTemp[2] & 255 ) << 8 ) |
                                ( ( szTemp[3] & 255 ) ) );
// распаковываем данные и сохраняем в структуру
hdr_Mp3.iSync = _getValue ( dwInfo, 22, 11 );
hdr_Mp3.iVersion  = _getValue ( dwInfo, 20, 2 );
hdr_Mp3.iLayer = _getValue ( dwInfo, 18, 2 );
hdr_Mp3.iCRC = _getValue ( dwInfo, 17, 1 );
hdr_Mp3.iBitrate = _getValue ( dwInfo, 13, 4 );
hdr_Mp3.iSampleRate = _getValue ( dwInfo, 11, 2 );
hdr_Mp3.iPad = _getValue ( dwInfo, 10, 1 );
hdr_Mp3.iPrivate = _getValue ( dwInfo, 9,  1 );
hdr_Mp3.iMode = _getValue ( dwInfo, 7,  2 );
hdr_Mp3.iModeExt = _getValue ( dwInfo, 5,  2 );
hdr_Mp3.iCopy = _getValue ( dwInfo, 4,  1 );
hdr_Mp3.iOriginal = _getValue ( dwInfo, 3,  1 );
hdr_Mp3.Emp = _getValue ( dwInfo, 1,  2 );
// проверяем наличие дополнительной информации
SetFilePointer ( hFile, -128, 0, FILE_END );
szTemp[3] = '\0';
ReadFile ( hFile, szTemp, 3, &dwReadBytes, NULL );
 if ( !( strcmp ( szTemp, "TAG" ) ) )
 {
  // есть дополнительная информация
  szTemp[30] = '\0';
  // название композиции
  ReadFile ( hFile, szTemp, 30, &dwReadBytes, NULL );
  strcpy ( szTitle, _rightTrim ( szTemp ) );
  // имя исполнителя
  szTemp[30] = '\0';
  ReadFile ( hFile, szTemp, 30, &dwReadBytes, NULL );
  strcpy ( szArtist, _rightTrim ( szTemp ) );
  // название альбома
  szTemp[30] = '\0';
  ReadFile ( hFile, szTemp, 30, &dwReadBytes, NULL );
  strcpy ( szAlbum, _rightTrim ( szTemp ) );
  // год записи альбома
  szTemp[4] = '\0';
```

```
       ReadFile ( hFile, szTemp, 4, &dwReadBytes, NULL );
       strcpy ( szYear, _rightTrim ( szTemp ) );
       // комментарии
       szTemp[30] = '\0';
       ReadFile ( hFile, szTemp, 30, &dwReadBytes, NULL );
       strcpy ( szComment, _rightTrim ( szTemp ) );
       // стиль музыки
       m_iGenre = GENRES_COUNT + 1;
       ReadFile ( hFile, szTemp, 1, &dwReadBytes, NULL );
       m_iGenre = szTemp[0];
      }
     // закрываем файл
     CloseHandle ( hFile );
    }
}
int CMp3Info :: GetVersion ( )
{
   switch ( hdr_Mp3.iVersion )
   {
    case 0: // 2.5
      return MPEG_25;
    case 1: // reserv
      return MPEG_0;
    case 2: // 2.0
      return MPEG_2;
    case 3: // 1.0.
      return MPEG_1;
   }
   return -1;
}
int CMp3Info :: GetLayer ( ) const
{
   switch ( hdr_Mp3.iLayer  )
   {
     case 0: // резерв
       return LAYER_0;
     case 1: // Layer 3
       return LAYER_3;
     case 2: // Layer 2
       return LAYER_2;
     case 3: // Layer 1
       return LAYER_1;
   }
```

```
   return -1;
}
bool CMp3Info :: IsCRC ( ) const
{
    if ( hdr_Mp3.iCRC )
      return true;
    return false;
}
int CMp3Info :: GetBitrate ( ) const
{
switch ( hdr_Mp3.iVersion )
  {
  case MPEG_1:
    {
     switch ( hdr_Mp3.iLayer )
           {
            case LAYER_1:
              return iBitRate[5] [hdr_Mp3.iBitrate];
            case LAYER_2:
              return iBitRate[4] [hdr_Mp3.iBitrate];
            case LAYER_3:
              return iBitRate[3] [hdr_Mp3.iBitrate];
           }
         }
  case MPEG_2:
  case MPEG_25:
   {
     switch ( hdr_Mp3.iLayer )
       {
           case LAYER_1:
               return iBitRate[2] [hdr_Mp3.iBitrate];
           case LAYER_2:
               return iBitRate[1] [hdr_Mp3.iBitrate];
           case LAYER_3:
               return iBitRate[0] [hdr_Mp3.iBitrate];
       }
   }
   return -1;
}
int CMp3Info :: GetSampleRate ( ) const
{
switch ( hdr_Mp3.iVersion )
  {
    case MPEG_1:
```

```
      {
        switch ( hdr_Mp3.iSampleRate )
           {
             case 0:
                return iSampleRate[3] [0];
             case 1:
                return iSampleRate[3] [1];
             case 2:
                return iSampleRate[3] [2];
           }
      }
    case MPEG_2:
      {
        switch ( hdr_Mp3.iSampleRate )
           {
             case 0:
                return iSampleRate[2] [0];
             case 1:
                return iSampleRate[2] [1];
             case 2:
                return iSampleRate[2] [2];
           }
       }
     case MPEG_25:
       {
         switch ( hdr_Mp3.iSampleRate )
            {
              case 0:
                 return iSampleRate[1] [0];
              case 1:
                 return iSampleRate[1] [1];
              case 2:
                 return iSampleRate[1] [2];
            }
      }
  }
   return -1;
}
bool CMp3Info :: IsPrivate ( ) const
{
   if ( hdr_Mp3.iPrivate )
        return true;
    return false;
}
```

```
int CMp3Info :: GetMode ( ) const
{
   switch ( hdr_Mp3.iMode  )
   {
     case 0:
        return STEREO;
     case 1:
        return JOINT_STEREO;
     case 2:
        return DUAL_CHANNEL;
     case 3:
        return SINGLE_CHANNEL;
   }
   return -1;
}
bool CMp3Info :: IsCopyright ( ) const
{
   if ( hdr_Mp3.iCopy )
        return true;
    return false;
}
bool CMp3Info :: IsOriginal() const
{
   if ( hdr_Mp3.iOriginal )
        return true;
    return false;
}
DWORD CMp3Info :: GetSize ( ) const
{
   return hdr_Mp3.dwSize;
}
void CMp3Info :: GetTitle ( char* buffer ) const
{
   strcpy ( buffer, szTitle );
}
void CMp3Info :: GetArtist ( char* buffer ) const
{
   strcpy ( buffer, szArtist );
}
void CMp3Info :: GetAlbum ( char* buffer ) const
{
   strcpy ( buffer, szAlbum );
}
```

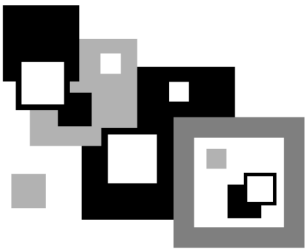
```
void CMp3Info :: GetYear ( char* buffer ) const
{
   strcpy ( buffer, szYear );
}
void CMp3Info :: GetComment ( char* buffer ) const
{
   strcpy ( buffer, szComment );
}
void CMp3Info :: GetGenre ( char* buffer ) const
{
   if ( m_iGenre > GENRES_COUNT )
     strcpy ( buffer, "Unknown" );
   else
     strcpy ( buffer, Genres[m_iGenre] );
}
void CMp3Info :: SetTitle ( char* pszTitle )
{
   if ( strlen ( pszTitle ) > 30 ) return;
     strcpy ( szTitle, pszTitle );
}
void CMp3Info :: SetArtist ( char* pszArtist )
{
   if ( strlen ( pszArtist ) > 30 ) return;
     strcpy ( szArtist, pszArtist );
}
void CMp3Info :: SetAlbum ( char* pszAlbum )
{
   if ( strlen ( pszAlbum ) > 30 ) return;
     strcpy ( szAlbum, pszAlbum );
}
void CMp3Info :: SetYear ( char* pszYear )
{
   if ( strlen ( pszYear ) > 4 ) return;
     strcpy ( szYear, pszYear );
}
void CMp3Info :: SetComment ( char* pszComment )
{
   if ( strlen ( pszComment ) > 30 ) return;
     strcpy ( szComment, pszComment );
}
void CMp3Info :: SetGenre ( int iGenre )
{
   if ( ( iGenre < 0 ) && ( iGenre > GENRES_COUNT ) ) return;
```

```
   m_iGenre = iGenre;
}
void CMp3Info :: Save ( char* szMp3File )
{
   if ( szMp3File[0] != '\0' ) return;
   // открываем файл для записи
   HANDLE hFile = NULL;
   char szTemp[30];
   DWORD dwWriteBytes = 0;
   hFile = CreateFile ( szMp3File, GENERIC_WRITE, FILE_SHARE_WRITE, NULL,
                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0 );
   // устанавливаем позицию для записи в файл
   SetFilePointer ( hFile, -128, 0, FILE_END );
   // записываем информационный указатель
   strcpy ( szBuffer, "TAG\0" );
   WriteFile ( hFile, szTemp, 3, & dwWriteBytes, NULL );
   // название композиции
   WriteFile ( hFile, szTitle, 30, &dwWriteBytes, NULL );
   // имя исполнителя
   WriteFile( hFile, szArtist, 30, &dwWriteBytes, NULL );
   // название альбома
   WriteFile ( hFile, szAlbum, 30, &dwWriteBytes, NULL );
   // год выпуска
   WriteFile ( hFile, szYear, 4, &dwWriteBytes, NULL );
   // комментарии
   WriteFile ( hFile, szComment, 30, &dwWriteBytes, NULL );
   // стиль музыки
   szTemp[0] = ( char ) m_iGenre;
   szTemp[1] = '\0';
   WriteFile ( hFile, szTemp, 1, &dwWriteBytes, NULL );
   // закрываем файл
   CloseHandle ( hFile );
}
```

Использование класса `CMp3Info` не должно вызвать никаких вопросов. Вначале объявляете его в своей программе, а далее вызываете функцию `Open`. После того как файл MP3 будет открыт, можно пользоваться остальными функциями. Перед тем как сохранить файл с помощью `Save`, необходимо заполнить информационные поля, вызывая поочередно соответствующие функции.

На этом я хотел бы завершить данную главу. Многие интересные темы остались "за бортом", но даже представленная здесь информация поможет вам быстрее разобраться с программированием звука в Windows.

ГЛАВА 8



Системный динамик

Системный динамик является таким же "древним" устройством, как и сам *x*86-совместимый компьютер. Он представляет собой маленький динамик, расположенный в системном корпусе и подключаемый к нескольким дополнительным электронным компонентам, размещенным на материнской плате. В прошлом веке, когда не было звуковых плат, именно системный динамик, или как его еще называют — *спикер*, выполнял основные функции по извлечению звуков. Большинство старых игрушек под DOS использовали спикер для озвучивания различных игровых ситуаций: от стрельбы до звуков летящего самолета. К сожалению, качество звука оставляло желать лучшего. Потом появились отдельные устройства для воспроизведения звука, и спикер перестали применять в игровых и мультимедийных программах. Однако и по сей день на подавляющем большинстве компьютерных систем установлен маленький круглый динамик. При каждой загрузке компьютера раздается одиночный звуковой сигнал, подтверждающий успешное тестирование оборудования и выполнение операции начальной загрузки (*POST* — Power-On Self Test). Для этого используется именно спикер, поскольку ни звуковая карта, ни любое другое устройство еще не может полноценно функционировать. Разработана целая комбинация различных звуковых кодов, позволяющих выявить сбои в подключенном оборудовании. Именно поэтому системный динамик стоит как в 486 простеньком компьютере, так и в современном "навороченном" мультимедийном "монстре" на базе двухъядерного или четырехъядерного процессора. Вы можете спросить, а зачем нужен спикер в операционных системах Windows, если там есть нормальная звуковая плата. Ответ очень прост: системный динамик гарантированно установлен на большинстве компьютеров и его можно применять для вывода системных сообщений и простых звуковых эффектов. Это, несомненно, даст вашей программе преимущество по сравнению с другими. В любом случае, вам самим ре-

шать, стоит ли использовать возможности системного динамика или нет, а я просто познакомлю вас с основными методами программирования спикера.

8.1. Программирование системного динамика

Для доступа к системному динамику используется порт с номером 61h. Он имеет размер 8 бит, но для управления спикером применяются только два младших бита (0 и 1): нулевой бит управляет включением канала 2 системного таймера, а первый бит включает динамик. Программирование системного таймера рассматривается в *главе 10*. Установка этих битов в 1 позволяет включить динамик, а сброс — выключить. Рассмотрим пример для выключения спикера (листинг 8.1).

Листинг 8.1. Выключение системного динамика

```
// пишем функцию управления динамиком
void Speaker ( bool bOn )
{
   DWORD dwResult = 0;
   // читаем состояние порта
   inPort ( 0x61, &dwResult, 1 );
   if ( bOn ) // включить динамик
   {
    dwResult |= 0x03;
    // записываем значение в порт
    outPort ( 0x61, dwResult, 1 );
   }
   else // выключить динамик
   {
    dwResult &= 0xFC;
    outPort ( 0x61, dwResult, 1 );
   }
}
// пишем реализацию работы спикера
Speaker ( true ); // включаем динамик
delay ( 3000 ); // пауза 3 секунды, можно использовать функцию Sleep
Speaker ( false ); // выключаем динамик
```

Рассмотренные функции управляют только включением и выключением динамика, сам же выводимый звуковой сигнал не изменяется по частоте. Для

решения этой задачи можно воспользоваться регистром управления. Пример для настройки частоты выводимого звука в герцах показан в листинге 8.2.

Листинг 8.2. Управление частотой звука посредством системного таймера

```
// реализуем функцию управления частотой
void SetFrequence ( unsigned int uFrequence )
{
   DWORD dwResult = 0;
   if ( uFrequence <= 0 ) return;
   int TimerClock = 1193180; // внутренняя частота таймера
   int iValue = 0; // значение делителя частоты
   // определяем значение делителя
   iValue = TimerClock / uFrequence;
   // настраиваем регистр управления системного таймера
   outPort ( 0x43, 0xB6, 1 ); // канал 2, 2 байта, прямоугольные импульсы
   dwResult = nDivider % 256; // младший байт делителя частоты
   outPort ( 0x42, dwResult, 2 ); // записываем значение в порт динамика
   dwResult = nDivider / 256; // старший байт делителя частоты
   outPort ( 0x42, dwResult, 2 ); // записываем значение в порт динамика
}
// попробуем вывести звуковой сигнал частотой примерно 1000 Гц
SetFrequence ( 1193 ); // устанавливаем желаемую частоту
Speaker ( true ); // включаем динамик
Sleep ( 2000 ); // устанавливаем длительность сигнала равным 2 секунды
Speaker ( true ); // выключаем динамик
```

Как видите, нет ничего сложного в программировании системного динамика. Используя порты таймера для задания частоты, можно воспроизводить любые мелодии. Для тех, кто захочет написать полноценное музыкальное произведение, приведу значения частот всех нот в табл. 8.1.

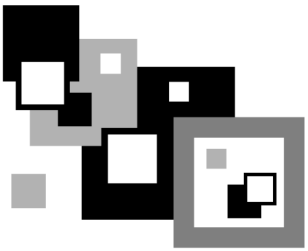
Таблица 8.1. *Список частот*

Нота	Значение частоты, Гц
До	4186
До-диез	4435
Ре	4699
Ре-диез	4978
Ми	5274
Фа	5588

Таблица 8.1 *(окончание)*

Нота	Значение частоты, Гц
Фа-диез	5920
Соль	6272
Соль-диез	6645
Ля	7040
Ля-диез	7459
Си	7902

ГЛАВА 9



Часы реального времени

Каждый компьютер, как минимум, ассоциируется с точностью. А точность, так или иначе, подразумевает постоянный отсчет времени. Для реализации этой задачи используется специальный модуль, называемый *часами реального времени* (*RTC* — Real Time Clock). Именно эти часы позволяют нам отслеживать дату и время, а компьютеру помогают упорядочить свою работу и, кроме того, два раза в год предупреждать нас о переводе стрелок часов вперед или назад. Но, как вы уже наверняка заметили, компьютер при необходимости выключается, а после включения все временные (и не только) параметры восстанавливаются в соответствии с текущим временем. Для этого в модуле RTC имеется небольшая микросхема памяти размером от 64 до 128 байт. Выполненная по специальной технологии *CMOS* (Complementary Metal-Oxide Semiconductor), эта микросхема потребляет очень мало энергии (в качестве источника используется литиевая батарейка) и может годами сохранять постоянные значения. Поэтому сюда записывается различная системная информация, в том числе и время. После включения компьютера операционная система считывает данные из CMOS и в соответствии с ними настраивает текущие значения даты и времени.

Не буду вдаваться в детали, скажу только, что аппаратная организация RTC базируется на задающем генераторе синусоидального сигнала частотой 32 кГц (точнее 32,768 кГц). Для подпитки применяется высококачественная батарейка (Intel рекомендует использовать элементы питания фирмы Duracell) с высокой емкостью заряда (170 мА). Для расчета времени разряда можно емкость батарейки разделить на среднее значение тока разряда (5 мкА) и мы получим значение времени, измеряемое в часах (34000 часов или 3,88 года). Поскольку точность часов реального времени напрямую зависит от напряжения питания, то следует периодически (раз в год) проверять параметры батарейки и при необходимости делать замену. Конечно, для домашних компьютеров небольшая потеря точности часов не принесет никаких

проблем, но для производственных систем с компьютерным управлением, а тем более работающих в режиме реального времени, любая неточность системных часов может привести к сбою всего технологического процесса.

Для программирования часов реального времени мы воспользуемся портами ввода-вывода.

9.1. Использование портов

Для доступа к часам реального времени используются всего два порта: 70h и 71h. Порт 70h используется только для записи и позволяет выбрать адрес регистра в CMOS-памяти. Порт 71h применяется как для записи, так и для чтения данных из указанного (через порт 70h) регистра CMOS. Оба порта являются 8-разрядными. Кроме того, бит 7 в порту 70h не относится к работе с RTC, а управляет режимом немаскируемых прерываний (1 — прерывания запрещены). Если у вас будут проблемы с доступом к регистрам CMOS, следует запрещать прерывания (команда CLI) перед началом работы и разрешать прерывания (команда STI) после. Но, как правило, это не требуется.

Данные в CMOS имеют упорядоченную структуру. Несмотря на полный размер памяти (от 64 до 128 байт), стандартизированы только первые 51 байт. Остальные зависят от производителя материнской платы и здесь рассматриваться не будут. Каждое смещение байта в CMOS-памяти определяет номер регистра, через который можно считывать и записывать информацию. Для RTC выделены первые 13 регистров, а остальные служат для других целей. Формат микросхемы памяти представлен в табл. 9.1.

Таблица 9.1. *Формат памяти CMOS*

Адрес регистра	Описание
00h	Текущее значение секунды (00h—59h в формате BCD или 00h—3Bh)
01h	Значение секунд будильника (00h—59h в формате BCD или 00h—3Bh)
02h	Текущее значение минуты (00h—59h в формате BCD или 00h—3Bh)
03h	Значение минут будильника (00h—59h в формате BCD или 00h—3Bh)
04h	Текущее значение часа: 24-часовой режим (00h—23h в формате BCD или 00h—17h), 12-часовой режим AM (01h—12h в формате BCD или 00h—0Ch), 12-часовой режим PM (81h—92h в формате BCD или 81h—8Ch)
05h	Значение часа будильника: 24-часовой режим (00h—23h в формате BCD или 00h—17h), 12-часовой режим AM (01h—12h в формате BCD или 00h—0Ch), 12-часовой режим PM (81h—92h в формате BCD или 81h—8Ch)

Таблица 9.1 *(продолжение)*

Адрес регистра	Описание
06h	Текущее значение дня недели (01h—07h в формате BCD или 01h—07h, где 01h соответствует воскресенью)
07h	Текущее значение дня месяца (01h—31h в формате BCD или 01h—1Fh)
08h	Текущее значение месяца (01h—12h в формате BCD или 01h—0Ch)
09h	Текущее значение года (00h—99h в формате BCD или 00h—63h)
0Ah	Регистр состояния A: бит 7 — обновление времени (1 — идет обновление времени, 0 — доступ разрешен), биты 4—6 — делитель частоты (010b — 32,768 кГц), биты 0—3 — значение пересчета частоты (0110b — 1024 Гц)
0Bh	Регистр состояния B: бит 7 — запрещение обновления часов (1 — идет установка, 0 — обновление разрешено), бит 6 — разрешение периодического прерывания IRQ8 (1 — разрешено, 0 — запрещено), бит 5 — разрешение прерывания от будильника (1 — разрешено, 0 — запрещено), бит 4 — вызов прерывания после цикла обновления (1 — разрешено, 0 — запрещено), бит 3 — разрешение генерации прямоугольных импульсов (1 — разрешено, 0 — запрещено), бит 2 — выбор формата представления даты и времени (1 — двоичный, 0 — BCD), бит 1 — выбор часового режима (1 — 24 часа, 0 — 12 часов), бит 0 — автоматический переход на летнее время (1 — разрешено, 0 — запрещено)
0Ch	Регистр состояния C (только для чтения): бит 7 — признак выполненного прерывания (1 — произошло, 0 — не было), бит 6 — периодическое прерывание (1 — есть, 0 — нет), бит 5 — прерывание от будильника (1 — есть, 0 — нет), бит 4 — прерывание после обновления часов (1 — есть, 0 — нет), биты 0—3 зарезервированы и должны быть равны 0
0Dh	Регистр состояния D: бит 7 — состояние батареи и памяти (1 — память в норме, 0 — батарейка разряжена)
0Eh	Состояние POST после загрузки компьютера: бит 7 — сброс часов по питанию (1 — нет питания), бит 6 — ошибка контрольной суммы (CRT) в CMOS-памяти (1 — ошибка), бит 5 — несоответствие конфигурации оборудования (1 — POST обнаружила ошибки конфигурации), бит 4 — ошибка размера памяти (1 — POST обнаружила несоответствие размера памяти с записанным в CMOS), бит 3 — сбой контроллера первого жесткого диска (1 — есть сбой), бит 2 — сбой в работе часов RTC (1 — есть сбой), биты 0 и 1 зарезервированы и должны быть равны 0
0Fh	Состояние компьютера перед последней загрузкой: 00h — был выполнен сброс по питанию (кнопка Reset или комбинация клавиш <Ctrl>+<Alt>+<Del>), 03h — ошибка тестирования памяти, 04h — POST была завершена и система перезагружена, 05h — переход (jmp dword ptr) на адрес в 0040h:0067h, 07h — ошибка защиты при тестировании, 08h — ошибка размера памяти

Таблица 9.1 *(окончание)*

Адрес регистра	Описание
10h	Тип дисковода (0—3 для A и 4—7 для B): 0000b — нет, 0001b — 360 Кбайт, 0010b — 1,2 Мбайт, 0011b — 720 Кбайт, 0100b — 1,44 Мбайт, 0101b — 2,88 Мбайт
11h	Резерв
12h	Тип жесткого диска (0—3 для D и 4—7 для C): 0000b — нет, 0001b—1110b — тип дисковода (от 1 до 14), 1111b — диск первый описывается в регистре 19h, а диск второй — в 1Ah
13h	Резерв
14h	Состояние оборудования: биты 6—7 — число флоппи-дисководов (00b — один, 01b — два), биты 4—5 — тип дисплея (00b — EGA или VGA, 01b — цветной 40×25, 10b — цветной 80×25, 11b — монохромный 80×25), биты 2 и 3 зарезервированы, бит 1 — наличие сопроцессора (1 — есть), бит 0 — наличие флоппи-дисковода (1 — есть)
15h	Младший байт размера основной памяти в килобайтах (80h)
16h	Старший байт размера основной памяти в килобайтах (02h)
17h	Младший байт размера дополнительной памяти в килобайтах
18h	Старший байт размера дополнительной памяти в килобайтах
19h	Тип первого жесткого диска
1Ah	Тип второго жесткого диска
1Bh—2Dh	Резерв
2Eh	Старший байт контрольной суммы регистров CMOS (10h—2Dh)
2Fh	Младший байт контрольной суммы регистров CMOS (10h—2Dh)
30h	Младший байт размера дополнительной памяти в килобайтах
31h	Старший байт размера дополнительной памяти в килобайтах
32h	Значение века в формате BCD
33h	Дополнительный флаг свойств: бит 7 — размер памяти (1 — больше 1 Мбайт, 0 — до 1 Мбайт), бит 6 — используется программой установки, биты 0—5 зарезервированы
34h—3Fh (7Fh)	Зависят от производителя

Используемый формат представления данных *BCD* (Binary Coded Decimal) представляет собой двоично-десятичный код, где каждый байт содержит два независимых значения. Первое из них кодируется битами 7—4, а второе — битами 3—0. Поддерживаются только положительные значения до 99 включительно.

Пример получения числа установленных флоппи-дисководов показан в листинге 9.1.

Листинг 9.1. Определение числа установленных флоппи-дисководов

```
// напишем функцию для определения числа флоппи-дисководов
int GetNumFDD ( )
{
   DWORD dwResult = 0;
   // записываем значение регистра CMOS 14h в порт
   outPort ( 0x70, 0x14, 1);
   // делаем небольшую паузу
   Sleep ( 1 );
   // читаем значение из порта 71h
   inPort ( 0x71, &dwResult, 1 );
   // проверяем наличие дисковода в системе
   if ( ( dwResult & 0x01 ) == 0x00 ) return 0;
   // выделяем значение в битах 6 и 7
   dwResult &= 0x0C;
   dwResult = dwResult >> 6; // выделяем результат
   // проверяем
   switch ( dwResult )
   {
    case 0:
     return 1; // один флоппи-дисковод
    case 1:
     return 2; // два флоппи-дисковода
   }
   return 0;
}
```

Таким же способом можно получить или записать любое значение в регистры CMOS. Давайте попробуем выполнить очистку памяти CMOS. Хотя эта операция и позволяет осуществить программный сброс памяти, пользоваться ею стоит очень осторожно. Желательно, чтобы на материнской плате были установлены две микросхемы CMOS. Причина в том, что на старых и некачественных платах затирание ячеек памяти может привести к полному отказу начальной загрузки компьютера. Это может быть связано и с изношенностью микросхемы, и с плохим качеством исполнения, а может зависеть от батарейки. На современных качественных платах таких проблем, как правило, нет, но возможность нарушения работы системы все равно существует. Исходя из этого, я приведу пример очистки CMOS-памяти, но ответственность за любые проблемы будет лежать целиком на вас. Итак, рассмотрим следующий пример, показанный в листинге 9.2.

Листинг 9.2. Очистка памяти CMOS

```
// напишем функцию для очистки CMOS
void Clear_CMOS ( unsigned int uSizeCMOS )
{
   if ( uSizeCMOS < 64 ) return;
   for ( int i = 0; i < uSizeCMOS; i++ )
   {
     outPort ( 0x70, i, 1 ); // записываем номер регистра в CMOS
     outPort ( 0x71, i, 1 ); // записываем значение в регистр
   }
}
// воспользуемся нашей функцией для очистки CMOS-памяти размером 64 байта
Clear_CMOS ( 64 );
```

И напоследок я хочу привести примеры функций для C++, позволяющие писать (листинг 9.3) и читать (листинг 9.4) память CMOS в файл.

Листинг 9.3. Сохранение памяти CMOS в файл

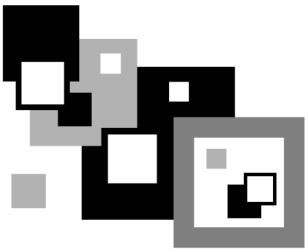
```
// пишем функцию сохранения CMOS в файл
 bool Save_CMOS ( char* FileName )
{
   FILE* outFile;
   DWORD dwResult = 0;
   BYTE buffer[64];
   // открываем новый файл
   if ( ( outFile = fopen ( FileName , "wb" ) ) == NULL )
      return false; // не удалось создать файл
   for ( int i = 0; i < 64; i++ )
   {
     outPort ( 0x70, i, 1 ); // записываем номер регистра в CMOS
     Sleep ( 1 ); // пауза
     inPort ( 0x71, &dwResult, 1 ); // читаем байт из CMOS
     buffer[i] = dwResult;
   }
   // записываем данные в файл
   fwrite ( buffer, 1, 64, outFile );
   // закрываем файл
   fclose( outFile );
   return true;
}
```

Листинг 9.4. Загрузка памяти CMOS из файла

```
// пишем функцию загрузки CMOS из файла
bool Load_CMOS ( char* FileName )
{
   FILE* inFile;
   DWORD dwResult = 0;
   BYTE buffer[64];
   // открываем новый файл
   if ( ( inFile = fopen ( FileName , "wb" ) ) == NULL )
      return false; // не удалось создать файл
   // читаем файл в буфер
   fread( buffer, 1, 64, inFile );
   // закрываем файл
   fclose( inFile );
   // сохраняем данные в CMOS
   for ( int i = 0; i < 64; i++ )
   {
     dwResult = buffer[i];
     outPort ( 0x70, i, 1 ); // записываем номер регистра в CMOS
     outPort ( 0x71, dwResult, 1 ); // записываем значение в регистр
   }
   return true;
}
```

На этом можно завершить программирование часов реального времени и CMOS-памяти.

ГЛАВА 10



Таймер

В любом компьютере наряду с часами реального времени существует устройство системного таймера, для реализации которого применяется специальная микросхема (например, 8254). Таймер помогает организовать всевозможные временные задержки, счетчики и управляющие сигналы. Из всех предоставляемых таймером функций можно выделить несколько основных:

1. Организация часов реального времени.
2. Программируемый генератор прямоугольных и синусоидальных импульсов.
3. Счетчик событий таймера.
4. Управление двигателями флоппи-дисководов.

Таймер предоставляет три независимых канала, каждый из которых имеет свое назначение. В первом канале (0) отслеживается текущее значение времени от момента включения компьютера, для хранения которого используется область памяти BIOS (`0040:006C`). Каждое изменение значения (18,2 раз в секунду) в этом канале генерирует прерывание `IRQ0` (`Int 8h`). Данное прерывание будет обработано процессором в первую очередь, но при этом должны быть разрешены аппаратные прерывания. При программировании первого канала следует всегда после выполнения задачи восстанавливать его первоначальное состояние. Второй канал (1) используется системой для работы с контроллером прямого доступа к памяти (DMA). Третий канал (2) позволяет управлять системным динамиком.

Генератор сигналов таймера вырабатывает импульсы с частотой 1 193 180 Гц. Поскольку максимальное значение 16-битного регистра ограничено значением 65 535, то используется делитель частоты. В итоге, результирующее значение равно 18,2 Гц. Именно с такой частотой выдается прерывание `IRQ0`.

Для работы с системным таймером используются порты от 40h до 43h. Все они имеют размер 8 бит. Порты с номерами 40h, 41h и 42h связаны соответственно с первым, вторым и третьим каналами, а порт 43h работает с управляющим регистром таймера. Принцип работы с портами очень простой: в порт 43h записывается управляющая команда, а после данные записываются или считываются из 40h, 41h и 42h, в зависимости от решаемой задачи. Формат управляющего командного регистра представлен в табл. 10.1.

Таблица 10.1. *Формат управляющего регистра таймера*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	Номер канала		Тип операции		Режим			Формат

Описание таблицы.

- Бит 0 определяет формат представления данных: 0 — двоичный (16-битное значение от 0000h до FFFFh), 1 — двоично-десятичный (BCD от 0000 до 9999).
- Биты 1—3 определяют режим работы таймера. Существует шесть режимов работы (от 0 до 5). Возможные значения перечислены в табл. 10.2.
- Биты 4—5 определяют тип операции. Имеется четыре возможных значения, которые перечислены в табл. 10.3.
- Биты 6—7 позволяют выбрать номер канала или управляющий регистр (только для операций чтения). Возможные значения этого поля представлены в табл. 10.4.

При установке битов 6 и 7 управляющего регистра в 1 будет использована команда считывания данных. При этом меняется формат определения этого регистра, согласно табл. 10.5.

Таблица 10.2. *Режимы работы таймера*

Бит 3	**Бит 2**	**Бит 1**	**Описание режима**
0	0	0	Генерация прерывания IRQ0 при установке счетчика в 0
0	0	1	Установка в режим ждущего мультивибратора
0	1	0	Установка в режим генератора импульсов
0	1	1	Установка в режим генератора прямоугольных импульсов
1	0	0	Установка в режим программно зависимого одновибратора
1	0	1	Установка в режим аппаратно-зависимого одновибратора

Таблица 10.3. *Тип операции*

Бит 5	Бит 4	Тип операции
0	0	Команда блокировки счетчика
0	1	Чтение/запись только младшего байта
1	0	Чтение/запись только старшего байта
1	1	Чтение/запись младшего, а за ним старшего байта

Таблица 10.4. *Возможные значения для поля 6—7*

Бит 7	Бит 6	Описание
0	0	Выбор первого канала (0)
0	1	Выбор второго канала (1)
1	0	Выбор третьего канала (2)
1	1	Команда считывания значений из регистров каналов

Таблица 10.5. *Формат управляющего регистра в режиме считывания*

Биты	7	6	5	4	3	2	1	0
Описание	1	1	Б	Статус	Канал 2	Канал 1	Канал 0	0

Приведем краткий комментарий к таблице.

- Бит 0 не используется и должен быть установлен в 0.
- Биты 1—3 позволяют установить номера каналов. Установка бита в 1 выбирает соответствующий номер канала.
- Бит 4 позволяет получить состояние для выбранного канала (каналов) при установке в 0.
- Бит 5 позволяет зафиксировать значение счетчика для выбранного канала (каналов) при установке в 0.
- Биты 6 и 7 определяют команду чтения и должны быть установлены в 1.

При выполнении команды блокировки счетчика (биты 4 и 5 установлены в 0), можно получить значения (состояние) выбранного счетчика без остановки самого таймера. Результат считывается из указанного канала (биты 6 и 7). При этом формат команды будет таким, как показано в табл. 10.6.

Таблица 10.6. *Формат команды блокировки*

Биты	7	6	5	4	3	2	1	0
Описание	Номер канала		0	0	Не используются			

Приведем краткое описание таблицы.

- Биты 0—3 не используются и должны игнорироваться.
- Биты 4 и 5 определяют команду блокировки и должны быть установлены в 0.
- Биты 6 и 7 определяют номер канала (табл. 10.4), для которого будет выполнена команда блокировки.

Полученный байт состояния имеет определенный формат (табл. 10.7).

Таблица 10.7. *Формат байта состояния канала*

Биты	7	6	5	4	3	2	1	0
Описание	OUT	Готов	Тип операции		Режим работы			Формат

Приведем краткое пояснение к таблице.

- Бит 0 определяет формат представления данных: 0 — двоичный (16-битное значение от `0000h` до `FFFFh`), 1 — двоично-десятичный (BCD от `0000` до `9999`).
- Биты 1—3 определяют режим работы таймера. Возможные значения перечислены в табл. 10.2.
- Биты 4—5 определяют тип операции. Возможные значения перечислены в табл. 10.3.
- Бит 6 определяет готовность счетчика для считывания данных (1 — готов, 0 — счетчик обнулен).
- Бит 7 определяет состояние выходного сигнала на канале в момент блокировки счетчика импульсов.

Рассмотрим стандартный пример для установки начального значения счетчика для второго канала (управляет динамиком) равным 5120 Гц.

Листинг 10.1. Установка начального значения счетчика для второго канала

```
// пишем функцию установки значения счетчика
void SetCount ( int iDivider )
```

```
{
   int iValue = 0;
   outPort ( 0x43, 0xB6, 1 ); // канал 2, операция 4, режим 3, формат 0
   iValue = iDivider & 0x0F;
   outPort ( 0x42, iValue , 1 ); // младший байт делителя
   outPort ( 0x42, ( iDivider >> 4 ) , 1 ); // старший байт делителя
}
```

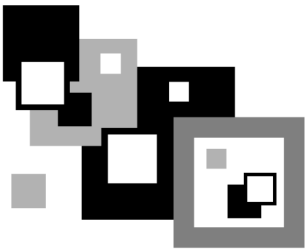
А теперь рассмотрим пример для получения случайного значения в установленном диапазоне.

Листинг 10.2. Получение случайного значения

```
// пишем функцию установки таймера
void InitCount ( int iMaximum )
{
   int iValue = 0;
   int iDiv = 1193180 / iMaximum;
   outPort ( 0x43, 0xB7, 1 ); // канал 2, операция 4, режим 3, формат 1
   iValue = iDiv & 0x0F;
   outPort ( 0x42, iValue , 1 ); // младший байт делителя
   outPort ( 0x42, ( iDiv >> 4 ) , 1 ); // старший байт делителя
}
// пишем функцию генерации случайного значения
int GetRandomValue ( int iMaximum )
{
   DWORD dwLSB = 0, dwMSB = 0;
   // канал 2, получить значение, биты 3-0 игнорируем
   outPort ( 0x43, 0x80, 1 );
   // читаем младший байт значения счетчика
   inPort ( 0x42, &dwLSB, 1 );
   // устанавливаем таймер заново
   InitCount ( iMaximum );
   // читаем старший байт значения счетчика
   inPort ( 0x42, &dwMSB, 1 );
   return ( int ) ( ( WORD ) ( ( ( BYTE ) ( dwLSB ) ) |
          ( ( ( WORD ) ( ( BYTE ) ( dwMSB ) ) ) << 8 ) ) );
}
```

На этом можно считать тему программирования системного таймера завершенной.

ГЛАВА 11



Дисковая подсистема

Каждый день мы садимся за компьютер и выполняем различную работу: кто-то заполняет бухгалтерские бланки, кто-то набирает текст, а кто-то скачивает музыку из Интернета. Чтобы вы не делали, постоянно приходится так или иначе сохранять результаты своего труда. Для этого используются наиболее популярные устройства: жесткий диск, гибкий диск или компакт-диск. Поэтому любой серьезный программист должен уметь работать с ними, несмотря на постоянное упрощение языков программирования. В этой главе мы научимся управлять данными устройствами, как минимум, двумя способами:

1. С использованием портов ввода-вывода.
2. С помощью интерфейса Win32 API.

Каждый способ имеет свои плюсы и минусы, а также определенную необходимость применения в той или иной ситуации. Окончательный выбор будет зависеть только от вас. И еще одно важное замечание: неправильное использование функций и других средств доступа к дискам может привести к полной или частичной потере информации, а также к поломке самого устройства.

11.1. Использование портов

Работа с портами напрямую является не такой сложной задачей, как может показаться сразу. Однако для правильной работы требуется хорошо знать структуру регистров дисковых устройств, а также иметь представление о протоколах работы с различными типами команд и интерфейсов. Сначала мы познакомимся с программированием накопителей на гибких дисках (флоппи-дисководов или иначе *FDD* — Floppy Disk Drive).

11.1.1. Регистры флоппи-дисковода

Для работы с данным устройством используются порты 3F0h–3F7h. Каждый из этих портов взаимодействует с определенным регистром дисковода. Назначение каждого регистра показано в табл. 11.1. Некоторые из них выполняют двойную роль.

Таблица 11.1. *Список портов ввода-вывода для флоппи-дисковода*

Регистр	Режим работы	Описание
3F0h	Чтение	Зарезервирован
3F1h	Чтение и запись	Дополнительный регистр состояния (SRB)
3F2h	Чтение и запись	Регистр цифрового вывода (DOR)
3F3h	Чтение и запись	Регистр управления (TDR)
3F4h	Чтение	Основной регистр состояния (MSR)
3F4h	Запись	Регистр управления скоростью передачи данных (DSR)
3F5h	Чтение и запись	Регистр данных (FIFO)
3F6h	—	Зарезервирован
3F7h	Чтение	Регистр цифрового ввода (DIR)
3F7h	Запись	Регистр управления конфигурацией (CCR)

Рассмотрим каждый из регистров подробнее.

11.1.1.1. Дополнительный регистр состояния

Определяет текущее состояние устройства. Может применяться как для чтения, так и для записи. Использует порт под номером 3F1h. Формат дополнительного регистра состояния показан в табл. 11.2.1—11.2.3.

Таблица 11.2.1. *Формат дополнительного регистра состояния*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	Резерв					УП	Резерв	СП

Таблица 11.2.2. *Формат дополнительного регистра состояния в режиме "Чтение"*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	Резерв					0	Резерв	Простой

Таблица 11.2.3. *Формат дополнительного регистра состояния в режиме "Запись"*

Биты	7	6	5	4	3	2	1	0
Описание	Резерв					Простой	Резерв	Резерв

Описание таблиц.

- Бит 0 определяет текущее состояние простоя (СП) устройства и доступен в режиме чтения регистра. В режиме записи он зарезервирован и не используется.
- Бит 1 зарезервирован.
- Бит 2 служит для управления режимом простоя (УП). В режиме чтения значение этого бита по умолчанию равно 0. В режиме записи, если бит установлен в 1, блокируется возможность выключения питания через регистр DSR. Восстановить нормальный режим можно только перезагрузкой системы.

Для использования этого регистра необходимо, чтобы он был включен управляющей командой (бит EREG EN установлен в 1). Пример считывания данного регистра приведен в листинге 11.1.

Листинг 11.1. Считывание информации из регистра состояния

```
// пишем функцию для проверки состояния питания
bool InPowerDown ( )
{
   DWORD dwResult  = 0; // переменная для хранения результата
   // читаем состояние порта
   inPort ( 0x3F1, &dwResult, 1 );
   if ( ( dwResult & 0x01 ) == 0x01 ) return true;
   return false;
}
```

11.1.1.2. Регистр цифрового вывода

Регистр позволяет управлять шаговыми двигателями флоппи-дисковода. Использует порт под номером `3F2h`. Может применяться как для чтения, так и для записи. Формат регистра показан в табл. 11.3.

Таблица 11.3. *Формат регистра цифрового вывода*

Биты	7	6	5	4	3	2	1	0
Описание	Резерв	Резерв	ШД2	ШД1	DMA	Сброс	Резерв	Диск

Описание таблицы.

- Бит 0 позволяет выбрать номер устройства. Возможны следующие значения: 0 — первый дисковод (1Ch) или 1 — второй дисковод (2Dh).
- Бит 1 зарезервирован и должен быть установлен в 0.
- Бит 2, установленный в 0, позволяет выполнить сброс устройства. После того как выполнена операция сброса, данный бит устанавливается в 1.
- Бит 3, установленный в 0, блокирует прерывания и доступ устройства к DMA, иначе доступ разрешен.
- Бит 4 управляет первым шаговым двигателем (ШД1) дисковода (1 — двигатель запущен, 0 — двигатель выключен).
- Бит 5 управляет вторым шаговым двигателем (ШД2) дисковода (1 — двигатель запущен, 0 — двигатель выключен).
- Биты 6 и 7 зарезервированы и должны быть установлены в 0.

Перед началом работы с дисководом следует включить двигатель.

Листинг 11.2. Управление двигателем флоппи-дисковода

```
// пишем функцию для инициализации дисковода и управления двигателем
void InitFDD ( bool Drive )
{
   if ( Drive ) // выполнить инициализацию и включить первый двигатель
   {
     outPort ( 0x3F2, 0x1C, 1 );
     Sleep ( 500 ); // ожидаем 0,5 секунды, пока двигатель раскрутится
   }
   else // выключить первый двигатель
   {
     outPort ( 0x3F2, 0x0C, 1 );
   }
}
```

11.1.1.3. Регистр управления

Регистр позволяет выбирать определенный дисковод, после чего он будет доступен для чтения и записи. По умолчанию инициализируется первый дисковод. Восстановить значения этого регистра можно только с помощью аппаратного сброса. Использует порт под номером 3F3h. Может применяться как для чтения, так и для записи. Формат регистра показан в табл. 11.4.1 и 11.4.2 Описание данного регистра рассматривается для контроллера фирмы Intel.

Таблица 11.4.1. *Формат регистра управления в режиме "Чтение"*

Биты	7	6	5	4	3	2	1	0
Описание	Резерв					Авто	Drive 1	Drive 0

Таблица 11.4.2. *Формат регистра управления в режиме "Запись"*

Биты	7	6	5	4	3	2	1	0
Описание	0	0	0	0	0	Авто	Drive 1	Drive 0

Описание таблиц.

- Бит 0, установленный в 1 (режим записи), позволяет блокировать первый дисковод (Drive 0). По умолчанию первый дисковод является загрузочным. При этом в бит 1 следует записать значение 0. В режиме чтения можно определить, активен ли первый дисковод (значение 1) или отключен (значение 0).
- Бит 1, установленный в 1 (режим записи), позволяет блокировать второй дисковод (Drive 1). При этом бит 0 следует установить в 0. В режиме чтения можно определить, активен ли второй дисковод (значение 1) или отключен (значение 0).
- Бит 2 управляет выбором загрузочного дисковода. По умолчанию бит равен 0. Это значит, что выбран первый дисковод. Значение бита, установленное в 1, означает, что выбран второй дисковод.
- Биты 3—7 в режиме чтения зарезервированы, а в режиме записи в них следует записать 0.

Пользоваться этим регистром не рекомендуется из-за различий использования его производителями (Intel, Via).

11.1.1.4. Основной регистр состояния

Регистр применяется для получения информации о завершении различных управляющих команд. Использует порт под номером `3F4h`. Может применяться только для чтения. Формат регистра показан в табл. 11.5.

Таблица 11.5. *Формат основного регистра состояния*

Биты	7	6	5	4	3	2	1	0
Описание	Данные	НП	Не DMA	Ком.	Резерв	Резерв	Диск 1	Диск 0

Описание таблицы.

- Бит 0 установлен в 1, когда первый дисковод (Drive 0) ищет данные команды или находится в режиме калибровки.
- Бит 1 установлен в 1, когда второй дисковод (Drive 1) ищет данные команды или находится в режиме калибровки.
- Биты 2 и 3 зарезервированы и равны 0.
- Бит 4 установлен в 1, когда происходит выполнение команды.
- Бит 5 установлен в 1, когда происходит передача данных без использования режима DMA.
- Бит 6 определяет направление передачи (НП) данных. Если бит равен 0, то от хоста (компьютера) к флоппи-дисководу, а когда бит установлен в 1 от флоппи-дисковода к хосту.
- Бит 7 информирует о готовности дисковода к приему данных. Если бит равен 1, можно передавать данные, если же бит установлен в 0, то дисковод не готов к приему данных и следует подождать.

Например, если регистр содержит значение `90h`, значит, выполняется команда и требуется подождать, а если в регистре записано значение `80h`, значит, контроллер готов принять очередную команду.

11.1.1.5. Регистр управления скоростью передачи данных

Регистр позволяет настроить параметры для скорости передачи данных. Управление скоростью передачи необходимо для совместимости с более старыми устройствами. Использует порт под номером `3F4h`. Может применяться только для записи. Формат регистра показан в табл. 11.6.

Таблица 11.6. *Формат регистра управления скоростью передачи*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	Сброс	П	ЗКГ	Предкомпенсация			Скорость	

Описание таблицы.

- Биты 0—1 позволяют установить одну из четырех возможных скоростей передачи данных. Список поддерживаемых значений показан в табл. 11.7. По умолчанию используется скорость 250 Кбит/с.
- Биты 2—4 определяют время задержки предкомпенсации. Это необходимо, поскольку магнитные носители имеют физически обоснованные задержки при намагничивании. Для компенсации магнитных явлений используется задержка по времени, измеряемая в наносекундах. Возможные

значения для этого поля представлены в табл. 11.8. По умолчанию используются следующие задержки: для скорости передачи 1 Мбит/с — 41,67 нс, а для скоростей 250—500 Кбит/с — 125 нс.

Таблица 11.7. *Список скоростей*

Значение битов 0 и 1	Скорость
00b	500 Кбит/с
01b	300 Кбит/с
10b	250 Кбит/с
11b	1 Мбит/с

Таблица 11.8. *Значения времени задержки*

Код задержки	Время, нс
000b	По умолчанию
001b	41,67
010b	83,34
011b	125,00
100b	166,67
101b	208,33
110b	250,00
111b	0,00 (не используется)

- Бит 5 управляет питанием задающего кварцевого генератора (ЗКГ). Установка бита в 1 позволит отключить подачу питания на генератор. Изменение этого бита следует выполнять только в режиме пониженного энергопотребления.
- Бит 6 управляет подачей питания к дисководу. При установке бита в 1 флоппи-дисковод будет переведен в режим пониженного потребления энергии. Аппаратный или программный сброс восстанавливают полный уровень подачи питания.
- Бит 7, установленный в 1, позволяет выполнить программный сброс контроллера, после чего он устанавливается в 0.

Рассмотрим пример кода для установки скорости передачи данных, показанный в листинге 11.3.

Листинг 11.3. Установка желаемой скорости

```
// раскручиваем двигатель и выполняем инициализацию
outPort ( 0x3F2, 0x1C, 1 );
// ожидаем 0,5 с, пока двигатель раскрутится
Sleep ( 500 );
// устанавливаем скорость 300 Кбит/с и время задержки 125,00 нс
outPort ( 0x3F4, 0x0D, 1 );
```

11.1.1.6. Регистр данных

Регистр предназначен для обмена данными между флоппи-дисководом и хостом (компьютером). Использует порт под номером 3F5h. Может применяться как для чтения, так и для записи. Размер регистра данных равен 16 байтам.

11.1.1.7. Регистр цифрового ввода

Регистр использует порт под номером 3F7h. Может применяться только для чтения. Задействован только старший бит (7), а остальные не используются. Если происходит смена диска, бит будет установлен в 1.

11.1.1.8. Регистр управления конфигурацией

Регистр позволяет установить скорость передачи данных (см. табл. 11.7). При этом задействуются только биты 0 и 1. Остальные должны быть равны 0. Регистр использует порт под номером 3F7h. Может применяться только для записи.

11.1.2. Команды управления для флоппи-дисковода

Для управления флоппи-дисководом используется специальный набор команд. Весь процесс выполнения команды можно разделить на три основные фазы: посылка команды, выполнение и получение результата. В первой фазе через регистр данных передается сама команда (строгая последовательность определенных байтов). Далее наступает вторая фаза, когда контроллер выполняет команду. После завершения обработки команды наступает последняя фаза, в которой считывается результат выполнения команды. Список команд показан в табл. 11.9.

Каждая из представленных команд имеет собственные значения параметров. Для упрощения работы используются специальные сокращения, полный список которых представлен в табл. 11.10.

Таблица 11.9. *Список команд для флоппи-дисковода*

Имя команды	Описание
READ DATA	Чтение данных
READ DELETED DATA	Чтение удаленных данных
WRITE DATA	Запись данных
WRITE DELETED DATA	Запись удаленных данных
READ TRACK	Чтение дорожки
VERIFY	Проверка данных
FORMAT TRACK	Форматирование дорожки
RECALIBRATE	Рекалибровка устройства
SENSE INTERRUPT STATUS	Получить состояние прерывания
SENSE DRIVE STATUS	Получить состояние флоппи-дисковода
SEEK	Поиск
READ ID	Получить идентификатор

Таблица 11.10. *Список принятых сокращений для описания параметров команд*

Сокращение	Описание
AUTO PD	Автоматическое управление энергопотреблением (0 — отключено, 1 — включено)
C	Номер цилиндра (от 0 до 255)
D0, D1	Выбор дисковода от 0 до 3
D	Шаблон данных для форматирования каждого сектора
DIR	Управление движением головки (0 — от шпинделя, 1 — к шпинделю)
DS0, DS1	Выбор дисковода (00b — первый, 01b — второй)
DTL	Размер сектора (если N больше 0, сюда следует записать FFh, если N равен 0, сюда записывается количество байтов для чтения или записи)
DRATE [0:1]	Скорость передачи данных в регистре DSR
DRT0, DRT1	Выбор таблицы скорости передачи данных (регистры DSR и CCR)
DT0, DT1	Выбор типа плотности для диска
EC	При установке бита в 1 разрешается счетчик для команды проверки данных VERIFY
EFIFO	Управление работой FIFO (0 — разрешить, 1 — запретить)
EIS	Управление поиском (0 — не использовать поиск, 1 — выполнять поиск в фазе выполнения команды перед операцией чтения или записи)

Таблица 11.10 *(продолжение)*

Сокращение	Описание
EOT	Конец дорожки (финальный номер сектора для текущей дорожки)
EREG EN	Включение дополнительного регистра состояния (0 — использовать стандартный регистр, 1 — использовать регистры TDR и SRB)
GAP	Размер промежутка 2 для перпендикулярного режима
GPL	Размер промежутка 3 для расстояния между секторами
FD0, FD1	Выбор флоппи-дисковода (00b — первый, 01b — второй)
HDS	Номер головки (0 или 1)
HLT	Время готовности к работе головки
HUT	Время удержания головки в рабочем состоянии
Lock	Блокирует параметры EFIFO, PRETRK и FIFOTHR
MFM	Установка бита в 1 включает режим двойной плотности для записи данных
MT	Установка мультидорожечного режима работы (1 — включить)
N	Код размера сектора в байтах (00h — 128 байт, 01h — 256 байт, 02h — 512 байт, 03h — 1024 байта, 04h — 2048 байт, 05h — 4096 байт, 06h — 8192 байта, 07h — 16384 байта)
NCN	Номер очередного цилиндра для операции поиска
ND	Управление режимом DMA (0 — разрешен, 1 — запрещен)
NRP	Установка бита в 1 позволит отключить фазу результата выполнения команды
PCN	Номер текущего цилиндра, получаемый с помощью команды SENSE INTERRUPT STATUS
PC2, PC2, PC0	Значение предкомпенсации в регистре DSR
PDOSC	Если бит равен 1, тактовый генератор отключен
PTS	Выбор предкомпенсации (0 — регистр DSR настроен на определенное время задержки, 1 — задержка не используется)
POLL	Управление режимом внутреннего опроса (0 — включен, 1 — выключен)
PRETRK	Номер стартовой дорожки для предкомпенсации (00h—FFh)
R	Номер сектора
SC	Количество секторов
SK	Флаг пропуска (1 — сектора, помеченные как удаленные, будут пропущены; 0 — сектора будут прочитаны или записаны)

Таблица 11.10 *(окончание)*

Сокращение	Описание
SRT	Временной интервал движения головки в миллисекундах (от 0,5 до 8 мс с шагом 0,5 мс)
STO, ST1, ST2	Внутренние регистры статуса (определяют результат выполнения команды)

А теперь разберем сами команды для работы с флоппи-дисководом и контроллером.

11.1.2.1. Команда *READ DATA*

Команда предназначена для чтения данных. Размер команды равен 9 байтам (2 байта описания команды и 7 байтов с параметрами). В третьей фазе команды возвращается 7 байтов. Формат команды для первой и третьей фаз показан в табл. 11.11.

Таблица 11.11. *Команда* `READ DATA`

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	MT	MFM	SK	0	0	1	1	0
	0	0	0	0	0	HDS	DS1	DS0
	C							
	H							
	R							
	N							
	EOT							
	GPL							
	DTL							
Третья фаза	ST0							
	ST1							
	ST2							
	C							
	H							
	R							
	N							

Размер блока данных в байтах, которые будут считаны, определяется параметром N. При мультидорожечной операции (бит MT установлен) данные будут считываться одновременно с двух дорожек (табл. 11.12).

Таблица 11.12. *Размеры передаваемых данных*

Количество секторов на дорожке	Размер одного сектора, байт	MT	N	Размер данных, байт
26	256	0	1	6 656
52	256	1	1	13 312
15	512	0	2	7 680
30	512	1	2	15 360

После выполнения команды будут возвращены в том числе 3 байта статуса: ST0, ST1, ST2. По ним можно судить о результате выполнения операции. Форматы этих байтов показаны соответственно в табл. 11.13, 11.14 и 11.15.

Таблица 11.13. *Формат байта статуса ST0*

Биты	7	6	5	4	3	2	1	0
Описание	Код завершения (IC)		Поиск (SE)	Ошибка (EC)	Не готов (NR)	Номер головки (HD)	Номер диска (US)	

Описание табл. 11.13.

- Биты 0—1 содержат номер дисковода.
- Бит 2 определяет номер головки дисковода.
- Бит 3 будет установлен в 1, если дисковод не готов к работе.
- Бит 4 указывает на ошибку в работе дисковода, если установлен в 1.
- Бит 5 будет установлен в 1 после выполнения команды поиска.
- Биты 6—7 определяют код завершения команды: `00b` — успешное завершение, `01b` — команда завершилась ошибкой, `10b` — неправильные параметры или код команды, `11b` — команда не выполнена из-за отсутствия диска в дисководе.

Таблица 11.14. *Формат байта статуса ST1*

Биты	7	6	5	4	3	2	1	0
Описание	Сектор (EN)	0	Ошибка (DE)	Буфер (OR)	0	Нет данных (ND)	Защита (NW)	Маркер (MA)

Описание табл. 11.14.

- Бит 0 будет установлен в 1, если отсутствует адресная метка (маркер), задаваемая при форматировании диска.
- Бит 1 устанавливается в 1, если выполняется попытка записи на защищенный диск.
- Бит 2 будет установлен в 1 при отсутствии данных в заданном секторе.
- Бит 4 будет установлен в 1, если переполнен буфер данных.
- Бит 5 будет установлен в 1, если возникла ошибка в передаваемых данных.
- Бит 7 будет установлен в 1, если выполнена попытка обращения к сектору, номер которого больше максимально возможного.

Таблица 11.15. *Формат байта статуса ST2*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	0	Метка (CM)	CRC (DD)	Ц (WC)	0	0	СЦ (BC)	Метка (MD)

Описание табл. 11.15.

- Бит 0 установлен в 1, если отсутствует адресная метка.
- Бит 1 установлен в 1, если обнаружен дефектный цилиндр.
- Бит 4 установлен в 1, если возникла ошибка с определением номера цилиндра.
- Бит 5 установлен в 1, если есть ошибка проверки по контрольной сумме CRC.
- Бит 6 установлен в 1, если попалась метка для удаленных данных.

Дополнительно существует еще один байт статуса (ST3), формат которого представлен в табл. 11.16.

Таблица 11.16. *Формат байта статуса ST3*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	Ошибка (FT)	Защита (WP)	Готов (RDY)	ТП (TO)	Число сторон (TS)	Номер головки (HD)	Номер диска (US)	

Описание табл. 11.16.

- Биты 1—0 определяют номер дисковода.
- Бит 2 указывает на номер головки.

- Бит 3 установлен в 1, если поддерживается работа с двухсторонними дисками.
- Бит 4 определяет текущую позицию головки, если она находится на нулевой дорожке.
- Бит 5 установлен в 1, если дисковод готов к работе.
- Бит 6 установлен в 1, если дисковод защищен от записи.
- Бит 7 установлен в 1, если произошла ошибка в устройстве.

В листинге 11.4 показан пример кода для считывания одного байта информации.

Листинг 11.4. Чтение одного байта

```
// пишем функцию для считывания одного байта
BYTE GetByteFDD ( )
{
   DWORD dwResult  = 0; // переменная для хранения результата
   int iTimeWait = 50000;
   // читаем порт статуса, пока биты 6 и 7 не будут установлены в 1
   while ( -- iTimeWait > 0 )
   {
       // читаем состояние порта
       inPort ( 0x3F4, &dwResult, 1 );
       if ( ( dwResult & 0xC0 ) == 0xC0 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return ERROR_TIME;
   }
   // читаем байт
   inPort ( 0x3F5, &dwResult, 1 );
   return (BYTE) dwResult;
}
```

Для чтения сектора флоппи-диска необходимо выполнить следующую последовательность операций:

1. Инициализировать устройство и включить двигатель.
2. Установить скорость передачи данных или использовать заданную по умолчанию.
3. Подождать не менее 0,5 сек, пока раскрутится двигатель.
4. Провести рекалибровку.
5. Инициализировать контроллер DMA.

6. Передать команду чтения.
7. Включить счетчик времени ожидания.
8. Проверить прерывание от контроллера гибких дисков. Если прерывания нет, увеличить счетчик времени. По истечении времени ожидания сгенерировать ошибку чтения данных и завершить программу.
9. Если прерывание получено, то прочитать блок данных (третья фаза). Если операция выполнена, то завершить программу.
10. Если операция не выполнена, сделать еще 3 попытки. Каждая попытка должна начинаться с 5-го пункта.
11. Если в течение 3 повторений не удалось прочитать данные, следует завершить программу с ошибкой чтения.

11.1.2.2. Команда *READ DELETED DATA*

Команда предназначена для чтения данных, в том числе и удаленных. Размер команды равен 9 байтам (2 байта описания команды и 7 байтов с параметрами). В третьей фазе команды возвращается 7 байтов. Аналогична команде `READ DATA` (см. табл. 11.11), за исключением первого байта (табл. 11.17).

Таблица 11.17. *Формат команды* `READ DELETED DATA` *(первый байт)*

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	MT	MFM	SK	0	1	1	0	0

Последовательность работы с данной командой ничем не отличается от обычной команды чтения.

11.1.2.3. Команда *WRITE DATA*

Команда позволяет записать данные на диск. Размер команды равен 9 байтам (2 байта описания команды и 7 байтов с параметрами). В третьей фазе команды возвращается 7 байтов. Формат команды совпадает с `READ DATA` (см. табл. 11.11), за исключением первого байта (табл. 11.18).

Таблица 11.18. *Формат команды* `WRITE DATA` *(первый байт)*

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	MT	MFM	0	0	0	1	0	1

Для записи байта в порт можно использовать код, показанный в листинге 11.5.

Листинг 11.5. Запись одного байта

```
// пишем функцию для записи одного байта
bool SetByteFDD ( BYTE Data )
{
   DWORD dwResult  = 0; // переменная для хранения результата
   int iTimeWait = 50000;
   // читаем порт статуса, пока бит 7 не будет установлен в 1
   while ( -- iTimeWait > 0 )
   {
       // читаем состояние порта
       inPort ( 0x3F4, &dwResult, 1 );
       if ( ( dwResult & 0x80 ) == 0x01 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return false;
   }
   // пишем байт
   outPort ( 0x3F5, Data, 1 );
   return true;
}
```

11.1.2.4. Команда *WRITE DELETED DATA*

Команда позволяет записать данные на диск. Каждый записываемый сектор помечается как удаленный. Размер команды равен 9 байтам (2 байта описания команды и 7 байтов с параметрами). В третьей фазе команды возвращается 7 байтов. Формат команды совпадает с `READ DATA` (см. табл. 11.11), за исключением первого байта (табл. 11.19).

Таблица 11.19. *Формат команды* `WRITE DELETED DATA` *(первый байт)*

Фаза	Биты							
	7	6	5	4	3	2	1	0
Первая фаза	MT	MFM	0	0	1	0	0	1

11.1.2.5. Команда *READ TRACK*

Команда позволяет прочитать данные для указанной дорожки.

Размер команды равен 9 байтам (2 байта описания команды и 7 байтов с параметрами). В третьей фазе команды возвращается 7 байтов. Формат коман-

ды совпадает с `READ DATA` (см. табл. 11.11), за исключением первого байта (табл. 11.20).

Таблица 11.20. *Формат команды* `READ TRACK` *(первый байт)*

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	0	MFM	0	0	0	0	1	0

11.1.2.6. Команда *VERIFY*

Команда позволяет проверить данные на диске. Размер команды равен 9 байтам (2 байта описания команды и 7 байтов с параметрами). В третьей фазе команды возвращается 7 байтов. Формат команды показан в табл. 11.21.

Таблица 11.21. *Команда* `VERIFY`

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	MT	MFM	SK	1	0	1	1	0
	EC	0	0	0	0	HDS	DS1	DS0
	C							
	H							
	R							
	N							
	EOT							
	GPL							
	DTL / SC							
Третья фаза	ST0							
	ST1							
	ST2							
	C							
	H							
	R							
	N							

11.1.2.7. Команда *FORMAT TRACK*

Команда позволяет отформатировать заданный носитель. Размер команды равен 6 байтам (2 байта описания команды и 4 байта с параметрами). Во второй фазе возвращается информация о форматировании текущего сектора. В третьей фазе команды возвращается 7 байтов. Формат команды показан в табл. 11.22.

Таблица 11.22. *Команда* `FORMAT TRACK`

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	0	MFM	0	0	1	1	0	1
	0	0	0	0	0	HDS	DS1	DS0
	N							
	SC							
	GPL							
	D							
Вторая фаза	C							
	H							
	R							
	N							
Третья фаза	ST0							
	ST1							
	ST2							
	Не определен							
	Не определен							
	Не определен							
	Не определен							

В табл. 11.23 приводятся стандартные значения для некоторых полей команды.

Таблица 11.23. *Стандартные значения для форматирования дисков*

Тип	Размер	Размер сектора, байт	N	SC	GPL (чтение и запись)	GPL (формат)
5,25"	360 Кбайт	512	02h	09h	2Ah	50h
	1,2 Мбайт	512	02h	0Fh	2Ah	50h

Таблица 11.23 *(окончание)*

Тип	Размер	Размер сектора, байт	N	SC	GPL (чтение и запись)	GPL (формат)
3,5"	720 Кбайт	512	02h	09h	1Bh	54h
	1,44 Мбайт	512	02h	18h	1Bh	54h
	2,88 Мбайт	512	02h	24h	38h	53h

11.1.2.8. Команда *RECALIBRATE*

Команда позволяет установить головки дисковода в нулевую позицию. Размер команды равен 2 байтам. Третья фаза отсутствует. Формат команды показан в табл. 11.24. После выполнения данной команды следует вызывать команду `SENSE INTERRUPT STATUS` для получения результатов выполнения.

Таблица 11.24. *Команда* `RECALIBRATE`

Фаза	Биты							
	7	6	5	4	3	2	1	0
Первая фаза	0	0	0	0	0	1	1	1
	0	0	0	0	0	0	DS1	DS0

В листинге 11.6 показан пример кода, выполняющий рекалибровку для первого дисковода (A).

Листинг 11.6. Выполнение рекалибровки флоппи-дисковода

```
// функция рекалибровки дисковода
void RecalibrateFDD ( unsigned int uDisk )
{
   SetByteFDD ( 0x07 ); // команда RECALIBRATE
   SetByteFDD ( (BYTE) uDisk ); // первый дисковод (A)
   GetWaitINT ( ); // ждем прерывания
}
```

11.1.2.9. Команда *SENSE INTERRUPT STATUS*

Команда позволяет получить информацию о состоянии прерывания для флоппи-дисковода. Размер команды равен 1 байту. В третьей фазе возвращается 2 байта. Формат команды показан в табл. 11.25.

Таблица 11.25. *Формат команды* `SENSE INTERRUPT STATUS`

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	0	0	0	0	1	0	0	0
Третья фаза	ST0							
	PCN							

Контроллер дисковода вырабатывает сигнал прерывания в следующих случаях:

1. После выполнения команды `READ DATA` (третья фаза).
2. После выполнения команды `READ DELETED DATA` (третья фаза).
3. После выполнения команды `READ TRACK` (третья фаза).
4. После выполнения команды `READ ID` (третья фаза).
5. После выполнения команды `WRITE DATA` (третья фаза).
6. После выполнения команды `WRITE DELETED DATA` (третья фаза).
7. После выполнения команды `FORMAT TRACK` (третья фаза).
8. После выполнения команды `SEEK`.
9. После выполнения команды `RECALIBRATE`.
10. При передаче данных не в DMA-режиме.

В результате выполнения команды `SENSE INTERRUPT STATUS`, будет возвращен байт состояния ST0 (см. табл. 11.13). Причина прерывания и результат будут закодированы в битах IC (6 и 7) и SE (5). Если сигнал прерывания отсутствует, то команда запишет в ST0 значение `80h`. Для получения информации о прерывании рассмотрим пример, показанный в листинге 11.7.

Листинг 11.7. Получение информации о прерывании

```
// пишем функцию для получения статуса прерывания
void GetWaitINT ( )
{
   DWORD dwResult  = 0; // переменная для хранения результата
   int iTimeWait = 50000;
   // команда SENSE INTERRUPT STATUS
   SetByteFDD ( 0x08 );
   // читаем порт 3F5h
   while ( -- iTimeWait > 0 )
```

```
    {
       inPort ( 0x3F5, &dwResult, 1 );
       // если прерывания есть, то выходим из функции
       if ( dwResult & 0x80 ) == 0x80 ) break;
       // закончилось время ожидания
        if ( iTimeWait < 1 ) return ERROR_TIME;
    }
}
```

11.1.2.10. Команда *SENSE DRIVE STATUS*

Команда позволяет получить информацию о состоянии дисковода.

Размер команды равен 1 байту. В третьей фазе возвращается 1 байт. Формат команды показан в табл. 11.26.

Таблица 11.26. *Формат команды* `SENSE DRIVE STATUS`

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	0	0	0	0	0	1	0	0
Третья фаза	ST3							

11.1.2.11. Команда *SEEK*

Команда позволяет переместить головки дисковода в заданную позицию (цилиндр). Размер команды равен 3 байтам. После начала выполнения команды сравнивается заданное значение цилиндра NCN с текущим PCN. Если значения не совпадают, то контроллер производит пошаговый поиск с проверкой результата после каждого шага. Формат команды показан в табл. 11.27. После выполнения данной команды следует вызывать команду `SENSE INTERRUPT STATUS` для получения результатов выполнения.

Таблица 11.27. *Формат команды* `SENSE DRIVE STATUS`

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	0	0	0	0	1	1	1	1
Вторая фаза	—							

11.1.2.12. Команда *READ ID*

Команда позволяет получить текущую позицию головок дисковода. Размер команды равен 2 байтам. Формат команды показан в табл. 11.28.

Таблица 11.28. *Команда* `READ ID`

Фаза	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Первая фаза	0	MFM	0	0	1	0	1	0
	0	0	0	0	0	HDS	DS1	DS0
Третья фаза	ST0							
	ST1							
	ST2							
	C							
	H							
	R							
	N							

На этом завершим описание команд для флоппи-дисководов и перейдем к программированию устройств с *интерфейсом ATA* (AT Attachment Interface) и *ATAPI* (AT Attachment with Packet Interface Extension).

11.1.3. Устройства ATA/ATAPI

Интерфейс ATAPI отличается от ATA способом передачи данных: запись и считывание информации происходят в пакетном режиме, похожим на работу устройств с *интерфейсом SCSI* (Small Computer Systems Interface). Интерфейс ATA позволяет управлять четырьмя дисководами, подключенными к контроллеру через два канала. На каждом канале одно из устройств является *ведущим (Master)* (Device 0), а одно *ведомым (Slave)* (Device 1). К устройствам ATA относятся в основном все современные жесткие диски, а к *устройствам ATAPI* — такие устройства, как CD-ROM, CD-RW, DVD-ROM, DVD-RW и т. п.

Для работы с устройствами ATA и ATAPI существуют жестко закрепленные порты ввода-вывода и номера прерываний (табл. 11.29).

Каждому каналу выделены собственные номера регистров. Назначение этих регистров описано в табл. 11.30.

Таблица 11.29. *Доступ к устройствам ATA и ATAPI*

Канал	Номер прерывания	Управляющие регистры	Дополнительные регистры
1	14	1F0—1F7 (8 байт)	3F6h (1 байт)
2	15	170—177 (8 байт)	376h (1 байт)

Таблица 11.30. *Назначение регистров*

Канал 1	Канал 2	Использование регистра	
		Чтение	Запись
1F0h	170h	Регистр данных (DR)	—
1F1h	171h	Регистр ошибки (ER)	Регистр особенностей (FT)
1F2h	172h	Регистр счетчика секторов (SC)	Регистр счетчика секторов (SC)
1F3h	173h	Регистр номера сектора (SN)	Регистр номера сектора (SN)
1F4h	174h	Регистр младшего байта номера цилиндра (CL)	Регистр младшего байта номера цилиндра (CL)
1F5h	175h	Регистр старшего байта номера цилиндра (CH)	Регистр старшего байта номера цилиндра (CH)
1F6h	176h	Регистр выбора устройства и номера головки (DH)	—
1F7h	177h	Регистр состояния (SR)	Регистр команд (CR)
3F6h	376h	Дополнительный регистр состояния (AC)	Регистр управления (DC)

Как видно из табл. 11.30, некоторые регистры имеют двойное назначение, в зависимости от выполняемой операции (запись или чтение). К сожалению, адресация *CHS* (Cylinder Head Sector) в последнее время не используется. Ее полностью заменил режим логической адресации *LBA* (Logical Block Addressing). При этой адресации для указания адреса сектора применяется 28-разрядный адрес LBA. Исходя из этого, изменилось и назначение регистров. В табл. 11.31 и 11.32 приводится новое назначение регистров для ATA- и ATAPI-устройств.

Таблица 11.31. *Назначение регистров для ATA*

Канал 1	Канал 2	Использование регистра	
		Чтение	Запись
1F0h	170h	Регистр данных (DR)	Регистр данных (DR)
1F1h	171h	Регистр ошибки (ER)	Регистр особенностей (FT)

Таблица 11.31 *(окончание)*

Канал 1	Канал 2	Использование регистра	
		Чтение	**Запись**
1F2h	172h	Регистр счетчика секторов (SC)	Регистр счетчика секторов (SC)
1F3h	173h	Первый байт адреса (0—7)	Первый байт адреса (0—7)
1F4h	174h	Второй байт адреса (8—15)	Второй байт адреса (8—15)
1F5h	175h	Третий байт адреса (16—23)	Третий байт адреса (16—23)
1F6h	176h	Регистр выбора устройства и 4 бита LBA адреса (24—27)	Регистр выбора устройства и 4 бита LBA адреса (24—27)
1F7h	177h	Регистр состояния (SR)	Регистр команд (CR)
3F6h	376h	Дополнительный регистр состояния (AC)	Регистр управления устройством (DC)

Таблица 11.32. *Назначение регистров для ATAPI*

Канал 1	Канал 2	Использование регистра	
		Чтение	**Запись**
1F0h	170h	Регистр данных (DR)	Регистр данных (DR)
1F1h	171h	Регистр ошибки (ER)	Регистр особенностей (FT)
1F2h	172h	Регистр прерывания	—
1F3h	173h	—	—
1F4h	174h	Младший байт пакета данных	Младший байт пакета данных
1F5h	175h	Старший байт пакета данных	Старший байт пакета данных
1F6h	176h	Выбор устройства	Выбор устройства
1F7h	177h	Регистр состояния (SR)	Регистр команд (CR)

Рассмотрим назначение каждого регистра подробнее.

11.1.3.1. Регистр данных

Регистр используется для передачи данных в операциях чтения и записи. Это единственный из всех регистров, имеющий размер 16 бит, однако дополнительные 8 бит он "уводит" у регистра ошибок, перекрывая последний. При этом младший байт расположен в регистре 1x0h, а старший — в 1x1h.

11.1.3.2. Регистр ошибки

Регистр доступен для считывания и позволяет получить результат выполненной операции. Формат регистра ошибки может меняться в зависимости от

выполненной команды. Не изменяется только значение бита 2 (ABRT). Если он установлен в единицу, значит, команда была прервана из-за ошибки. Проверку этого регистра следует проводить только тогда, когда бит 0 (ERR) в регистре состояния установлен в 1. Для устройств ATAPI формат регистра показан в табл. 11.33.

Таблица 11.33. *Формат регистра ошибки для ATAPI*

Биты	7	6	5	4	3	2	1	0
Описание	Sense Key				—	ABRT	EOM	ILI

Описание таблицы.

- Бит 0 установлен в 1, когда указан неправильный размер пакета команды или блока с данными.
- Бит 1 установлен в 1, когда найден конец носителя.
- Бит 2 установлен в 1, когда команда была прервана из-за ошибки.
- Бит 3 зарезервирован.
- Биты 4—7 определяют значение ключа смысла, описывающего ошибку.

11.1.3.3. Регистр особенностей

Регистр доступен только для записи. Используется для выполнения различных установок и зависит от используемой команды. Для устройств ATAPI формат регистра показан в табл. 11.34.

Таблица 11.34. *Формат регистра особенностей для ATAPI*

Биты	7	6	5	4	3	2	1	0
Описание	Резерв						OVL	DMA

Описание таблицы.

- Бит 0 управляет выбором режима передачи: 1 — DMA, 0 — *PIO* (Programmed Input/Output, программируемый ввод-вывод).
- Бит 1 управляет выбором режима перекрытия команд: 1 — включен, 0 — выключен.

11.1.3.4. Регистр счетчика секторов

Регистр определяет общее количество секторов, которое будет записано или считано. Доступен для записи и чтения. При передаче очередного сектора значение в этом регистре уменьшается на 1, что позволяет проконтролиро-

вать число реально обработанных секторов. Для этого достаточно прочитать значение из регистра.

11.1.3.5. Регистр прерывания

Регистр позволяет определить различную информацию о состоянии прерывания. Доступен только для чтения. Бит 0 (C/D) определяет тип данных (1 — управляющая команда, 0 — пользовательские данные). Бит 1 (I/O) указывает направление передачи данных (1 — от устройства к компьютеру, 0 — от компьютера к устройству). Бит 2 (REL) помогает определить, свободна ли шина (1 — шина свободна).

11.1.3.6. Регистр номера сектора

Регистр позволяет установить начальный сектор для операций записи или считывания. Доступен для записи и чтения. При использовании логической адресации содержит младший байт адреса (0—7).

11.1.3.7. Регистр младшего байта номера цилиндра

Регистр позволяет установить начальный номер цилиндра (младший байт). Доступен для записи и чтения. При использовании логической адресации содержит второй байт адреса (8—15).

11.1.3.8. Регистр старшего байта номера цилиндра

Регистр позволяет установить начальный номер цилиндра (старший байт). Доступен для записи и чтения. При использовании логической адресации содержит второй байт адреса (16—23).

11.1.3.9. Регистр выбора устройства и номера головки

Регистр позволяет выбрать номер устройства (0 или 1), а также дополнительные параметры в зависимости от режима адресации. Формат регистра показан в табл. 11.35.

Таблица 11.35. *Формат регистра выбора устройства*

Биты	7	6	5	4	3	2	1	0
Описание	1	LBA	1	DEV	x	x	x	x

Описание таблицы.

- Биты 0—3 при использовании CHS-адресации должны определять номер головки. В режиме LBA должны определять четыре старших разряда адреса (24—27): 0 — 24, 1 — 25, 2 — 26 и 3 — 27.

- Бит 4 позволяет выбрать номер устройства на канале: 1 — ведущее (Master), 0 — ведомое (Slave).
- Бит 5 устарел. Может быть установлен в 1 для совместимости со старыми стандартами.
- Бит 6 позволяет выбрать режим адресации: 0 — CHS, 1 — LBA.
- Бит 7 устарел. Может быть установлен в 1 для совместимости со старыми стандартами.

11.1.3.10. Регистр состояния

Позволяет получить текущее состояние устройства. Доступен только для чтения. Если бит 7 установлен в 1, то содержание регистра игнорируется. Формат регистра показан в табл. 11.36.

Таблица 11.36. *Формат регистра состояния*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	BSY	DRDY	DF	x	DRQ	—	—	ERR

Описание таблицы.

- Бит 0, установленный в 1, означает, что произошла ошибка при выполнении команды. Для устройств ATAPI этот бит называется CHK.
- Биты 1—2 устарели и не должны использоваться.
- Бит 3, установленный в 1, определяет, что устройство готово к передаче данных.
- Бит 4 может менять свое назначение в зависимости от выполняемой команды.
- Бит 5, установленный в 1, означает, что произошла неизвестная ошибка, например, аппаратный сбой устройства.
- Бит 6 равен 1, когда устройство может выполнить любую поддерживаемую команду.
- Бит 7, установленный в 1, означает, что устройство занято.

11.1.3.11. Регистр команд

Регистр позволяет передать устройству определенную команду. Доступен только для записи. Запись команды должна производиться, когда очищены биты BSY и DRQ в регистре состояния. Выполнение команды начинается сразу же после записи в этот регистр, поэтому если команда имеет дополни-

тельные параметры, то вначале следует записать их в соответствующие регистры.

11.1.3.12. Дополнительный регистр состояния

Регистр содержит ту же информацию, что и регистр состояния. Доступен только для чтения. Если бит 7 установлен в 1, содержание регистра игнорируется. Чтение регистра не сбрасывает сигнал прерывания.

11.1.3.13. Регистр управления

Регистр позволяет управлять программным сбросом устройств и сигналом прерывания. Доступен только для записи. Любые изменения в регистре сразу же вступают в силу. Формат регистра показан в табл. 11.37.

Таблица 11.37. *Формат регистра управления*

Биты	7	6	5	4	3	2	1	0
Описание	HOB	Резерв	Резерв	Резерв	Резерв	SRST	nIEN	0

Описание таблицы.

- Бит 0 должен быть установлен в 0.
- Бит 1 управляет сигналом прерывания: 1 — запрещен, 0 — разрешен.
- Бит 2 управляет программным сбросом устройств: 1 — выполнить сброс.
- Биты 3—6 зарезервированы.
- Бит 7 является старшим разрядом для 48-разрядного адреса набора особенностей. Запись любого значения в регистр команд очищает этот бит.

Теперь, когда мы разобрались с имеющимися регистрами, пора познакомиться с управляющими командами. Все команды можно разделить на две группы: команды обычные и пакетные. К последней группе также можно отнести набор команд SCSI (Multimedia Commands). Здесь мы рассмотрим только команды ATA/ATAPI.

11.1.4. Команды управления для ATA/ATAPI-устройств

Обычные команды могут быть трех видов: обязательные для всех устройств, необязательные и определяемые производителем. Список команд для интерфейса ATA/ATAPI-7 представлен в табл. 11.38.

Таблица 11.38. *Список команд интерфейса ATA/ATAPI-7*

Имя команды	Код	Поддержка
CFA ERASE SECTORS	C0h	Необязательная
CFA REQUEST EXTENDED ERROR	03h	Необязательная
CFA TRANSLATE SECTOR	87h	Необязательная
CFA WRITE MULTIPLE WITHOUT ERASE	CDh	Необязательная
CFA WRITE SECTORS WITHOUT ERASE	38h	Необязательная
CHECK MEDIA CARD TYPE	D1h	Необязательная
CHECK POWER MODE	E5h	Обязательная
CONFIGURE STREAM	51h	Необязательная
DEVICE CONFIGURATION FREEZE LOCK	B1h	Необязательная
DEVICE CONFIGURATION IDENTIFY	B1h	Необязательная
DEVICE CONFIGURATION RESTORE	B1h	Необязательная
DEVICE CONFIGURATION SET	B1h	Необязательная
DEVICE RESET	08h	Обязательная для ATAPI
DOWNLOAD MICROCODE	92h	Необязательная
EXECUTE DEVICE DIAGNOSTIC	90h	Обязательная
FLUSH CACHE	E7h	Обязательная для ATA
FLUSH CACHE EXT	EAh	Необязательная
GET MEDIA STATUS	DAh	Необязательная
IDENTIFY DEVICE	ECh	Обязательная для ATA
IDENTIFY PACKET DEVICE	A1h	Обязательная для ATAPI
IDLE	E3h	Обязательная для ATA
IDLE IMMEDIATE	E1h	Обязательная
MEDIA EJECT	EDh	Необязательная
MEDIA LOCK	DEh	Необязательная
MEDIA UNLOCK	DFh	Необязательная
NOP	00h	Обязательная для ATAPI
PACKET	A0h	Обязательная для ATAPI
READ BUFFER	E4h	Необязательная
READ DMA	C8h	Обязательная для ATA
READ DMA EXT	25h	Необязательная
READ DMA QUEUED	C7h	Необязательная

Таблица 11.38 *(продолжение)*

Имя команды	Код	Поддержка
READ DMA QUEUED EXT	26h	Необязательная
READ LOG EXT	2Fh	Необязательная
READ MULTIPLE	C4h	Обязательная для ATA
READ MULTIPLE EXT	29h	Необязательная
READ NATIVE MAX ADDRESS	F8h	Необязательная
READ NATIVE MAX ADDRESS EXT	27h	Необязательная
READ SECTOR(S)	20h	Обязательная
READ SECTOR(S) EXT	24h	Необязательная
READ STREAM DMA	2Ah	Необязательная
READ STREAM PIO	2Bh	Необязательная
READ VERIFY SECTOR(S)	40h	Обязательная для ATA
READ VERIFY SECTOR(S) EXT	42h	Необязательная
SECURITY DISABLE PASSWORD	F6h	Необязательная
SECURITY ERASE PREPARE	F3h	Необязательная
SECURITY ERASE UNIT	F4h	Необязательная
SECURITY FREEZE LOCK	F5h	Необязательная
SECURITY SET PASSWORD	F1h	Необязательная
SECURITY UNLOCK	F2h	Необязательная
SERVICE	A2h	Необязательная
SET FEATURES	EFh	Обязательная
SET MAX ADDRESS	F9h	Необязательная
SET MAX ADDRESS EXT	37h	Необязательная
SET MULTIPLE MODE	C6h	Обязательная для ATA
SLEEP	E6h	Обязательная
SMART DISABLE OPERATIONS	B0h	Необязательная
SMART ENABLE/DISABLE AUTOSAVE	B0h	Необязательная
SMART ENABLE OPERATIONS	B0h	Необязательная
SMART EXECUTE OFF_LINE IMMEDIATE	B0h	Необязательная
SMART READ DATA	B0h	Необязательная
SMART READ LOG	B0h	Необязательная
SMART RETURN STATUS	B0h	Необязательная

Таблица 11.38 *(окончание)*

Имя команды	Код	Поддержка
SMART WRITE LOG	B0h	Необязательная
STANDBY	E2h	Обязательная для ATA
STANDBY IMMEDIATE	E0h	Обязательная
WRITE BUFFER	E8h	Необязательная
WRITE DMA	CAh	Обязательная для ATA
WRITE DMA EXT	35h	Необязательная
WRITE DMA FUA EXT	3Dh	Необязательная
WRITE DMA QUEUED	CCh	Необязательная
WRITE DMA QUEUED EXT	36h	Необязательная
WRITE DMA QUEUED FUA EXT	3Eh	Необязательная
WRITE LOG EXT	3Fh	Необязательная
WRITE MULTIPLE	C5h	Обязательная для ATA
WRITE MULTIPLE EXT	39h	Необязательная
WRITE MULTIPLE FUA EXT	CEh	Необязательная
WRITE SECTOR(S)	30h	Обязательная для ATA
WRITE SECTOR(S) EXT	34h	Необязательная
WRITE STREAM DMA	3Ah	Необязательная
WRITE STREAM PIO	3Bh	Необязательная

Примечание

Интерфейс ATA/ATAPI-7 версии 7 представляет собой дальнейшее расширение набора программных и аппаратных ресурсов, описывающих интерфейс (на базе шины ATA) обмена данными между процессором и различными устройствами хранения данных.

Мы не будем рассматривать все имеющиеся команды, а разберем только обязательные для всех устройств.

11.1.4.1. Команда *CHECK POWER MODE*

Команда предназначена для получения информации о питании устройства. Команда является обязательной для обычных и пакетных устройств. Формат команды показан в табл. 11.39.

Таблица 11.39. *Формат команды* `CHECK POWER MODE`

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Не используется							
1x3h	Не используется							
1x4h	Не используется							
1x5h	Не используется							
1x6h	1	Резерв	1	DEV	Резерв	Резерв	Резерв	Резерв
1x7h	E5h							

Для инициализации команды следует в регистр выбора устройства `1x6h` записать номер устройства в бит 4 (DEV) (0 или 1), а затем в регистр команд `1x7h` записать код команды `E5h`. При успешном выполнении регистры будут содержать следующую информацию:

- Регистр прерывания `1x2h` будет хранить результат команды: `00h` — дежурный режим (Standby), `80h` — режим простоя (Idle), `FFh` — режим простоя (Idle) или активный (Active) режим.
- Регистр выбора устройства `1x6h` будет хранить номер выбранного устройства.
- Регистр состояния `1x7h` будет иметь вид (см. табл. 11.36): BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

В случае ошибки регистры будут иметь следующий вид:

- В регистре ошибки `1x1h` (см. табл. 11.22) бит ABRT будет установлен в 1, если устройство не поддерживает команду или произошло аварийное завершение.
- Регистр выбора устройства `1x6h` будет хранить номер выбранного устройства.
- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 1 (при сбое устройства), DRQ = 0 и ERR = 1.

Например, чтобы проверить состояние питания для жесткого диска на первом канале, можно использовать код из листинга 11.8.

Листинг 11.8. Проверка состояния питания жесткого диска

```
// пишем функцию для ожидания готовности устройства
void IsReadyATA ( )
```

```
{
   DWORD dwResult  = 0; // переменная для хранения результата
   int iTimeWait = 50000;
   // читаем порт 1F7h
   while ( -- iTimeWait > 0 )
   {
      inPort ( 0x1F7, &dwResult, 1 );
      // если бит 7 равен 0, выходим
      if ( dwResult & 0x80 ) == 0x00 ) break;
      // закончилось время ожидания
       if ( iTimeWait < 1 ) return ERROR_TIME;
   }
}
// получаем состояние питания
int GetPowerMode ( )
{
   DWORD dwResult  = 0; // переменная для хранения результата
   IsReadyATA ( ); // ждем, пока порт освободится
   outPort ( 0x1F6, 0xA0, 1 ); // первое устройство на первом канале
   outPort ( 0x1F7, 0xE5, 1 ); // код команды CHECK POWER MODE
   IsReadyATA ( ); // ждем, пока порт освободится
   inPort ( 0x1F7, &dwResult, 1 ); // читаем регистр статуса
   if ( ( dwResult & 0x01 ) == 0x01 ) // произошла ошибка
      return -1;
   IsReadyATA ( ); // ждем, пока порт освободится
   inPort ( 0x1F2, &dwResult, 1 ); // выбираем регистр
                                   // с результатом операции
   switch ( dwResult )
   {
    case 0: // дежурный режим
      return 1;
    case 128: // режим простоя
      return 2;
    case 256: // активный или простой
      return 3;
   }
}
```

11.1.4.2. Команда *DEVICE RESET*

Команда позволяет выполнить сброс выбранного устройства на шине. Используется только для ATAPI-устройств. Формат команды показан в табл. 11.40.

Таблица 11.40. *Формат команды* `DEVICE RESET`

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Не используется							
1x3h	Не используется							
1x4h	Не используется							
1x5h	Не используется							
1x6h	1	Резерв	1	DEV	Резерв	Резерв	Резерв	Резерв
1x7h	08h							

Для инициализации команды следует в регистр выбора устройства `1x6h` записать номер устройства в бит 4 (DEV) (0 или 1), а затем в регистр команд `1x7h` записать код команды `08h`. При успешном выполнении регистры будут содержать следующую информацию:

- В регистр `1x1h` будет помещен диагностический код, согласно табл. 11.41.
- В регистры `1x2h`–`1x6h` будет записана сигнатура (табл. 11.42).
- В регистр `1x7h` будет записана информация в соответствии с протоколом команды.

Таблица 11.41. *Диагностические коды*

Код	Описание
Первое устройство (Device 0)	
01h	Device 0 выполнило команду, Device 1 выполнило команду или отсутствует
00h, 02h–7Fh	Сбой в устройстве Device 0, Device 1 выполнило команду или отсутствует
81h	Device 0 выполнило команду, сбой в устройстве Device 1
80h, 82h–FFh	Сбой в обоих устройствах
Второе устройство (Device 1)	
01h	Device 1 выполнило команду
00h, 02h–7Fh	Сбой в устройстве Device 1

Таблица 11.42. *Значения для сигнатуры*

Регистры	ATA	ATAPI
1x2h	01h	01h
1x3h	01h	01h

Таблица 11.42 *(окончание)*

Регистры	ATA	ATAPI
1x4h	00h	14h
1x5h	00h	EBh
1x6h	00h	00h (Device 0) или 10h (Device 1)

11.1.4.3. Команда *EXECUTE DEVICE DIAGNOSTIC*

Команда позволяет выполнить диагностику устройства для определения его работоспособности. Будет выполнена для всех подключенных устройств. Формат команды показан в табл. 11.43.

Таблица 11.43. *Формат команды* EXECUTE DEVICE DIAGNOSTIC

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Не используется							
1x3h	Не используется							
1x4h	Не используется							
1x5h	Не используется							
1x6h	Не используется							
1x7h	90h							

Для инициализации команды следует в регистр команд 1x7h записать код команды 90h. При успешном выполнении регистры будут содержать следующую информацию:

- В регистр 1x1h будет помещен диагностический код, согласно табл. 11.41.
- В регистры 1x2h–1x6h будет записана сигнатура (см. табл. 11.42).
- В регистр 1x7h будет записана информация в соответствии с протоколом команды.

11.1.4.4. Команда *FLUSH CACHE*

Команда позволяет сохранить данные из кэша на диск. Является обязательной для всех устройств ATA. Для полного завершения операции команде может понадобиться больше полминуты. Формат команды показан в табл. 11.44.

Таблица 11.44. *Формат команды* `FLUSH CACHE`

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Не используется							
1x3h	Не используется							
1x4h	Не используется							
1x5h	Не используется							
1x6h	1	Резерв	1	DEV	Резерв	Резерв	Резерв	Резерв
1x7h	E7h							

Для инициализации команды следует в регистр выбора устройства `1x6h` записать номер устройства в бит 4 (DEV) (0 или 1), а затем в регистр команд `1x7h` записать код команды `E7h`. При успешном выполнении регистры будут содержать следующую информацию:

- Регистр `1x6h` будет хранить номер выбранного устройства.
- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

В случае ошибки регистры будут иметь следующий вид:

- В регистре ошибки `1x1h` (см. табл. 11.33) бит ABRT будет установлен в 1, если устройство не поддерживает команду или произошло аварийное завершение.
- Регистр `1x3h` будет хранить LBA-адрес сектора (0—7), на котором произошла ошибка.
- Регистр `1x4h` будет хранить LBA-адрес сектора (8—15), на котором произошла ошибка.
- Регистр `1x5h` будет хранить LBA-адрес сектора (16—23), на котором произошла ошибка.
- Регистр `1x6h` будет хранить номер устройства (бит 4) и LBA-адрес сектора в битах 0—3 (24—27).
- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 1 (при сбое устройства), DRQ = 0 и ERR = 1.

11.1.4.5. Команда *IDENTIFY DEVICE*

Команда позволяет получить различную информацию об устройстве. Является обязательной для всех устройств ATA. Формат команды показан в табл. 11.45.

Таблица 11.45. *Формат команды* `IDENTIFY DEVICE`

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Не используется							
1x3h	Не используется							
1x4h	Не используется							
1x5h	Не используется							
1x6h	1	Резерв	1	DEV	Резерв	Резерв	Резерв	Резерв
1x7h	ECh							

Для инициализации команды следует в регистр выбора устройства `1x6h` записать номер устройства в бит 4 (DEV) (0 или 1), а затем в регистр команд `1x7h` записать код команды `ECh`. При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройства `1x6h` будет хранить номер выбранного устройства.
- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

После выполнения команды устройство возвращает через регистр данных 256 слов (по 16 битов), описывающих параметры дисковода. Все зарезервированные биты и слова установлены в 0. Формат возвращаемых данных представлен в табл. 11.46.

Таблица 11.46. *Формат данных для команды* `IDENTIFY DEVICE`

Смещение слова	Описание слова
0	Конфигурация устройства: бит 15 — тип интерфейса (0 — ATA), 8—14 не используются, 7 — тип устройства (1 — со сменным носителем), 6—3 не используются, 2 — неполные данные (1 — получены не все возможные данные), 1 — не используются, 0 — зарезервирован
1	Устарело
2	Определяется производителем
3—6	Устарели
7—8	Зарезервированы
9	Не используется

Таблица 11.46 *(продолжение)*

Смещение слова	Описание слова
10—19	Серийный номер устройства (содержит 20 ASCII-символов)
20—21	Не используются
22	Устарело
23—26	Номер версии (содержит 8 ASCII-символов)
27—46	Номер модели (содержит 40 ASCII-символов)
47	Биты 8—15 установлены в 80h, а биты 0—7 определяют максимальное число для команд многоблочного чтения и записи (01h—FFh)
48	Зарезервировано
49	Поддерживаемые возможности: биты 14—15 — резерв, 13 — управление таймером для Standby (1 — поддержка в ATA есть, 0 — таймером должно управлять устройство), 12 — резерв, 11 — поддержка сигнала IORDY (0 — есть, 1 — нет), 10 — состояние сигнала IORDY (1 — выключен), 9 — поддержка LBA (1 — есть), 8 — поддержка DMA (1 — есть), 0—7 не используются
50	Поддерживаемые возможности: бит 15 равен 0, бит 14 равен 1, 2—13 — резерв, 1 — устарел, 0 — равен 1 для индикации минимального значения таймера Standby
51—52	Устарели
53	Состав: биты 3—15 — зарезервированы, 2 — допустимость слова 88 (1 — поле со смещением 88 допустимо), 1 — слов 64—70 (1 — допустимы), 0 — устарел
54—58	Устарели
59	Состав: биты 9—15 — резерв, 8 — настройка многоблочной передачи секторов (1 — допустима), 0—7 — текущее значение числа секторов для многоблочной передачи
60—61	Полное количество секторов, адресуемых пользователей
62	Устарело
63	Состав: биты 11—15 — резерв, 10 — режим Multiword DMA 2 (1 — выбран), 9 — режим Multiword DMA 1 (1 — выбран), 3—7 — резерв, 2 — поддержка режима Multiword DMA 2 (1 — есть), 1 — поддержка режима Multiword DMA 1 (1 — есть), 0 — поддержка режима Multiword DMA 0 (1 — есть)
64	Состав: 8—15 — резерв, 0—7 — поддержка режимов PIO
65	Минимальное время передачи для режима Multiword DMA в наносекундах
66	Рекомендуемое производителем время передачи для режима Multiword DMA в наносекундах
67	Минимальное время передачи для режима PIO в наносекундах
68	Минимальное время передачи для режима PIO с IORDY в наносекундах

Таблица 11.46 *(продолжение)*

Смещение слова	Описание слова
69—79	Необязательные и зарезервированные поля
80	Главный (major) номер версии интерфейса: биты 8—15 — резерв, 7 — поддержка версии ATA/ATAPI-7 (1 — есть), 6 — поддержка версии ATA/ATAPI-6 (1 — есть), 5 — поддержка версии ATA/ATAPI-5 (1 — есть), 4 — поддержка версии ATA/ATAPI-4 (1 — есть), 3 — поддержка версии ATA/ATAPI-3 (1 — есть), 1—2 устарели, 0 — резерв
81	Дополнительный (minor) номер версии интерфейса
82	Поддержка команд: бит 15 устарел, 14 — команда NOP (1 — есть), 13 — команда READ BUFFER (1 — есть), 12 — команда WRITE BUFFER (1 — есть), 11 устарел, 10 — защищенная область хоста (1 — есть), 9 — команда DEVICE RESET (1 — есть), 8 — прерывание SERVICE (1 — есть), 7 — прерывание при освобождении шины (1 — есть), 6 — упреждение (1 — есть), 5 — кэширование записи (1 — есть), 4 — поддержка пакетных команд (0 — есть), 3 — управление питанием (1 — есть), 2 — поддержка команд для сменных носителей (1 — есть), 1 — поддержка команд секретности (1 — есть), 0 — поддержка команд SMART (1 — есть)
83	Поддержка команд: бит 15 — равен 0, 14 — равен 0, 13 — команда FLUSH CACHE EXT (1 — есть), 12 — команда FLUSH CACHE (1 — есть), 11 — набор команд конфигурации (1 — есть), 10 — 48-битная адресация (1 — есть), 9 — автоматическое управление для акустики (1 — есть), 8 — SET MAX (1 — есть), 7 — резерв, 6 — SET FEATURES (1 — есть), 5 — подача питания в режиме Standby (1 — есть), 4 — уведомления для сменных носителей (0 — есть), 3 — расширенное управление питанием (1 — есть), 2 — поддержка команд CFA (1 — есть), 1 — READ/WRITE DMA QUEUED (1 — есть), 0 — DOWNLOAD MICROCODE (1 — есть)
84	Поддержка команд: 15 бит равен 0, 14 бит равен 0, 13 — резерв, 12 — ограничение по времени для записи и чтения непрерывно (1 — есть), 11 — ограничение по времени для записи и чтения (1 — есть), 10 — бит URG для команд WRITE STREAM DMA и WRITE STREAM PIO (1 — есть), 9 — бит URG для команд READ STREAM DMA и READ STREAM PIO (1 — есть), 8 — уникальные имена (1 — есть), 7 — команда WRITE DMA QUEUED FUA EXT (1 — есть), 6 — WRITE DMA FUA EXT и WRITE MULTIPLE FUA EXT (1 — есть), 5 — набор свойств регистрации (1 — есть), 4 — набор потоковых свойств (1 — есть), 3 — Media Card (1 — есть), 2 — серийный номер для Media Card (1 — есть), 1 — самотестирование SMART (1 — есть), 0 — отчет об ошибках для SMART (1 — есть)
85	Установка набора команд или свойств: 15 бит устарел, 14 — команда NOP (1 — включить), 13 — READ BUFFER (1 — включить), 12 — WRITE BUFFER (1 — включить), 11 бит устарел, 10 — защищенная область хоста (1 — включить), 9 — DEVICE RESET (1 — включить), 8 — прерывание SERVICE (1 — включить), 7 — освобождение шины (1 — включить), 6 — упреждение (1 — включить), 5 — кэширование записи (1 — включить), 4 — упреждение (1 — включить), 3 — управление питанием (1 — включить), 2 — работа со сменными носителями (1 — включить), 1 — набор команд секретности (1 — включить), 0 — набор команд SMART (1 — включить)

Таблица 11.46 *(окончание)*

Смещение слова	Описание слова
86	Установка набора команд или свойств: биты 14—15 — резерв, 13 — команда FLUSH CACHE EXT (1 — включить), 12 — команда FLUSH CACHE (1 — включить), 11 — набор команд конфигурации (1 — включить), 10 — 48-битная адресация (1 — включить), 9 — автоматическое управление для акустики (1 — включить), 8 — SET MAX (1 — включить), 7 — резерв, 6 — SET FEATURES (1 — включить), 5 — подача питания в режиме Standby (1 — включить), 4 — уведомления для сменных носителей (0 — включить), 3 — расширенное управление питанием (1 — включить), 2 — поддержка команд CFA (1 — включить), 1 — READ/WRITE DMA QUEUED (1 — включить), 0 — DOWNLOAD MICROCODE (1 — включить)
87	Установка набора команд или свойств: 15 бит равен 0, 14 бит равен 0, 13 — резерв, 12 — ограничение по времени для записи и чтения непрерывно (1 — включить), 11 — ограничение по времени для записи и чтения (1 — включить), 10 — бит URG для команд WRITE STREAM DMA и WRITE STREAM PIO (1 — включить), 9 — бит URG для команд READ STREAM DMA и READ STREAM PIO (1 — включить), 8 — уникальные имена (1 — включить), 7 — команда WRITE DMA QUEUED FUA EXT (1 — включить), 6 — WRITE DMA FUA EXT и WRITE MULTIPLE FUA EXT (1 — включить), 5 — набор свойств регистрации (1 — включить), 4 — набор потоковых свойств (1 — включить), 3 — Media Card (1 — включить), 2 — серийный номер для Media Card (1 — включить), 1 — самотестирование SMART (1 — включить), 0 — отчет об ошибках для SMART (1 — включить)
88—254	Необязательные параметры
255	Контрольное слово: 8—15 — контрольная сумма, 0—7 — сигнатура (A5h)

Примечание

Технология SMART (Self-Monitoring, Analysis, and Reporting Technology) представляет собой систему автоматического самотестирования, сбора и анализа данных, используемых производителями устройств хранения данных для диагностики и выявления на ранней стадии потенциальных проблем. Это технология позволяет не только определить ошибки в работе оборудования, но и предотвратить потерю данных.

11.1.4.6. Команда *IDENTIFY PACKET DEVICE*

Команда позволяет получить различную информацию о пакетном устройстве. Является обязательной для всех устройств ATAPI. Формат команды показан в табл. 11.47.

Для инициализации команды следует в регистр выбора устройства 1x6h записать номер устройства в бит 4 (DEV) (0 или 1), а затем в регистр команд 1x7h записать код команды ECh.

Таблица 11.47. *Формат команды* `IDENTIFY PACKET DEVICE`

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Не используется							
1x3h	Не используется							
1x4h	Не используется							
1x5h	Не используется							
1x6h	1	Резерв	1	DEV	Резерв	Резерв	Резерв	Резерв
1x7h	ECh							

При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройств `1x6h` будет хранить номер выбранного устройства.
- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

После выполнения команды устройство возвращает через регистр данных 256 слов (по 16 битов), описывающих параметры дисковода. Все зарезервированные биты и слова установлены в 0. Формат возвращаемых данных представлен в табл. 11.48.

Таблица 11.48. *Формат данных для команды* `IDENTIFY PACKET DEVICE`

Смещение слова	Описание слова
0	Конфигурация устройства: биты 14—15 — тип интерфейса (10b — ATAPI), 13 — резерв, 8—12 — тип устройства (см. табл. 9.69), 7 — устройство поддерживает сменные носители (1 — да), 5—6 — время установки сигнала DRQ (00b — через 3 мс после приема команды, 10b — через 50 мс после приема команды), 3—4 — резерв, 2 — неполные данные (1 — получены не все возможные данные), 0—1 — поддерживаемый размер пакетной команды (00b— 12 байт, 01b — 16 байт)
1—9	Зарезервированы
10—19	Серийный номер устройства (содержит 20 ASCII-символов)
20—22	Не используются
23—26	Номер версии (содержит 8 ASCII-символов)
27—46	Номер модели (содержит 40 ASCII-символов)

Таблица 11.48 *(продолжение)*

Смещение слова	Описание слова
47—48	Зарезервированы
49	Поддерживаемые возможности: бит 15 — чередование для DMA (1 — есть), 14 — очередь команд (1 — есть), 13 — перекрытие команд (1 — есть), 12 — программный сброс (устарел), 11 — поддержка сигнала IORDY (0 — есть, 1 — нет), 10 — блокировка сигнала IORDY (1 — есть), бит 9 равен 1, 8 — поддержка DMA (1 — есть), 0—7 не используются
50—52	Зарезервированы
53	Состав: биты 3—15 зарезервированы, 2 — допустимость слова 88 (1 — поле со смещением 88 допустимо), 1 — слов 64—70 (1 — допустимы), 0 — устарел
54—62	Зарезервированы
63	Состав: биты 11—15 резерв, 10 — режим Multiword DMA 2 (1 — выбран), 9 — режим Multiword DMA 1 (1 — выбран), 3—7 — резерв, 2 — поддержка режима Multiword DMA 2 (1 — есть), 1 — поддержка режима Multiword DMA 1 (1 — есть), 0 — поддержка режима Multiword DMA 0 (1 — есть)
64	Состав: биты 8—15 — резерв, 0—7 — поддержка режимов PIO
65	Минимальное время передачи для режима Multiword DMA в наносекундах
66	Рекомендуемое производителем время передачи для режима Multiword DMA в наносекундах
67	Минимальное время передачи для режима PIO в наносекундах
68	Минимальное время передачи для режима PIO с IORDY в наносекундах
69—79	Необязательные и зарезервированные поля
80	Главный (major) номер версии интерфейса: биты 8—15 — резерв, 7 — поддержка версии ATA/ATAPI-7 (1 — есть), 6 — поддержка версии ATA/ATAPI-6 (1 — есть), 5 — поддержка версии ATA/ATAPI-5 (1 — есть), 4 — поддержка версии ATA/ATAPI-4 (1 — есть), 3 — поддержка версии ATA/ATAPI-3 (1 — есть), 1—2 устарели, 0 — резерв
81	Дополнительный (minor) номер версии интерфейса
82	Поддержка команд: бит 15 устарел, 14 — команда `NOP` (1 — есть), 13 — команда `READ BUFFER` (1 — есть), 12 — команда `WRITE BUFFER` (1 — есть), 11 устарел, 10 — защищенная область хоста (1 — есть), 9 — команда `DEVICE RESET` (1 — есть), 8 — прерывание SERVICE (1 — есть), 7 — прерывание при освобождении шины (1 — есть), 6 — упреждение (1 — есть), 5 — кэширование записи (1 — есть), 4 — поддержка пакетных команд (0 — есть), 3 — управление питанием (1 — есть), 2 — поддержка команд для сменных носителей (1 — есть), 1 — поддержка команд секретности (1 — есть), 0 — поддержка команд Smart (1 — есть)
83—84	Зарезервированы

Таблица 11.48 *(окончание)*

Смещение слова	Описание слова
85	Установка набора команд или свойств: 15 бит устарел, 14 — команда NOP (1 — включить), 13 — READ BUFFER (1 — включить), 12 — WRITE BUFFER (1 — включить), 11 бит устарел, 10 — защищенная область хоста (1 — включить), 9 — DEVICE RESET (1 — включить), 8 — прерывание SERVICE (1 — включить), 7 — освобождение шины (1 — включить), 6 — упреждение (1 — включить), 5 — кэширование записи (1 — включить), 4 — упреждение (1 — включить), 3 — управление питанием (1 — включить), 2 — работа со сменными носителями (1 — включить), 1 — набор команд секретности (1 — включить), 0 — набор команд SMART (1 — включить)
86—87	Зарезервированы
88	Режимы DMA: бит 15 — резерв, 14 — выбор Ultra DMA 6 (1 — выбран), 13 — выбор Ultra DMA 5 (1 — выбран), 12 — выбор Ultra DMA 4 (1 — выбран), 11 — выбор Ultra DMA 3 (1 — выбран), 10 — выбор Ultra DMA 2 (1 — выбран), 9 — выбор Ultra DMA 1 (1 — выбран), 8 — выбор Ultra DMA 0 (1 — выбран), 7 — резерв, 6 — поддержка Ultra DMA 6 (1 — есть), 5 — поддержка Ultra DMA 5 (1 — есть), 4 — поддержка Ultra DMA 4 (1 — есть), 3 — поддержка Ultra DMA 3 (1 — есть), 2 — поддержка Ultra DMA 2 (1 — есть), 1 — поддержка Ultra DMA 1 (1 — есть), 0 — поддержка Ultra DMA 0 (1 — есть),
89—255	Зарезервированы

11.1.4.7. Команда *IDLE*

Позволяет задавать промежуток времени для перевода устройства в режим простоя (Idle). Команда является обязательной для устройств ATA. Формат команды показан в табл. 11.49.

Таблица 11.49. *Формат команды* IDLE

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Значение времени простоя (timevalue)							
1x3h	Не используется							
1x4h	Не используется							
1x5h	Не используется							
1x6h	1	Резерв	1	DEV	Резерв	Резерв	Резерв	Резерв
1x7h	E3h							

В регистре 1x2h задается промежуток времени. Возможные значения показаны в табл. 11.50.

Таблица 11.50. *Значения времени для команды* IDLE

Код времени	Значение времени
00h	Время простоя не используется
01h-F0h	(timevalue * 5) сек
F1h-FBh	((timevalue — 240) * 30) мин
FCh	21 мин
FDh	от 8 до 12 ч
FEh	Резерв
FFh	21 мин 15 сек

Для инициализации команды следует в регистр 1x2h записать код времени и в регистр выбора устройства 1x6h — номер устройства в бит 4 (DEV) (0 или 1), а затем в регистр команд 1x7h записать код команды E3h. При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройства 1x6h будет хранить номер выбранного устройства.
- Регистр состояния 1x7h (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

В случае ошибки регистры будут иметь следующий вид:

- В регистре ошибки 1x1h (см. табл. 11.33) бит ABRT будет установлен в 1, если устройство не поддерживает команду или произошло аварийное завершение.
- Регистр выбора устройства 1x6h будет хранить номер устройства (бит 4).
- Регистр состояния 1x7h (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 1 (при сбое устройства), DRQ = 0 и ERR = 1.

11.1.4.8. Команда *IDLE IMMEDIATE*

Команда позволяет немедленно перевести устройство в режим простоя (Idle). Команда является обязательной для устройств ATA. Формат команды показан в табл. 11.51.

Для инициализации команды следует в регистр выбора устройства 1x6h записать номер устройства в бит 4 (DEV) (0 или 1), а затем в регистр команд 1x7h записать код команды E1h.

Таблица 11.51. *Формат команды* `IDLE IMMEDIATE`

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Не используется							
1x3h	Не используется							
1x4h	Не используется							
1x5h	Не используется							
1x6h	1	Резерв	1	DEV	Резерв	Резерв	Резерв	Резерв
1x7h	E1h							

При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройств `1x6h` будет хранить номер выбранного устройства.
- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

В случае ошибки регистры будут иметь следующий вид:

- В регистре ошибки `1x1h` бит ABRT будет установлен в 1, если устройство не поддерживает команду или произошло аварийное завершение.
- Регистр выбора устройства `1x6h` будет хранить номер устройства (бит 4).
- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 1 (при сбое устройства), DRQ = 0 и ERR = 1.

Рассмотрим пример для перевода жесткого диска в режим простоя, представленный в листинге 11.9.

Листинг 11.9. Перевод жесткого диска в дежурный режим

```
// функция для перевода устройства в режим простоя
void SetIdle ( )
{
   IsReadyATA ( ); // ждем, пока порт освободится
   outPort ( 0x1F6, 0xA0, 1 ); // первое устройство на первом канале
   outPort ( 0x1F7, 0xE5, 1 ); // код команды CHECK POWER MODE
}
```

В рассматриваемых примерах не обрабатываются ошибки, чтобы максимально упростить принципы использования управляющих команд.

11.1.4.9. Команда *NOP*

Команда позволяет прервать все невыполненные команды, размещенные в очереди, и установить бит аварийного завершения ABRT. Команда является обязательной для устройств ATAPI. Формат команды показан в табл. 11.52.

Таблица 11.52. *Формат команды* `NOP`

Регистры	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
`1x1h`	Дополнительный код							
`1x2h`	Не используется							
`1x3h`	Не используется							
`1x4h`	Не используется							
`1x5h`	Не используется							
`1x6h`	1	Резерв	1	DEV	Резерв	Резерв	Резерв	Резерв
`1x7h`	`00h`							

Для инициализации команды следует в регистр особенностей `1x1h` записать дополнительный код (табл. 11.53) и в регистр выбора устройства `1x6h` — номер устройства в бит 4 (DEV) (0 или 1), а затем в регистр команд `1x7h` записать код команды `00h`.

Таблица 11.53. *Дополнительный код для команды* `NOP`

Дополнительный код	Описание
`00h` (`NOP`)	Прерывает выполнение очереди команды и возвращает аварийное завершение (ABRT)
`01h` (`NOP` Auto Poll)	Не прерывает выполнение команд и возвращает аварийное завершение (ABRT)
`02h-FFh` (резерв)	Не прерывает выполнение команд и возвращает аварийное завершение (ABRT)

Данная команда всегда возвращает ошибку. В регистре ошибки `1x1h` бит ABRT устанавливается в 1, а регистр состояния `1x7h` (см. табл. 11.36) имеет вид: BSY = 0, DRDY = 1, DF = 1 (при сбое устройства), DRQ = 0 и ERR = 1.

11.1.4.10. Команда *PACKET*

Команда позволяет подготовить устройство для последующей передачи пакетной команды. Команда используется только для устройств ATAPI. Важно помнить, что данную команду всегда нужно вызывать перед применением любой пакетной команды из набора SCSI. Формат команды показан в табл. 11.54.

Таблица 11.54. *Формат команды* `PACKET`

Регистры	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1x1h	Не используется						OVL	DMA
1x2h	Очередь команд					Не используется		
1x3h	Не используется							
1x4h	Количество байтов передачи (0—7)							
1x5h	Количество байтов передачи (8—15)							
1x6h	1	Резерв	1	DEV	Резерв	Резерв	Резерв	Резерв
1x7h	A0h							

В регистре особенностей `1x1h` следует установить режим передачи в бите DMA (1 — DMA, 0 — PIO) и возможность перекрытия команд в бите OVL (1 — использовать). Если применяется очередь команд, в регистр `1x2h` (3—7) следует записать значение тега для команды (от 0 до 31), иначе этот регистр не используется. В регистры `1x4h` и `1x5h` следует поместить младший и старший байты для размера блока данных, используемых последующей пакетной командой. К слову, если пакетная команда из набора SCSI не содержит дополнительных параметров и данных, то регистры `1x4h` и `1x5h` не используются.

При успешном выполнении регистры будут содержать следующую информацию:

- Регистр прерывания `1x2h` будет иметь вид: 0 — установлен в 1, 1 — установлен в 1, 2 — 0, 3—7 — значение от 0 до 31 при поддержке очереди команд.
- Регистр выбора устройства `1x6h` будет хранить номер выбранного устройства.
- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DMRD = 1 (для DMA), SERV = 1 (при использовании очереди команд), DRQ = 0 и CHK = 0.

В случае ошибки регистры будут иметь следующий вид:

- Регистр ошибки `1x1h` (см. табл. 11.33): бит ILI зависит от команды, бит EOM зависит от команды, бит ABRT = 1, поле Sense Key (4—7) будет содержать код ошибки согласно спецификации SCSI.
- Регистр прерывания `1x2h`: C/D = 1, I/O = 1, REL = 0, поле очереди команд (3—7) — значение от 0 до 31.
- Регистр выбора устройства `1x6h` будет хранить номер устройства (бит 4).
- Регистр состояния `1x7h` (см. табл. 11.36): BSY = 0, DRDY = 1, DF = 1 (при сбое устройства), SERV = 1 (если есть перекрытие), DRQ = 0 и CHK = 1.

Например, чтобы подготовить дисковод к приему простой SCSI-команды, не имеющей параметров, можно сделать так, как показано в листинге 11.10.

Листинг 11.10. Подготовка дисковода к работе с пакетными командами

```
// функция для подготовки устройства к приему пакетной команды
void SetPacket ( )
{
   IsReadyATA ( ); // ждем, пока порт освободится
   outPort ( 0x171, 0x00, 1 ); // использовать режим PIO
   outPort ( 0x172, 0x00, 1 ); // не используем
   outPort ( 0x174, 0x00, 1 ); // данных для передачи не будет
   outPort ( 0x175, 0x00, 1 ); // данных для передачи не будет
   outPort ( 0x176, 0xB0, 1 ); // первое устройство на втором канале
   outPort ( 0x177, 0xA0, 1 ); // код команды PACKET
}
```

11.1.4.11. Команда *READ DMA*

Команда позволяет считывать данные с использованием канала DMA. Команда является обязательной для устройств ATA. Формат команды показан в табл. 11.55.

Таблица 11.55. *Формат команды* `READ DMA`

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Число секторов							
1x3h	Адрес LBA (0—7)							
1x4h	Адрес LBA (8—15)							

Таблица 11.55 *(окончание)*

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x5h	Адрес LBA (16—24)							
1x6h	1	Резерв	1	DEV	Адрес LBA (24—27)			
1x7h	C8h							

В регистр 1x2h следует записать количество секторов (от 1 до 256), которые будут переданы. Если установить этот регистр в 00h, то будет использовано значение в 256 секторов. В регистры 1x3h, 1x4h и 1x5h записывается начальный адрес сектора, с которого начнется передача данных. В регистр выбора устройства 1x6h записывается номер выбранного устройства на канале (бит 4) и старшие 4 разряда адреса сектора (биты 0—3). В регистр команд 1x7h записывается код команды C8h.

При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройства 1x6h будет хранить номер выбранного устройства.
- Регистр состояния 1x7h (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

11.1.4.12. Команда *READ MULTIPLE*

Команда предназначена для считывания с диска определенного количества секторов. Команда является обязательной для устройств ATA. Формат команды показан в табл. 11.56.

Таблица 11.56. *Формат команды* *READ MULTIPLE*

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Число секторов							
1x3h	Адрес LBA (0—7)							
1x4h	Адрес LBA (8—15)							
1x5h	Адрес LBA (16—24)							
1x6h	1	Резерв	1	DEV	Адрес LBA (24—27)			
1x7h	C4h							

В регистр 1x2h следует записать количество секторов (от 1 до 256), которые будут переданы. Если установить этот регистр в 00h, то будет использовано значение 256 секторов. В регистры 1x3h, 1x4h и 1x5h записывается начальный адрес сектора, с которого начнется передача данных. В регистр выбора устройства 1x6h записывается номер выбранного устройства на канале (бит 4) и старшие 4 разряда адреса сектора (биты 0—3). В регистр команд 1x7h записывается код команды C4h.

При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройства 1x6h будет хранить номер выбранного устройства.
- Регистр состояния 1x7h (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

11.1.4.13. Команда *READ SECTOR (S)*

Команда позволяет прочитать с диска указанное количество секторов. Команда является обязательной для устройств ATA. Формат команды показан в табл. 11.57.

Таблица 11.57. *Формат команды* READ SECTOR (S)

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Число секторов							
1x3h	Адрес LBA (0—7)							
1x4h	Адрес LBA (8—15)							
1x5h	Адрес LBA (16—24)							
1x6h	1	Резерв	1	DEV	Адрес LBA (24—27)			
1x7h	20h							

В регистр 1x2h следует записать количество секторов (от 1 до 256), которые будут переданы. Если установить этот регистр в 00h, то будет использовано значение в 256 секторов. В регистры 1x3h, 1x4h и 1x5h записывается начальный адрес сектора, с которого начнется передача данных. В регистр выбора устройства 1x6h записывается номер выбранного устройства на канале (бит 4) и старшие 4 разряда адреса сектора (биты 0—3). В регистр команд 1x7h записывается код команды 20h.

При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройства `1x6h` будет хранить номер выбранного устройства.
- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

В случае ошибки команда завершит выполнение. При этом состояние регистров будет таким:

- Регистр ошибки `1x1h`: бит EOM (1) будет установлен в 1 при отсутствии сменного носителя, бит ABRT (2) будет установлен в 1; бит MCR (3) будет установлен в 1 при наличии сигнала для смены носителя; бит IDNF (4) будет установлен в 1, если указанный адрес не найден; бит MC (5) будет установлен в 1 при попытке смены носителя во время выполнения команды; бит UNC (6) будет установлен в 1 при некорректных данных.
- Регистры `1x3h`, `1x4h` и `1x5h` будут содержать адрес сектора, при чтении которого произошла ошибка.
- Регистр выбора устройства `1x6h` будет хранить номер устройства (бит 4) и младшие разряды адреса сектора.
- Регистр состояния `1x7h` (см. табл. 11.36): ERR = 1, DRQ = 0, DF = 1 (при сбое устройства), DRDY = 1 и BSY = 0.

Рассмотрим пример кода для чтения сектора жесткого диска, представленный в листинге 11.11.

Листинг 11.11. Чтение сектора жесткого диска

```
// пишем функцию для чтения одного сектора
void ReadSector ( )
{
  // буфер для данных
   WORD wData[256];
   IsReadyATA ( ); // ждем, пока порт освободится
   outPort ( 0x1F2, 0x01, 1 ); // читаем один сектор
   outPort ( 0x1F3, 0x01, 1 ); // номер сектора
   outPort ( 0x1F4, 0x00, 1 ); // цилиндр 0
   outPort ( 0x1F5, 0x00, 1 ); // старший байт цилиндра равен 0
   outPort ( 0x1F6, 0xA0, 1 ); // первое устройство на первом канале
                               // и режим CHS
   outPort ( 0x1F7, 0x20, 1 ); // код команды PACKET
   IsReadyATA ( ); // ждем, пока порт освободится
```

```
    // читаем данные в буфер
    for ( int i = 0; i < 256; i++ )
    {
      inPort ( 0x1F0, ( PDWORD ) &( wData[i] ), 2 );
    }
}
```

Функцию можно сделать универсальной, добавив аргументы для номера сектора, числа секторов и указателя на буфер данных. При этом, естественно, следует изменить код внутри самой функции.

11.1.4.14. Команда *READ VERIFY SECTOR (S)*

Команда позволяет читать данные с диска с проверкой, но, в отличие от предыдущей команды, не передает их в компьютер. Команда является обязательной для устройств ATA. Формат команды показан в табл. 11.58.

Таблица 11.58. *Формат команды* `READ VERIFY SECTOR (S)`

Регистры	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1x1h	Не используется							
1x2h	Число секторов							
1x3h	Адрес LBA (0—7)							
1x4h	Адрес LBA (8—15)							
1x5h	Адрес LBA (16—24)							
1x6h	1	Резерв	1	DEV	Адрес LBA (24—27)			
1x7h	40h							

В регистр `1x2h` следует записать количество секторов (от 1 до 256), которые будут переданы. Если установить этот регистр в `00h`, то будет использовано значение в 256 секторов. В регистры `1x3h`, `1x4h` и `1x5h` записывается начальный адрес сектора, с которого начнется передача данных. В регистр выбора устройства `1x6h` записывается номер выбранного устройства на канале (бит 4) и старшие 4 разряда адреса сектора (биты 0—3). В регистр команд `1x7h` записывается код команды `40h`.

При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройства `1x6h` будет хранить номер выбранного устройства.

- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

В случае ошибки команда завершит выполнение. При этом состояние регистров будет таким:

- Регистр ошибки `1x1h`: бит NM (1) будет установлен в 1 при отсутствии сменного носителя, бит ABRT (2) будет установлен в 1; бит MCR (3) будет установлен в 1 при наличии сигнала для смены носителя; бит IDNF (4) будет установлен в 1, если указанный адрес не найден; бит MC (5) будет установлен в 1 при попытке смены носителя во время выполнения команды; бит UNC (6) будет установлен в 1 при некорректных данных.
- Регистры `1x3h`, `1x4h` и `1x5h` будут содержать адрес сектора, при чтении которого произошла ошибка.
- Регистр выбора устройства `1x6h` будет хранить номер устройства (бит 4) и младшие разряды адреса сектора.
- Регистр состояния `1x7h` (см. табл. 11.36): ERR = 1, DRQ = 0, DF = 1 (при сбое устройства), DRDY = 1 и BSY = 0.

11.1.4.15. Команда *SLEEP*

Команда позволяет перевести устройство в режим "сна". Команда является обязательной для устройств ATA. Формат команды показан в табл. 11.59.

Таблица 11.59. *Формат команды* `SLEEP`

Регистры	Биты							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
`1x1h`	Не используется							
`1x2h`	Не используется							
`1x3h`	Не используется							
`1x4h`	Не используется							
`1x5h`	Не используется							
`1x6h`	1	Резерв	1	DEV	Не используется			
`1x7h`	`E6h`							

При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройства `1x6h` будет хранить номер выбранного устройства.

- Регистр состояния 1x7h (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

В случае ошибки регистры будут иметь следующий вид:

- В регистре ошибки 1x1h бит ABRT будет установлен в 1, если устройство не поддерживает команду или произошло аварийное завершение.
- Регистр выбора устройства 1x6h будет хранить номер устройства (бит 4).
- Регистр состояния 1x7h (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 1 (при сбое устройства), DRQ = 0 и ERR = 1.

11.1.4.16. Команда *WRITE SECTOR (S)*

Команда позволяет записать указанное количество секторов на диск. Команда является обязательной для устройств ATA. Формат команды показан в табл. 11.60.

Таблица 11.60. *Формат команды* `WRITE SECTOR (S)`

Регистры	Биты							
	7	6	5	4	3	2	1	0
1x1h	Не используется							
1x2h	Число секторов							
1x3h	Адрес LBA (0—7)							
1x4h	Адрес LBA (8—15)							
1x5h	Адрес LBA (16—24)							
1x6h	1	Резерв	1	DEV	Адрес LBA (24—27)			
1x7h	30h							

В регистр 1x2h следует записать количество секторов (от 1 до 256), которые будут переданы. Если установить этот регистр в 00h, то будет использовано значение в 256 секторов. В регистры 1x3h, 1x4h и 1x5h записывается начальный адрес сектора, с которого начнется передача данных. В регистр 1x6h записывается номер выбранного устройства на канале (бит 4) и старшие 4 разряда адреса сектора (биты 0—3). В регистр команд 1x7h записывается код команды 30h.

При успешном выполнении регистры будут содержать следующую информацию:

- Регистр выбора устройства 1x6h будет хранить номер выбранного устройства.

- Регистр состояния `1x7h` (см. табл. 11.36) будет иметь вид: BSY = 0, DRDY = 1, DF = 0, DRQ = 0 и ERR = 0.

В случае ошибки команда завершит выполнение. При этом состояние регистров будет таким:

- Регистр ошибки `1x1h`: бит NM (1) будет установлен в 1 при отсутствии сменного носителя, бит ABRT (2) будет установлен в 1; бит MCR (3) будет установлен в 1 при наличии сигнала для смены носителя; бит IDNF (4) будет установлен в 1, если указанный адрес не найден; бит MC (5) будет установлен в 1 при попытке смены носителя во время выполнения команды; бит UNC (6) будет установлен в 1 при некорректных данных.
- Регистры `1x3h`, `1x4h` и `1x5h` будут содержать адрес сектора, при чтении которого произошла ошибка.
- Регистр выбора устройства `1x6h` будет хранить номер устройства (бит 4) и младшие разряды адреса сектора.
- Регистр состояния `1x7h` (см. табл. 11.36): ERR = 1, DRQ = 0, DF = 1 (при сбое устройства), DRDY = 1 и BSY = 0.

Перед тем как приводить пример работы с этой командой, хочу предупредить читателей о серьезной опасности выполнения кода записи на диск. *Пример приводится исключительно в демонстрационных целях и его ни в коем случае не следует повторять*. Игнорирование этого замечания может привести к потере информации или выходу из строя жесткого диска! Рассмотрим пример кода для записи сектора жесткого диска, показанный в листинге 11.12.

Листинг 11.12. Запись сектора на жесткий диск

```
// пишем функцию для записи одного сектора
void WriteSector ( )
{
   // буфер для данных
   WORD wData[256];
   memset ( &wData, 0xFF, 256 );
   IsReadyATA ( ); // ждем, пока порт освободится
   outPort ( 0x1F2, 0x01, 1 ); // записываем один сектор
   outPort ( 0x1F3, 0x01, 1 ); // номер сектора
   outPort ( 0x1F4, 0x00, 1 ); // цилиндр 0
   outPort ( 0x1F5, 0x00, 1 ); // старший байт цилиндра равен 0
   outPort ( 0x1F6, 0xA0, 1 ); // первое устройство на первом канале
                               // и режим CHS
   outPort ( 0x1F7, 0x30, 1 ); // код команды WRITE SECTOR
```

```
    IsReadyATA ( ); // ждем, пока порт освободится
    // пишем данные в порт
    for ( int i = 0; i < 256; i++ )
    {
      outPort ( 0x1F0, wData[i], 2 );
    }
}
```

На этом можно завершить тему программирования портов ввода-вывода. Многое осталось не сказанным, но полученные здесь сведения помогут вам без проблем разобраться со всем остальным. Я же хотел бы познакомить вас с некоторыми способами программирования дисковой подсистемы через интерфейс Win32.

11.2. Использование Win32 API

Для работы с дисковой подсистемой используется универсальная функция `DeviceIoControl`. Функция является каналом между программой и драйвером устройства. Мы разберем несколько способов ее использования. Для начала попробуем применить эту функцию для открывания лотка CD-ROM (или аналогичного устройства со сменными носителями). Исходный код для выполнения данной задачи представлен в листинге 11.13.

Листинг 11.13. Использование функции `DeviceIoControl` для открывания лотка

```
#include "stdafx.h"
#include <Winioctl.h>
#include <Mmsystem.h>
// значение флага и константы
#define VWIN32_DIOC_DOS_IOCTL 1
#define CF_FLAG 0x0001
// определяем структуру для хранения значений регистров
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, *PDIOC_REGISTERS;
```

```
// функция для открытия лотка
void Eject_98ME ( bOpen ) ; // открывает и закрывает лотки
                            // для всех устройств
// объявляем переменные модуля
DIOC_REGISTERS Reg;
// реализуем функцию Eject
void Eject_98ME ( bOpen )
{
   HANDLE hDevice = NULL;
   bool bResult = false;
   DWORD dwResult = 0;
   DWORD dwDrives = 0;
   int nPos = 0, iCH = 0;
   unsigned int uDriveType = 0;
   TCHAR szLetter[8];
    ZeroMemory ( &Reg, sizeof ( Reg ) );
    // определяем число установленных устройств
    dwDrives = GetLogicalDrives ( );
    if( bOpen ) // открыть лоток
    {
     while ( dwDrives )
     {
       if ( dwDrive & 1 ) // если есть диск
        {
          iCH = 0x41 + nPos;
          strcpy ( szLetter, ( const char* ) &iCH );
          strcat ( szLetter, ":\\" );
          // определяем тип устройства
          uDriveType = GetDriveType ( szLetter );
          // если устройство со сменным носителем
          if ( uDriveType == DRIVE_CDROM )
          {
           // открываем драйвер устройства
           hDevice = CreateFile ( "\\\\.\\vwin32", 0, 0, NULL, 0,
FILE_FLAG_DELETE_ON_CLOSE, NULL );
           if ( hDevice != INVALID_HANDLE_VALUE ) // не удалось
                                                  // открыть драйвер
           {
             // заполняем поля структуры
             Reg.reg_EAX = 0x440D; // номер функции
             Reg.reg_EBX = nPos + 1; // номер устройства
             Reg.reg_ECX = MAKEWORD ( 0x49, 0x08 ); // код подфункции
             Reg.reg_Flags = 0x0001; // устанавливаем флаг переноса
```

```
            // вызываем функцию DeviceIoControl
            bResult = DeviceIoControl ( hDevice, VWIN32_DIOC_DOS_IOCTL,
&Reg,
            sizeof ( Reg ), &Reg, sizeof ( Reg ), & dwResult, 0 );
            // проверяем результат операции и флаг CF
            if ( !bResult || ( Reg.reg_Flags & CF_FLAG ) )
            {
              // закрываем драйвер устройства
              CloseHandle ( hDevice );
              hDevice = NULL;
            }
           }
          // закрываем драйвер устройства
          CloseHandle ( hDevice );
          hDevice = NULL;
          }
        }
       dwDrives >>= 1;
       nPos ++;
       strcpy ( szLetter, "" );
     }
    }
    else // закрыть лотки
    {
     while ( dwDrives )
     {
      if ( dwDrives & 1 )
      {
        iCH =  0x41 + nPos;
        strcpy ( szLetter, ( const char* ) &iCH );
        strcat( szLetter, ":\\" );
        // определяем тип устройства
        uDriveType = GetDriveType ( szLetter );
        if ( uDriveType == DRIVE_CDROM ) // если устройство
                                         // со сменным носителем
        {
          // выполняем функцию MCI для закрытия лотка
          mciSendString ( "Set CDAudio Door Closed Wait" ), NULL, NULL,
NULL );
        }
       }
       dwDrives >>= 1;
       nPos ++;
```

```
       strcpy ( szLetter, "" );
      }
    }
}
```

Не забудьте добавить в опции компоновщика ссылку на библиотеку WINMM.LIB. Рассмотренная выше функция рассчитана для работы в Windows 95/98/ME. Для использования ее в Windows NT/2000/XP/SR3/Vista придется немного изменить код, согласно листингу 11.14.

Листинг 11.14. Использование функции `DeviceIoControl` в Windows NT/2000/XP/SR3/Vista

```
#include "stdafx.h"
#include <Winioctl.h>
// функция для открытия лотка
void Eject_NT ( bOpen ) ; // открывает и закрывает лотки
                          // для всех устройств
// реализуем функцию Eject
void Eject_NT ( bOpen )
{
   HANDLE hDevice = NULL;
   DWORD dwCode = 0, dwDrives = 0, dwResult = 0;
   TCHAR szLetter[8];
   TCHAR szDeviceName[15];
   int nPos = 0, iCH = 0;
   unsigned int uDriveType = 0;
   if( bOpen ) // открыть лоток
     dwCode = IOCTL_STORAGE_EJECT_MEDIA;
   else // закрыть лоток
     dwCode = IOCTL_STORAGE_LOAD_MEDIA;
   // определяем число установленных устройств
   dwDrives = GetLogicalDrives ( );
   while ( dwDrives )
   {
    if ( dwDrives & 1 ) // если есть диск
    {
      iCH = 0x41 + nPos;
      strcpy ( szLetter, ( const char* ) & iCH );
      strcat ( szLetter, ":" );
      // определяем тип устройства
      uDriveType = GetDriveType ( szLetter );
      if ( uDriveType == DRIVE_CDROM ) // если устройство
                                       // со сменными носителями
```

```
      {
        // получаем имя для устройства
        strcpy ( szDeviceName, "\\\\.\\" );
        strcat( szDeviceName, szLetter );
        // открываем устройство
        hDevice = CreateFile ( szDeviceName, GENERIC_READ,
FILE_SHARE_READ |
       FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL );
        if ( hDevice != INVALID_HANDLE_VALUE ) // если устройство открыто
        {
          // вызываем функцию DeviceIoControl
          DeviceIoControl ( hDevice, dwCode , NULL, 0, NULL, 0,
&dwResult, NULL );
          // закрываем устройство
          CloseHandle ( hDevice );
          strcpy ( szDeviceName, "" );
        }
      }
    }
    // переходим к следующему
    dwDrives >>= 1;
    nPos ++;
    strcpy ( szLetter, "" );
   }
}
```

Представленные вашему вниманию тексты кодов достаточно упрощены для понимания. Теперь поговорим о возможности блокировать лоток устройства, поддерживающего такую возможность. Для выполнения этой задачи нам потребуется все та же функция `DeviceIoControl`. Чтобы блокировать лоток в операционных системах Windows 95/98/ME, следует написать код, как показано в листинге 11.15.

Листинг 11.15. Блокировка лотка в Windows 95/98/ME

```
#include "stdafx.h"
#include <Winioctl.h>
// значение флага и константы
#define VWIN32_DIOC_DOS_IOCTL 1
#define CF_FLAG 0x0001
// определяем структуры
#pragma pack ( 1 )
typedef struct _PARAMBLOCK
```

```
{
   BYTE bOperation;
   BYTE bNumLocks;
} PARAMBLOCK, *PPARAMBLOCK;
#pragma pack ( )
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, *PDIOC_REGISTERS;
// функция для блокировки лотка
void Lock_98ME ( bLock ) ;
// объявляем переменные модуля
DIOC_REGISTERS Reg;
PARAMBLOCK lockTray;
// реализуем функцию Lock_98ME
void Lock_98ME ( bLock )
{
   HANDLE hDevice = NULL;
   bool bResult = false;
   DWORD dwResult = 0;
   DWORD dwDrives = 0;
   int nPos = 0, iCH = 0;
   unsigned int uDriveType = 0, uFlag = 0;
   TCHAR szLetter[8];
    ZeroMemory ( &Reg, sizeof ( Reg ) );
    // определяем режим блокировки
    if ( bLock ) // блокировать лоток
      uFlag = 0;
    else // разблокировать лоток
      uFlag = 1;
    // определяем число установленных устройств
    dwDrives = GetLogicalDrives ( );
    if( bOpen ) // открыть лоток
    {
     while ( dwDrives )
     {
       if ( dwDrive & 1 ) // если есть диск
        {
          iCH = 0x41 + nPos;
```

```
        strcpy ( szLetter, ( const char* ) &iCH );
        strcat ( szLetter, ":\\" );
        // определяем тип устройства
        uDriveType = GetDriveType ( szLetter );
        // если устройство со сменным носителем
        if ( uDriveType == DRIVE_CDROM )
        {
         // открываем драйвер устройства
         hDevice = CreateFile ( "\\\\.\\vwin32", 0, 0, NULL, 0,
                                FILE_FLAG_DELETE_ON_CLOSE, NULL );
         if ( hDevice != INVALID_HANDLE_VALUE ) // не удалось открыть
                                                // драйвер
         {
           // заполняем поля структуры
           lockTray.bOperation = uFlag;
           Reg.reg_EAX = 0x440D; // номер функции
           Reg.reg_EBX = nPos + 1; // номер устройства
           Reg.reg_ECX = MAKEWORD ( 0x48, 0x08 ); // код подфункции
           Reg.reg_EDX = ( DWORD ) &lockTray; // указатель на структуру
           Reg.reg_Flags = 0x0001; // устанавливаем флаг переноса
           // вызываем функцию DeviceIoControl
           bResult = DeviceIoControl ( hDevice, VWIN32_DIOC_DOS_IOCTL,
                                                                &Reg,
          sizeof ( Reg ), &Reg, sizeof ( Reg ), & dwResult, 0 );
           // проверяем результат операции и флаг CF
           if ( !bResult || ( Reg.reg_Flags & CF_FLAG ) )
           {
             // закрываем драйвер устройства
             CloseHandle ( hDevice );
             hDevice = NULL;
           }
         }
        // закрываем драйвер устройства
        CloseHandle ( hDevice );
        Sleep ( 50 ); // небольшая задержка
        hDevice = NULL;
        }
       }
      dwDrives >>= 1;
      nPos ++;
      strcpy ( szLetter, "" );
    }

}
```

Чтобы создать аналогичную функцию для Windows NT/2000/XP/SR3/Vista, следует написать код, как показано в листинге 11.16.

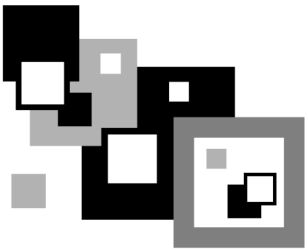
Листинг 11.16. Блокировка лотка в Windows NT/2000/XP/SR3/Vista

```
#include "stdafx.h"
#include <Winioctl.h>
// функция для открытия лотка
void Lock_NT ( bLock ) ; // открывает и закрывает лотки
                         // для всех устройств
// реализуем функцию Lock_NT
void Lock_NT ( bLock )
{
   HANDLE hDevice = NULL;
   DWORD dwDrives = 0, dwResult = 0;
   TCHAR szLetter[8];
   TCHAR szDeviceName[15];
   int nPos = 0, iCH = 0;
   unsigned int uDriveType = 0;
   PREVENT_MEDIA_REMOVAL lockTray;
   ZeroMemory ( &lockTray, sizeof ( PREVENT_MEDIA_REMOVAL ) );
   if( bLock ) // блокировать лоток
     lockTray.PreventMediaRemoval = true;
   else // разблокировать лоток
     lockTray.PreventMediaRemoval = false;
   // определяем число установленных устройств
   dwDrives = GetLogicalDrives ( );
   while ( dwDrives )
   {
    if ( dwDrives & 1 ) // если есть диск
    {
      iCH = 0x41 + nPos;
      strcpy ( szLetter, ( const char* ) & iCH );
      strcat ( szLetter, ":" );
      // определяем тип устройства
      uDriveType = GetDriveType ( szLetter );
      if ( uDriveType == DRIVE_CDROM ) // если устройство
                                       // со сменными носителями
      {
        // получаем имя для устройства
        strcpy ( szDeviceName, "\\\\.\\" );
        strcat( szDeviceName, szLetter );
```

```
        // открываем устройство
        hDevice = CreateFile ( szDeviceName, GENERIC_READ,
                                                    FILE_SHARE_READ |
        FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL );
        if ( hDevice != INVALID_HANDLE_VALUE ) // если устройство открыто
        {
          // вызываем функцию DeviceIoControl
          DeviceIoControl ( hDevice, IOCTL_STORAGE_MEDIA_REMOVAL,
                                                  (LPVOID) &lockTray,
         sizeof ( PREVENT_MEDIA_REMOVAL ), NULL, 0, &dwResult, NULL );
          // закрываем устройство
          CloseHandle ( hDevice );
          strcpy ( szDeviceName, "" );
        }
      }
    }
    // переходим к следующему
    dwDrives >>= 1;
    nPos ++;
    strcpy ( szLetter, "" );
   }
}
```

На этом я завершаю тему программирования дисковых устройств в Windows, а читателям советую самостоятельно изучить файл WINIOCTL.H на предмет других возможностей управления устройствами CD-ROM. Данный файл входит в пакет для разработки драйверов Windows 2000 DDK (Windows Driver Development Kit 2000) или в аналогичный пакет разработки драйверов для Windows Vista — Windows KMDF (Kernel Mode Driver Foundation).

ГЛАВА 12



Пространство шины PCI

Почему автор называет шину *PCI* (Peripheral Component Interconnect) пространством, спросите вы? Да, в целом шина осталась шиной и выполняет те же функции, что и всегда. Однако современная шина PCI представляет собой не просто общие каналы передачи данных для различных устройств, но выполняет глобальную управляющую функцию. Она имеет собственное адресное пространство, что позволяет получить доступ ко всем устройствам в системе, независимо от интерфейса. Но самым важным преимуществом шины является возможность гарантированного получения всех доступных портов ввода-вывода и номеров прерываний для последующей работы с устройствами. Не секрет, что немногие системные модули имеют жестко прописанные адреса портов и номера прерываний. Как узнать, где расположено то или иное устройство. На своем компьютере можно, конечно, зайти в настройки оборудования и посмотреть, а на чужом придется придумывать что-нибудь другое. Например, попросить пользователя вашей программы ввести нужные данные. Согласитесь, этот способ хорош только для продвинутых пользователей, не говоря уже о коммерческой несостоятельности такого "интерактивного" проекта. Вот для того, чтобы решить эти и другие проблемы, следует пользоваться возможностями шины PCI. Только в этом случае вам удастся написать приложение, не зависящее от конфигурации оборудования.

На сегодняшний момент активно используются следующие стандарты шины PCI:

- PCI с частотой 33 МГц и полосой пропускания 133 Мбайт/с;
- PCI с частотой 66 МГц и полосой пропускания 266 Мбайт/с;
- PCI-X с частотой 66 МГц и полосой пропускания 266 Мбайт/с;
- PCI-X с частотой 133 МГц и полосой пропускания 533 Мбайт/с;

- PCI-X с частотой 266 МГц и полосой пропускания 1066 Мбайт/с;
- PCI-X с частотой 533 МГц и полосой пропускания 2131 Мбайт/с.

Как видите, на смену привычной шины PCI пришла более скоростная и эффективная *PCI-X* (PCI Express). С ее появлением намного увеличилась пропускная способность и скорость обмена данными. Вместе с тем, расширились возможности подключения различных интерфейсов к новой шине.

Мы с вами познакомимся с программированием базовой шины PCI посредством портов ввода-вывода.

12.1. Общие сведения

Современная шина PCI объединяет практически все устройства в компьютере: аудио и видео (AGP и PCI), сетевые и SCSI, южные и северные мосты. Через мосты осуществляется связь с процессором, памятью, кэшем, дисковой подсистемой, мышью и клавиатурой. Шина PCI не просто связывает перечисленные устройства, но и представляет удобный интерфейс для доступа и управления ими. Для управления используются так называемые *циклы шины* с частотой 33 МГц, 66 МГц, 133 МГц (PCI-X), 266 МГц (PCI-X) и 533 МГц (PCI-X). Данные частоты являются общими для всех устройств на шине и позволяют синхронизировать работу. Чтобы упорядочить функционирование разнообразного оборудования, шина PCI назначает каждому свой номер. По этому номеру впоследствии определяется тот или иной тип устройства. Кроме того, шина PCI предоставляет общее конфигурационное пространство, которое можно использовать из своей программы для перечисления подключенных устройств и получения различной информации о них (адреса ввода-вывода, идентификационные номера, номера прерываний). Конфигурационное пространство представляет собой шаблон размером 256 байт. Первые 16 байт являются неизменными для любых типов устройств. Остальные могут иметь специфические значения, зависящие от самого устройства. В эти байты записывается различная опознавательная информация, по которой пользователь может определить категорию устройства. Формат пространства шины PCI показан в табл. 12.1. Поскольку программист может получить доступ к пространству шины PCI в режиме чтения и записи, необходимо всегда придерживаться одного правила: *для изменения параметров (в том числе отдельных битов) необходимо сначала прочитать значение параметра, а затем, изменив желаемые, биты записать обратно.* В обязательном порядке производителями должны поддерживаться следующие поля: Device ID, Vendor ID, Status, Command, Class Code, Revision ID и Header Type. Остальные поля необязательны.

Таблица 12.1. *Конфигурационное пространство шины PCI*

Смещение	Биты			
	31 24	23 16	15 8	7 0
00h	Device ID		Vendor ID	
04h	Status		Command	
08h	Class Code			Revision ID
0Ch	BIST	Header Type	Latency Timer	Cache Line Size
10h	Base Address Registers			
14h				
18h				
1Ch				
20h				
24h				
28h	Cardbus CIS Pointer			
2Ch	Subsystem ID		Subsystem Vendor ID	
30h	Expansion ROM Base Address			
34h	Резерв			Совместимость
38h	Резерв			
3Ch	Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line
40h–FFh	Определяются производителем устройства			

Описание табл. 12.1.

- Vendor ID — определяет уникальный идентификатор производителя устройства. Только для чтения. Идентификаторы для наиболее известных у нас устройств представлены в табл. 12.2.

Таблица 12.2. *Идентификаторы производителей устройств (Vendor ID)*

Vendor ID	Производитель
0E11h	Compaq
1000h	Symbios Logic (LSI Logic)
1002h	ATI Technologies
1005h	Advance Logic (ADL) Inc
1008h	Epson
100Ah	Phoenix Technologies

Таблица 12.2 *(продолжение)*

Vendor ID	Производитель
100Bh	National Semiconductor
1013h	Cirrus Logic
1014h	IBM
1016h	Fujitsu ICL Personal Systems
1019h	Elitegroup Computer Sys
101Ch	Western Digital
1020h	Hitachi Computer Electronics
1022h	Advanced Micro Devices (AMD)
1023h	Trident Microsystems
1024h	Zenith Data Systems
1025h	Acer Inc
1028h	Dell Computer Corp
1029h	Siemens Nixdorf AG
102Bh	Matrox Graphics Inc
102Eh	Olivetti Advanced Technology
102Fh	Toshiba America
1033h	NEC Electronics Hong Kong
1037h	Hitachi Micro Systems Inc
1039h	Silicon Integrated Systems (SiS)
103Ah	Seiko Epson Corp
103Ch	Hewlett-Packard Company
1043h	Asustek Computer Inc
1044h	Adaptec
1045h	OPTi Inc
1048h	ELSA GmbH
104Ch	Texas Instruments (TI)
104Dh	Sony Corp
1054h	Hitachi Ltd
1057h	Motorola
105Ah	Promise Technology
1064h	Alcatel CIT

Таблица 12.2 *(продолжение)*

Vendor ID	Производитель
1067h	Mitsubishi Electronics
106Ch	Hyundai Electronics America
1070h	Daewoo Telecom Ltd
1073h	YAMAHA Corp
1076h	Chaintech Computer Co Ltd
1078h	Cyrix Corp
107Ch	LG Electronics
107Fh	Data Technology Corp (DTC)
108Eh	Sun Microsystems
1092h	Diamond Multimedia Systems
1099h	Samsung Electronics Co Ltd
109Dh	Zida Technologies Ltd
10A2h	Quantum Corp
10A9h	Silicon Graphics
10B7h	3COM Corp, Networking Division
10B9h	Acer Labs Incorporated (ALI)
10C4h	Award Software International Inc
10CAh	Fujitsu Microelectronic
10DEh	Nvidia Corp
10E1h	Tekram Technology Corp Ltd
1102h	Creative Labs
1106h	VIA Technologies Inc
1131h	Philips Semiconductors
115Fh	MAXTOR Corp
117Ah	A-Trend Technology
1180h	Ricoh Co Ltd
11ADh	Lite-On Communications Inc
11BDh	Pinnacle Systems Inc
11CFh	Pioneer Electronic Corp
1240h	Marathon Technologies Corp
125Dh	ESS Technology

Таблица 12.2 *(окончание)*

Vendor ID	Производитель
126Ah	Lexmark International Inc
1274h	Ensoniq
127Ah	Rockwell Semiconductor Systems
12B9h	3COM Corp, Modem Division
12D2h	STB/Nvidia/SGS Thompson
1414h	Microsoft
142Eh	Vitec Multimedia
1458h	Giga-Byte Technology
1563h	A-Trend Technology Co Ltd
4843h	Hercules Computer Technology Inc
5333h	S3 Inc
8086h	Intel Corporation
9004h–9005h	Adaptec
A0A0h	Aopen Inc
E000h	Winbond

- Device ID — идентификатор определенного устройства. Только для чтения. Идентификаторы для наиболее известных у нас устройств представлены в табл. 12.3.

Таблица 12.3. *Идентификаторы популярных устройств*

Device ID	Название устройства
00D1h	i740 PCI
7020h	USB Controller
4747h	Rage 3D Pro
4966h	RV250 Radeon 9000/9000 Pro
4E45h	R300 Radeon 9500 Pro
4E49h	R350 Radeon 9800
5043h	Rage 128 Pro AGP 4x
5048h–5058h	Rage 128
5144h	Radeon DDR

Таблица 12.3 *(продолжение)*

Device ID	Название устройства
5159h	Radeon VE
5245h	Rage 128 GL PCI
6005h	Crystal CS4281 PCI Audio
0002h	PCI to MCA Bridge (IBM)
0047h	PCI to PCI Bridge (IBM)
7004h	AMD-751 CPU to PCI Bridge (AMD)
7007h	AMD-751 PCI to AGP Bridge (AMD)
7404h	AMD-755 USB Open Host Controller (AMD)
7411h	AMD-766 EIDE Controller (AMD)
7449h	AMD-768 USB Controller
7469h	AMD-8111 UltraATA/133 Controller
5240h	EIDE Controller (Acer)
051Eh	MGA-1164SG Mystique 220 AGP (Matrox)
0520h	MGA-G200B Chipset (Matrox)
0525h	MGA-G400 Chipset (Matrox)
1000h	MGA-G100 Chipset PCI (Matrox)
1525h	Fusion G450 AGP (Matrox)
6057h	MiroVIDEO DC10/DC30 (Miro)
1521h	ALI M1521 Aladdin III CPU to PCI Bridge (Acer)
1531h	ALI M1531 Aladdin IV/IV+ CPU to PCI Bridge (Acer)
0018h	Riva 128
0019h	Riva 128ZX
0020h	Riva TNT
0028h	RIVA TNT2
002Bh	Riva TNT2
002Dh	RIVA TNT2 Model 64
002Fh	Vanta
00A0h	RIVA TNT2 Aladdin
0100h	GeForce 256
0101h	GeForce 256 DDR

Таблица 12.3 *(продолжение)*

Device ID	Название устройства
0102h	GeForce 256 Ultra
0110h	GeForce2 MX
0112h	GeForce2 MX Ultra
0150h	GeForce2 GTS
0151h	GeForce2 GTS DDR
0152h	GeForce2 Ultra
0200h	GeForce3
0002h	EMU10K1 Audio Chipset (Creative)
7002h	PCI Gameport Joystick (Creative)
0305h	VT8363 KT133 System Controller (VIA)
0680h	VT82C680 Apollo P6 (VIA)
1106h	VT82C570 MV IDE Controller (VIA)
0001h	3Dfx Voodoo
0002h	3Dfx Voodoo2
0003h	3Dfx Voodoo Banshee
0005h	3Dfx Voodoo3
00EDh	nForce3 250 PCI-PCI Bridge
00FCh	NVBR02.4 GeForce PCX 5300
01A4h	nForce AGP Controller
025Bh	NV25GL.4 Quadro4 700 XGL
0282h	NV28.3 GeForce4 Ti 4800 SE
0344h	NV36.4 GeForce FX 5700VE
5880h	5880 AudioPCI
8813h	S3 Trio64
8814h	S3 Trio64UV+
8900h	S3 Trio64V2/DX
8A10h	S3 ViRGE/GX2
8A20h	S3 Savage3D
8A22h	S3 Savage4
8A26h	S3 ProSavage
9102h	S3 Savage 2000

Таблица 12.3 *(окончание)*

Device ID	Название устройства
0122h	82437FX 430FX (Intel)
1222h	82092AA EIDE Controller (Intel)
1239h	82371FB 430FX PCI EIDE Controller (Intel)
2412h	82801AA 8xx Chipset USB Controllers (Intel)
2416h	82801AA 8xx Chipset AC'97 PCI Modem (Intel)
2418h	82801AA 8xx Chipset Hub to PCI Bridge (Intel)
2421h	82801AB 8xx Chipset IDE Controller (Intel)
2422h	82801AB 8xx Chipset USB Controller (Intel)
2441h	82801BA Bus Master IDE Controller (Intel)
244Ah	82801BAM (ICH2) UltraDMA/100 IDE Controller (Intel)
2561h	82845G/GL/GV/GE/PE Host-to-AGP Bridge
2573h	82865G/PE/P, 82848P PCI-to-CSA Bridge
265Ch	82801FB/FR/FW/FRW USB 2.0 EHCI Controller
277Ch	82975X Intel 975X Express Chipset
4233h	Intel 4965AGN Intel® Wireless WiFi Link 4965AGN
7010h	82371SB PIIX3 EIDE Controller (Intel)
7020h	82371SB PIIX3 USB Controller (Intel)
7111h	82371AB/EB/MB PIIX4 EIDE Controller (Intel)
7112h	82371AB/EB/MB PIIX4 USB Controller (Intel)
7113h	82371AB/EB/MB PIIX4 Power Management Controller (Intel)
719Ah	82443MX USB Universal Host Controller
7605h	82372FB IEEE1394 OpenHCI Host Controller
9970h	W9970CF VGA controller

- Revision ID — определяет номер версии устройства в дополнении к Device ID. Поле предназначено только для чтения.
- Header Type — определяет информацию по смещению `10h`. Бит 7 указывает на многофункциональное устройство (1). Биты 0—6 идентифицируют формат дальнейшей информации (`00h` — формат согласно данной таблице, `01h` — формат определяется спецификацией моста PCI-to-PCI). Поле предназначено только для чтения.

- Class Code — определяет класс устройства. Поле предназначено только для чтения. Формат этого поля представлен в табл. 12.4. Код базового класса и подкласса позволяют идентифицировать тип устройства. Возможные значения кодов для базового класса показаны в табл. 12.5. Значения кодов подкласса и интерфейса перечислены в табл. 12.6. Формат байта интерфейса для контроллеров IDE представлен в табл. 12.7.

Таблица 12.4. *Формат поля Class Code*

Смещение	0Bh	0Ah	09h
Описание	Код базового класса	Код подкласса	Интерфейс

Таблица 12.5. *Коды базового класса*

Код базового класса	Описание
00h	Устройство появилось до введения кодов класса
01h	Контроллер устройства хранения большой емкости
02h	Сетевой контроллер
03h	Контроллер дисплея
04h	Мультимедийное устройство
05h	Контроллер памяти
06h	Устройство моста
07h	Обычный контроллер связи
08h	Системная периферия
09h	Устройство ввода
0Ah	Стыковочный узел
0Bh	Процессор
0Ch	Контроллер последовательной шины
0Dh	Контроллер для беспроводных интерфейсов
0Eh	Интеллектуальный контроллер ввода-вывода
0Fh	Контроллер спутниковой связи
10h	Контроллер кодирования/декодирования
11h	Контроллер-накопитель обработки сигналов
12h–FEh	Резерв
FFh	Устройство не подходит ни под один класс

Таблица 12.6. *Коды подкласса и интерфейса*

Базовый класс	Подкласс	Интерфейс	Описание
00h	00h	00h	Все выпущенные ранее устройства, кроме VGA-совместимых
	01h	00h	VGA-совместимые устройства
01h	00h	00h	Контроллер SCSI
	01h	xxh	Контроллер IDE
	02h	00h	Контроллер флоппи-дисковода
	03h	00h	Контроллер IPI
	04h	00h	Контроллер RAID
	05h	20h	Контроллер ATA с одиночным DMA
		30h	Контроллер ATA с цепочкой DMA
	06h	00h	Контроллер SATA
		01h	Контроллер SATA с интерфейсом AHCI 1.0
	07h	00h	Контроллер SAS
	80h	00h	Контроллер для другого устройства хранения
02h	00h	00h	Контроллер Ethernet
	01h	00h	Контроллер Token Ring
	02h	00h	Контроллер FDDI
	03h	00h	Контроллер ATM
	04h	00h	Контроллер ISDN
	80h	00h	Сетевой контроллер другого типа
03h	00h	00h	VGA-совместимый контроллер
		01h	8514-совместимый контроллер
	01h	00h	Контроллер XGA
	02h	00h	Контроллер 3D
	80h	00h	Контроллер дисплея другого типа
04h	00h	00h	Устройство для работы с видео
	01h	00h	Устройство для работы с аудио
	02h	00h	Коммуникационное компьютерное устройство
	80h	00h	Другое мультимедийное устройство
05h	00h	00h	Контроллер оперативной памяти
	01h	00h	Контроллер флэш-памяти
	80h	00h	Другой тип контроллера памяти

Таблица 12.6 *(продолжение)*

Базовый класс	Подкласс	Интерфейс	Описание
06h	00h	00h	Мост хоста
	01h	00h	Мост ISA
	02h	00h	Мост EISA
	03h	00h	Мост MCA
	04h	00h	Мост PCI-to-PCI
	05h	00h	Мост PCMCIA
	06h	00h	Мост NuBus
	07h	00h	Мост CardBus
	0Ah	00h	Мост InfiniBand
	80h	00h	Другой тип моста
07h	00h	00h	Контроллер последовательного порта (XT-совместимый)
		01h	16450-совместимый контроллер последовательного порта
		02h	16550-совместимый контроллер последовательного порта
		03h	16650-совместимый контроллер последовательного порта
		04h	16750-совместимый контроллер последовательного порта
		05h	16850-совместимый контроллер последовательного порта
		06h	16950-совместимый контроллер последовательного порта
	01h	00h	Контроллер параллельного порта
		01h	Контроллер двунаправленного параллельного порта
		02h	Контроллер параллельного порта ECP 1.X
		03h	Контроллер IEEE 1284
		FEh	Основное устройство (не контроллер) IEEE 1284
	02h	00h	Многопортовый последовательный контроллер
	80h	00h	Контроллер другого устройства связи

Таблица 12.6 *(продолжение)*

Базовый класс	Подкласс	Интерфейс	Описание
08h	00h	00h	Контроллер прерываний 8259
		01h	Контроллер прерываний ISA
		02h	Контроллер прерываний EISA
	01h	00h	Контроллер DMA 8237
		01h	Контроллер DMA ISA
		02h	Контроллер DMA EISA
	02h	00h	Контроллер системного таймера 8254
		01h	Контроллер системного таймера ISA
		02h	Контроллер системного таймера EISA
	03h	00h	Контроллер часов реального времени
		01h	Контроллер часов реального времени ISA
	04h	00h	Контроллер для "горячего" подключения
	80h	00h	Другой тип контроллера
09h	00h	00h	Контроллер клавиатуры
	01h	00h	Контроллер дигитайзера
	02h	00h	Контроллер мыши
	03h	00h	Контроллер сканера
	04h	00h (10h)	Контроллер игрового порта
	80h	00h	Другой тип контроллера ввода
0Ah	00h	00h	Контроллер стыковочного узла
	80h	00h	Другой тип контроллера
0Bh	00h	00h	386
	01h	00h	486
	02h	00h	Pentium
	10h	00h	Alpha
	20h	00h	Power PC
	30h	00h	MIPS
	40h	00h	Сопроцессор
0Ch	00h	00h	Контроллер IEEE 1394
	01h	00h	ACCESS
	02h	00h	SSA

Таблица 12.6 *(окончание)*

Базовый класс	Подкласс	Интерфейс	Описание
	03h	00h	USB
	04h	00h	Fibre Channel
	05h	00h	SMBus (шина управления питанием)
	06h	00h	InfiniBand
	07h	00h	Интерфейс IPMI SMIC
	80h	00h	Другой тип контроллера последовательной шины
0Dh	00h	00h	Совместимый контроллер iRDA
	01h	00h	Контроллер IR
	10h	00h	Контроллер RF
	11h	00h	Bluetooth
	12h	00h	Broadband
	80h	00h	Другой тип беспроводного контроллера
0Eh	00h	xxh	Интеллектуальный контроллер I2O
	00h	00h	Сообщение FIFO по смещению 40h
	80h	00h	Другой тип интеллектуального контроллера
0Fh	01h	00h	Контроллер TV
	02h	00h	Контроллер аудио
	03h	00h	Голосовой контроллер
	04h	00h	Контроллер данных
	80h	00h	Другой тип спутникового контроллера связи
10h	00h	00h	Сетевой или компьютерный контроллер кодирования/декодирования
	10h	00h	Расширенный контроллер кодирования/декодирования
	80h	00h	Другой тип контроллера кодирования/декодирования
11h	00h	00h	Контроллер модулей DPIO
	01h	00h	Счетчик производительности
	10h	00h	Контроллер синхронизации связи с тестированием времени и частоты
	20h	00h	Управляющая карта
	80h	00h	Другой тип контроллера обработки сигналов

Таблица 12.7. *Формат байта интерфейса для контроллеров IDE*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	Master	0	0	0	Режим	Режим	Режим	Режим

Описание табл. 12.7.

- Бит 0 определяет режим работы для первого канала контроллера. Если этот бит установлен в 1, то канал работает в режиме PCI, иначе в режиме совместимости.
- Бит 1 указывает поддержку первичным каналом фиксированного режима выполнения операций (0 — режим установлен, 1 — поддерживаются оба режима).
- Бит 2 определяет режим работы для вторичного канала контроллера. Если бит установлен в 1, то канал работает в режиме PCI, иначе в режиме совместимости.
- Бит 3 указывает поддержку вторичным каналом фиксированного режима выполнения операций (0 — режим установлен, 1 — поддерживаются оба режима).
- Биты 4—6 зарезервированы и должны быть установлены в 0.
- Бит 7 определяет поддержку выбора ведущего устройства на канале.

❐ Command — это поле открывает доступ к командному регистру, который позволяет грубую регулировку циклов шины PCI. Поле предназначено для чтения и записи. Формат регистра показан в табл. 12.8. Следует помнить, что не каждое устройство поддерживает данное поле.

Таблица 12.8. *Формат командного регистра (Command)*

Биты	**Описание**
0	Управляет возможностью устройства принимать и посылать данные в пространстве PCI (1 — включить, 0 — выключить)
1	Управляет возможностью устройства работать с памятью (1 — включить, 0 — выключить)
2	Управляет возможностью устройства работать в качестве ведущего на шине PCI (1 — включить, 0 — выключить)
3	Поддержка устройством специальных циклов шины (1 — включить контроль, 0 — игнорировать)
4	Управление сигналом для записи в память (1 — включить, 0 — выключить)
5	Управление доступом к регистрам палитры для VGA-совместимых и других графических устройств (1 — разрешить доступ к палитре, 0 — запретить доступ к палитре)

Таблица 12.8 *(окончание)*

Биты	Описание
6	Поддержка устройством ошибок четности (1 — включить, 0 — выключить)
7	Резерв
8	Резерв
9	Возможность транзакций для устройства (1 — включить, 0 — выключить)
10—15	Резерв

- Status — является необязательным полем и позволяет получить информацию о состоянии шины PCI. Поле предназначено для чтения и записи. Формат регистра показан в табл. 12.9.

Таблица 12.9. *Формат регистра состояния (Status)*

Биты	Описание
0—4	Резерв
5	Только для чтения. Указывает на поддержку устройством частоты шины 66 МГц (1 — поддерживает 66 МГц, 0 — только 33 МГц)
6	Только для чтения. Указывает на поддержку устройством пользовательских настроек конфигурации (1 — да, 0 — нет)
7	Только для чтения. Указывает на поддержку устройством транзакций (1 — да, 0 — нет)
8	Резерв
9—10	Определяют типы синхронизации для устройства
11	Бит устанавливается в 1 при аварийном завершении транзакции для целевого устройства
12	Бит устанавливается в 1 ведущим устройством при аварийном завершении транзакции
13	Бит устанавливается в 1 ведущим устройством при аварийном завершении транзакции
14	Резерв
15	Бит устанавливается в 1 при обнаружении устройством ошибок четности

- Cache Line Size — поле является необязательным и доступно, если устройство поддерживает запись в память; предназначено для чтения и записи.
- Latency Timer — поле является необязательным и определяет время ожидания для ведущего устройства на шине PCI; предназначено для чтения и записи.

- BIST — поле является необязательным и управляет режимом самотестирования для устройства. Если устройство не поддерживает самодиагностику, то значение поля будет равно 0. Бит 7 указывает на поддержку режима самотестирования (1 — есть). Запись в бит 6 единицы позволит выполнить самотестирование указанного устройства. Биты 0—3 определяют результат операции (0 — ошибка). Поле предназначено для чтения и записи.
- Base Address Registers — поле служит для записи всех доступных адресов ввода-вывода для работы с устройством.
- Cardbus CIS Pointer — поле используется для специфических устройств связи. Подробнее можно узнать из спецификации устройств PCMCIA.
- Совместимость — поле определяет некоторые особенности шины. Некоторые из возможных значений представлены в табл. 12.10.

Таблица 12.10. *Формат байта совместимости*

Код	Описание
00h	Резерв
01h	Поддержка функций управления питанием
02h	Совместимость с графической шиной AGP
03h	Поддержка дополнительного описателя особенностей
04h	Поддержка идентификатора слотов моста для внешних подключений
05h	Поддержка сообщений, сигнализирующих о прерываниях
06h	Поддержка кэшируемого обмена данными
07h	Поддержка устройств PCI-X
08h	Зарезервировано для AMD
09h	Поддержка настроек, определяемых производителем устройства
0Ah	Поддержка порта для отладки
0Bh	Поддержка централизованного управления ресурсами
0Ch	Поддержка "горячего" подключения
0Dh—0Fh	Зарезервированы
10h	Поддержка настроек для шины PCI-X
11h—FFh	Зарезервированы

- Interrupt Line — поле позволяет получить номер выделенной для устройства линии прерывания; предназначено для чтения и записи.

- Min_Gnt и Max_Lat — определяют время ожидания и параметры его настройки. Время ожидания измеряется в микросекундах; предназначено только для чтения.

Остальные поля необязательны и могут отличаться для различных устройств.

12.2. Использование портов

Представленный здесь материал основывается на устройствах фирмы Intel. Любую дополнительную информацию по данной теме можно свободно найти на сайте этого производителя (**www.intel.com**). Как правило, для доступа к шине PCI используются адреса 0CF8h–0CFFh. За определенными адресами закреплены аппаратные регистры, позволяющие получить доступ к разнообразным устройствам на шине PCI. Рассмотрим их подробнее.

12.2.1. Регистр конфигурации адреса

Расположен по адресу 0CF8h. Этот 32-разрядный регистр доступен для чтения и записи. Позволяет определить конфигурационные параметры устройства на шине PCI, а также номер регистра для последующего доступа к устройству. Формат этого регистра показан в табл. 12.11. По умолчанию значение регистра равно 00000000h.

Таблица 12.11. *Формат регистра конфигурации адреса*

Бит	Описание
0—1	Резерв
2—7	Определяют значение регистра для доступа к устройству
8—10	Номер функции для многофункционального устройства
11—15	Номер устройства на шине PCI
16—23	Номер шины PCI
24—30	Резерв
31	Установка этого бита в 1 разрешает доступ к конфигурационному пространству шины PCI, иначе доступ заблокирован (бит равен 0)

12.2.2. Регистр конфигурации данных

Регистр расположен по адресу 0CFCh. Этот 32-разрядный регистр доступен для чтения и записи. Предназначен для чтения или записи данных в конфигурационное пространство шины PCI. Для обмена данными используются все биты (0—31).

С помощью этих двух регистров можно получить доступ к любому устройству на шине. Для этого в регистр адреса следует записать параметры устройства и после этого можно читать и писать данные через регистр данных. Рассмотрим пример кода для сканирования шины PCI, представленный в листинге 12.1.

Листинг 12.1. Сканирование шины PCI

```
#include "stdafx.h"
// определяем значения смещений для пространства шины PCI
#define VENDOR_ID          0x00
#define DEVICE_ID          0x02
#define CODE               0x04
#define STATUS             0x06
#define REVISION_ID        0x08
#define INTERFACE          0x09
#define SUBCLASS           0x0A
#define CLASSCODE          0x0B
#define CACHE_LINE_SIZE    0x0C
#define LATENCY_TIMER       0x0D
#define HEADER_TYPE        0x0E
#define BIST               0x0F
#define BASE_ADDRESS_0     0x10
#define BASE_ADDRESS_1     0x14
#define BASE_ADDRESS_2     0x18
#define BASE_ADDRESS_3     0x1C
#define BASE_ADDRESS_4     0x20
#define BASE_ADDRESS_5     0x24
#define CARDBUS_POINTER    0x28
#define SUBVEN_ID          0x2C
#define SUBSYSTEM_ID        0x2E
#define ROM_BASE_ADDRESS    0x30
#define INTERRUPT_LINE      0x3C
#define INTERRUPT_PIN       0x3D
#define MIN_GNT            0x3E
#define MAX_LAT            0x3F
// функция для сканирования шины PCI
void ScanPCI ( int iBusPCI );
// определение номера слота для устройства
int GetDeviceSlot ( int iDevice, int iFunction );
// получаем конфигурационные параметры для устройства
DWORD GetDevice ( int iBusPCI, int iSlot, int iAddress );
```

```
// реализуем наши функции
int GetDeviceSlot ( int iDevice, int iFunction )
{
   return ( ( ( ( iDevice ) & 0x1f ) << 3 ) | ( ( iFunction ) & 0x07 ) );
}
DWORD GetDevice ( int iBusPCI, int iSlot, int iAddress )
{
   return ( 0x80000000L | ( ( iBus & 0xff ) << 16 ) | ( iSlot << 8 ) |
          ( iAddress & ~3 ) );
}
// единственный аргумент функции определяет номер шины ( от 0 до 255 )
void ScanPCI ( int iBusPCI )
{
   DWORD dwResult = 0;
   BYTE buffer[256];
   // создаем двойной цикл для перебора всех устройств
   for ( int iDevice = 0; iDevice < 32; iDevice++ )
   {
      for ( int iFunction = 0; iFunction < 8; iFunction++ )
      {
        memset ( &buffer, 0, 256 );
        // вычисляем номер очередного слота
        int iSlot = GetNumSlot ( iDevice, iFunction );
        // проверяем поле Vendor ID для определения наличия устройства
        DWORD dwVendorID = 0;
        // получаем конфигурацию устройства
        dwResult = GetDevice ( iBusPCI, iSlot, VENDOR_ID );
        // пишем в адресный порт параметры устройства
        outPort ( 0xCF8, dwResult, 4 );
        dwResult = 0;
        // читаем из порта данных идентификатор производителя
        inPort ( 0xCFC, &dwResult, 4 );
        // если полученное значение равно 0 или 0xFFFFFFFF, то
        // выходим из вложенного цикла и продолжаем поиск
        if (  dwVendorID == 0x00000000 || dwVendorID == 0xFFFFFFFF )
            break;
        // если устройство присутствует, читаем его параметры
        // из конфигурационного пространства шины PCI
        for ( int j = 1; j < 256; j++ )
        {
          // получаем конфигурацию устройства
          dwResult = GetDevice ( iBusPCI, iSlot, j );
          // пишем в адресный порт параметры устройства
          outPort ( 0xCF8, dwResult, 4 );
```

```
          // получаем из порта очередной байт
          inPort ( 0xCFC + ( j&0x03 ), &dwResult, 1 );
          // сохраняем полученный байт в буфер
          buffer[j] = dwResult;
          // здесь мы можем извлечь нужные данные из буфера и сохранить
          // их для последующего использования
          // например, получим номер прерывания для устройства
          // переменная uINT определена где-то ранее
          uINT = buffer[INTERRUPT_LINE];
          // поле Header Type
          uHeader = buffer[HEADER_TYPE];
          // поле Revision ID
          uRevID = buffer[REVISION_ID];
          // поле Device ID
          dwDeviceID = ( ( WORD ) ( ( ( BYTE ) ( buffer[PCI_DEVICE_ID] ))
  | ( ( ( WORD ) ( ( BYTE ) ( buffer[PCI_DEVICE_ID] + 1 ) ) ) << 8 ) ) );
          // поле Code
          dwCode = ( ( WORD ) ( ( ( BYTE ) ( buffer[CODE] ) )
          | ( ( ( WORD ) ( ( BYTE ) ( buffer[CODE] + 1 ) ) ) << 8 ) ) );
          // поле Subsystem Vendor ID
          dwSubVendorID = ( ( WORD ) ( ( ( BYTE ) ( buffer[SUBVEN_ID] ) )
     | ( ( ( WORD ) ( ( BYTE ) ( buffer[SUBVEN_ID] + 1 ) ) ) << 8 ) ) );
          // базовый адрес 0
          WORD low = 0, high = 0;
          // получаем младшее слово адреса
          low = ( ( WORD ) ( ( ( BYTE ) ( buffer[BASE_ADDRESS_0] ) )
| ( ( ( WORD ) ( ( BYTE ) ( buffer[BASE_ADDRESS_0] + 1 ) ) ) << 8 ) ) );
           // получаем старшее слово адреса
          high = ( ( WORD ) ( ( ( BYTE ) ( buffer[BASE_ADDRESS_0] + 2 ) )
| ( ( ( WORD ) ( ( BYTE ) ( buffer[BASE_ADDRESS_0] + 3 ) ) ) << 8 ) ) );
          // вычисляем полный адрес
          dwBaseAddress_0 = ( ( LONG ) ( ( ( WORD ) ( low ) ) |
                      ( ( ( DWORD ) ( ( WORD ) ( high ) ) ) << 16 ) ) );
        }
      }
   }
}
// пример вызова функции ScanPCI для первой шины
ScanPCI ( 0 );
// пример вызова функции ScanPCI для второй шины
ScanPCI ( 1 );
```

Итак, наша функция `ScanPCI` имеет всего один аргумент, в качестве которого служит номер шины PCI. Из примера видно, что, работая всего с двумя пор-

тами 0CF8h и 0CFCh, мы получили возможность доступа ко всем устройствам компьютера. Кроме общих сведений о каждом устройстве, нам удалось определить базовые адреса ввода-вывода и номер прерывания. Однако следует заметить, что далеко не все устройства возвращают номер прерывания. Связано это с тем, что наряду с привычными модулями (видеоконтроллер, IDE или звуковой контроллер) возвращаются данные о специфических устройствах, типа мостов, USB-контроллеров и других, использующих иные принципы работы.

Фирмы Intel и Via на своих сайтах предоставляют самую свежую информацию о новых устройствах и *наборах микросхем*, или иначе *чипсетах* (*chipset*). В этой документации содержится подробная информация о доступе и управлении устройствами посредством конфигурационного пространства шины PCI. Поскольку каждый новый набор микросхем имеет свои особенности, рекомендую читателям постоянно следить за этой информацией.

Давайте в качестве примера рассмотрим конфигурационное пространство шины PCI для современного контроллера IDE фирмы Intel. Формат контроллера (256 байт) представлен в табл. 12.12.

Таблица 12.12. *Контроллер IDE фирмы Intel*

Смещение	Описание
00h–01h	Vendor ID
02h–03h	Device ID
04h–05h	Командный регистр
06h–07h	Status
08h	Revision ID
09h	Интерфейс
0Ah	Код подкласса
0Bh	Код базового класса
0Ch	Резерв
0Dh	Latency Timer
0Eh	Header Type
0Fh	Резерв
10h–13h	Базовый адрес для первого канала
14h–17h	Базовый адрес для первого канала
18–1Bh	Базовый адрес для второго канала
1Ch–1Fh	Базовый адрес для второго канала

Таблица 12.12 *(окончание)*

Смещение	Описание
20h–23h	Адрес регистра для шины IDE
24h–27h	Расширения
28h–2Bh	Резерв
2Ch–2Dh	Subsystem Vendor ID
2Eh–2Fh	Subsystem ID
24h–3Bh	Резерв
3Ch	Interrupt Line
3Dh	Interrupt Pin
3Fh	Резерв
40h–41h	Синхронизация для первого канала
42h–43h	Синхронизация для второго канала
44h	Синхронизация для ведомого устройства на первом и втором каналах
45h–47h	Резерв
48h	Управляющий регистр Ultra DMA
49h	Резерв
4Ah–4Bh	Регистр синхронизации Ultra DMA
4Ch–53h	Резерв
54h–55h	Конфигурация ввода-вывода
56h–F7h	Резерв
F8h–FBh	Идентификатор производителя
FCh–FFh	Резерв

Описание табл. 12.12.

- 00h–01h — байты определяют идентификатор производителя; предназначены только для чтения. В нашем случае всегда будут равны значению 8086h.
- 02h–03h — байты определяют идентификатор изделия; предназначены только для чтения. Возможны следующие значения:
  - 2411h — контроллер ATA (82801AA);
  - 2421h — контроллер ATA (82801AB);
  - 244Ah — контроллер ATA для мобильных ПК (82801BA);

- 244Bh — контроллер ATA для высокопроизводительных ПК (82801BA);
- 248Ah — контроллер ATA для мобильных ПК (82801CA);
- 248Bh — контроллер ATA для высокопроизводительных ПК (82801CA);
- 24CAh — контроллер ATA для мобильных ПК (82801DB);
- 24CBh — контроллер ATA для высокопроизводительных ПК (82801CA);
- 24DBh — контроллер ATA для высокопроизводительных ПК (82801EB).

❒ 04h–05h — байты определяют управляющий регистр, который доступен для чтения и записи. Формат регистра показан в табл. 12.13.

Таблица 12.13. *Формат управляющего регистра*

Бит	Описание
0	Управление операциями ввода-вывода (1 — разрешить, 0 — запретить)
1	Управление доступом к памяти (1 — разрешить, 0 — запретить)
2	Управление шиной (1 — включить, 0 — выключить)
3—15	Зарезервированы и равны 0

❒ 06h–07h — байты определяют регистр состояния, который доступен для чтения и записи.

❒ 08h — байт содержит номер версии устройства; предназначен только для чтения.

❒ 09h — байт определяет интерфейсный регистр, который доступен для чтения и записи. Формат регистра показан в табл. 12.14.

Таблица 12.14. *Формат интерфейсного регистра*

Бит	Описание
0	Выбор режима для первичного канала (1 — использовать совместимый, 0 — использовать "родной" PCI)
1	Поддержка режима на первичном канале (1 — использовать совместимый и "родной", 0 — только совместимый)
2	Выбор режима для вторичного канала (1 — использовать совместимый, 0 — использовать "родной" PCI)
3	Поддержка режима на вторичном канале (1 — использовать совместимый и "родной", 0 — только совместимый)
4—6	Если бит 7 равен 1, используется "родной" режим шины PCI, иначе — совместимый
7	Управляет значением битов 4—6

- 0Ah — байт определяет код подкласса для устройства; предназначен только для чтения.
- 0Bh — байт определяет код базового класса для устройства; предназначен только для чтения.
- 0Dh — байт определяет время ожидания для ведущего устройства на шине PCI; доступен для чтения и записи.
- 0Eh — байт определяет многофункциональное устройство; предназначен только для чтения.
- 10h–13h, 14h-17h, 18-1Bh, 1Ch-1Fh и 20h-23h — байты определяют базовые адреса каналов; доступны для чтения и записи. Формат регистров показан в табл. 12.15.

Таблица 12.15. *Формат регистра базового адреса*

Бит	Описание
0	Тип ресурса (должен быть равен 1), только для чтения
1—3	Зарезервированы и должны быть установлены в 000h
4—31	Значение базового адреса ввода-вывода

- 24h–27h — байты определяют базовый адрес ввода-вывода в памяти; доступны для чтения и записи. Формат регистров показан в табл. 12.16.

Таблица 12.16. *Формат регистра адреса, расположенного в памяти*

Бит	Описание
0	Тип ресурса (должен быть равен 0), только для чтения
1–2	32-разрядный тип отображения в памяти (должен быть равен 0), только для чтения
3	0
4–31	Значение базового адреса ввода-вывода в памяти

- 2Ch–2Dh — байты определяют регистр дополнительного идентификатора производителя; который доступен для чтения и однократной записи (для повторной записи придется перезагрузить компьютер). По умолчанию значение этого регистра равно 0.
- 2Eh–2Fh — байты определяют идентификатор подсистемы; доступны для чтения и однократной записи (для повторной записи придется перезагрузить компьютер). По умолчанию значение этого регистра равно 0.

- 3Ch — байт определяет регистр номера выделенного прерывания; который доступен для чтения и однократной записи (для повторной записи придется перезагрузить компьютер). По умолчанию значение этого регистра равно 0.
- 3Dh — байт определяет регистр дополнительных опций линии прерывания для контроллера; который предназначен только для чтения. Используется в стандартном режиме работы шины PCI. По умолчанию значение этого регистра равно 0.
- 40h–41h и 42h–43h — байты управляют режимом синхронизации контроллера; доступны для чтения и записи. Формат соответствующих регистров показан в табл. 12.17.

Таблица 12.17. *Формат регистра синхронизации контроллера*

Бит	Описание
0	Использование ускоренной синхронизации для первого канала (1 — включить, 0 — выключить)
1	Использование сигнала IORDY для первого канала (1 — включить, 0 — выключить)
2	Использование упреждения для первого канала (1 — включить, 0 — выключить)
3	Использование синхронизации с DMA для первого канала (1 — включить, 0 — выключить)
4	Использование ускоренной синхронизации для второго канала (1 — включить, 0 — выключить)
5	Использование сигнала IORDY для второго канала (1 — включить, 0 — выключить)
6	Использование упреждения для второго канала (1 — включить, 0 — выключить)
7	Использование синхронизации с DMA для второго канала (1 — включить, 0 — выключить)
8—9	Время восстановления (00h — 4 такта, 01h — 3 такта, 10h — 2 такта, 11h — 1 такт)
10—11	Зарезервированы и должны быть установлены в 00h
12—13	Частота сигнала IORDY (00h — 5 тактов, 01h — 4 такта, 10h — 3 такта, 11h — 2 такта)
14	Управление регистром синхронизации для ведомого устройства (1 — включить, 0 — выключить)
15	Использование декодирования для IDE (1 — включить, 0 — выключить)

- 44h — байт соответствует регистру, который управляет параметрами синхронизации для обоих каналов; доступен для чтения и записи. Формат регистра показан в табл. 12.18.

Таблица 12.18. *Формат регистра 44h*

Бит	Описание
0—1	Время восстановления для первого ведомого канала (00h — 4 такта, 01h — 3 такта, 10h — 2 такта, 11h — 1 такт)
2—3	Частота сигнала IORDY для первого ведомого канала (00h — 5 тактов, 01h — 4 такта, 10h — 3 такта, 11h — 2 такта)
4—5	Время восстановления для второго ведомого канала (00h — 4 такта, 01h — 3 такта, 10h — 2 такта, 11h — 1 такт)
6—7	Частота сигнала IORDY для второго ведомого канала (00h — 5 тактов, 01h — 4 такта, 10h — 3 такта, 11h — 2 такта)

- 48h — байт соответствует регистру, который определяет параметры синхронизации для режима Ultra DMA. Регистр доступен для чтения и записи. Формат регистра показан в табл. 12.19.

Таблица 12.19. *Формат регистра 48h*

Бит	Описание
0	Управляет режимом Ultra DMA для первого устройства на первом канале (1 — включить, 0 — выключить)
1	Управляет режимом Ultra DMA для второго устройства на первом канале (1 — включить, 0 — выключить)
2	Управляет режимом Ultra DMA для первого устройства на втором канале (1 — включить, 0 — выключить)
3	Управляет режимом Ultra DMA для второго устройства на втором канале (1 — включить, 0 — выключить)
4–7	Зарезервированы и должны быть установлены в 0000h

- 4Ah–4Bh — байты определяют дополнительные параметры синхронизации для режима Ultra DMA; доступны для чтения и записи.

- 54h–55h — байты определяют регистр конфигурации контроллера; доступен для чтения и записи. Формат регистров показан в табл. 12.20.

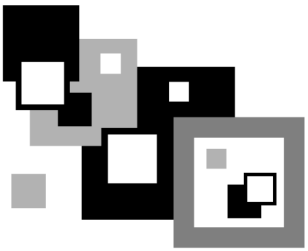
Таблица 12.20. *Формат регистра конфигурации контроллера*

Бит	Описание
0	Частота для первого диска на первом канале (1 — 66 МГц, 0 — 33 МГц)
1	Частота для второго диска на первом канале (1 — 66 МГц, 0 — 33 МГц)
2	Частота для первого диска на втором канале (1 — 66 МГц, 0 — 33 МГц)
3	Частота для второго диска на втором канале (1 — 66 МГц, 0 — 33 МГц)
4	Тип используемого кабеля для первого диска на первом канале (1 — 80 проводов, 0 — 40 проводов)
5	Тип используемого кабеля для второго диска на первом канале (1 — 80 проводов, 0 — 40 проводов)
6	Тип используемого кабеля для первого диска на втором канале (1 — 80 проводов, 0 — 40 проводов)
7	Тип используемого кабеля для второго диска на втором канале (1 — 80 проводов, 0 — 40 проводов)
8—9	Зарезервированы и должны быть установлены в 00h
10	Эффективность передачи данных для режима PIO (1 — включить, 0 — выключить)
11	Зарезервирован и должен быть установлен в 0h
12	Повышение частоты для первого диска на первом канале (1 — 100 МГц, 0 — 33 МГц или 66 МГц)
13	Повышение частоты для второго диска на первом канале (1 — 100 МГц, 0 — 33 МГц или 66 МГц)
14	Повышение частоты для первого диска на втором канале (1 — 100 МГц, 0 — 33 МГц или 66 МГц)
15	Повышение частоты для второго диска на втором канале (1 — 100 МГц, 0 — 33 МГц или 66 МГц)

- `F8h`—`FBh` — байты определяют дополнительный идентификатор производителя; доступен только для чтения.

Как видите, структура описания контроллера IDE полностью оговаривает все необходимые для работы с дисками параметры. Вам достаточно найти данное устройство на шине PCI (используя функцию `ScanPCI`) и сохранить базовые адреса и требуемую информацию. После этого можно спокойно работать с диском посредством набора команд ATA/ATAPI.

ГЛАВА 13



Контроллер DMA

Контроллер прямого доступа к памяти (*DMA* — Direct Memory Access) предназначен для обмена данными между оперативной памятью и устройством без помощи центрального процессора. Это значит, что во время операций чтения или записи с использованием каналов DMA процессор свободен и может обрабатывать другие данные, что ощутимо увеличивает общую производительность компьютера.

Контроллер DMA реализован на основе микросхемы 8237 (8237A). Она содержит четыре независимых канала. Число каналов может быть увеличено за счет каскадного подключения дополнительных микросхем 8237. Как правило, в компьютере имеется два контроллера: первый 8-разрядный (поддерживает каналы 0, 1, 2 и 3) и 16-разрядный (поддерживает каналы 4, 5, 6 и 7). Первый позволяет адресацию до 1 Мбайта памяти, а второй — до 16 Мбайт. Кроме того, первый канал поддерживает адресуемые блоки размером 64 Кбайт, а второй — 128 Кбайт, поэтому при передаче данных следует контролировать границы адресуемых участков. Контроллер DMA поддерживает четыре режима работы:

1. Одиночная передача данных.
2. Передача данных блоками.
3. Передача данных по внешнему запросу.
4. Каскадный режим работы.

Первый канал DMA (0) используется для регенерации обновления памяти и его не рекомендуется программировать. Второй канал (1) поддерживает работу с контроллером гибких дисков. Третий канал DMA (2) отведен для параллельного порта принтера (ECP). Четвертый выполняет роль связующего со вторым контроллером DMA посредством метода каскадирования. Для доступа к контроллеру применяются порты, перечисленные в табл. 13.1.

Таблица 13.1. *Список портов для работы с контроллером DMA*

Номер порта	Описание
00h	Регистр данных канала 0
01h	Регистр числа обработанных байтов канала 0
02h	Регистр данных канала 1
03h	Регистр числа обработанных байтов канала 1
04h	Регистр данных канала 2
05h	Регистр числа обработанных байтов канала 2
06h	Регистр данных канала 3
07h	Регистр числа обработанных байтов канала 3
08h	Управляющий регистр (запись) и регистр состояния (чтение) для DMA-1
09h	Регистр запроса для DMA-1
0Ah	Регистр маски для DMA-1
0Bh	Регистр режима работы DMA-1
0Ch	Регистр сброса триггера DMA-1
0Dh	Регистр сброса контроллера DMA-1
0Eh	Регистр сброса маскирующих битов DMA-1
0Fh	Регистр установки масок для всех каналов DMA-1
81h	Регистр адреса страницы канала 2
82h	Регистр адреса страницы канала 3
83h	Регистр адреса страницы канала 1
87h	Регистр адреса страницы канала 0
89h	Регистр адреса страницы канала 6
8Ah	Регистр адреса страницы канала 5
8Bh	Регистр адреса страницы канала 7
8Fh	Регистр обновления памяти
C0h	Регистр данных канала 4
C2h	Регистр числа обработанных байтов канала 4
C4h	Регистр данных канала 5
C6h	Регистр числа обработанных байтов канала 5
C8h	Регистр данных канала 6
CAh	Регистр числа обработанных байтов канала 6
CCh	Регистр данных канала 7

Таблица 13.1 *(окончание)*

Номер порта	Описание
CEh	Регистр числа обработанных байтов канала 7
D0h	Управляющий регистр (запись) и регистр состояния (чтение) для DMA-2
D2h	Регистр запроса для DMA-2
D4h	Регистр маски для DMA-2
D6h	Регистр режима работы DMA-2
D8h	Регистр сброса триггера DMA-2
DAh	Регистр сброса контроллера DMA-2
DCh	Регистр сброса маскирующих битов DMA-2
DEh	Регистр установки масок для всех каналов DMA-2

Передача данных через порты 00h—07h происходит в два этапа: первые 8 бит (0—7) и вторые 8 бит (8—15). Связано это с тем, что регистры 8-разрядные. Кроме того, эти порты доступны для чтения и записи. В режиме чтения они возвращают младший (00h, 02h, 04h и 06h) и старший (01h, 03h, 05h и 07h) байты для текущего адреса в соответствующем канале DMA-1. В режиме чтения они возвращают младший (00h, 02h, 04h и 06h) и старший (01h, 03h, 05h и 07h) байты для текущего адреса в соответствующем канале DMA-1.

Порты C0h—CEh доступны для чтения и записи. В режиме чтения они возвращают младший (C0h, C4h, C8h и CCh) и старший (C2h, C6h, CAh и CEh) байты для текущего адреса в соответствующем канале DMA-2.

Порты 08h (DMA-1) и D0h (DMA-2) в режиме записи используются как управляющие, а в режиме чтения возвращают текущее состояние. Размер обоих регистров составляет 8 бит. Они позволяют настроить работу всех четырех каналов. Регистры состояния также имеют размер 8 бит и помогают получить состояние каналов контроллера. Формат регистра управления показан в табл. 13.2, а регистра состояния — в табл. 13.3.

Таблица 13.2. *Формат управляющего регистра для DMA-1 и DMA-2*

Бит	Описание
0	Режим память-память (1 — включен, 0 — выключен)
1	Удержание канала 0 (1 — включено, 0 — выключено). Если бит 0 равен 0, то этот бит не используется
2	Управление контроллером (1 — выключен, 0 — включен)
3	Режим сжатия по времени (1 — включен, 0 — выключен). Если бит 0 равен 1, этот бит не используется

Таблица 13.2 *(окончание)*

Бит	Описание
4	Управление приоритетом (1 — с чередованием, 0 — фиксированный)
5	Цикл записи (1 — расширенный, 0 — с запаздыванием). Если бит 3 равен 1, то этот бит не используется
6	Сигнал DREQ (1 — выключен, 0 — включен)
7	Сигнал DACK (1 — выключен, 0 — включен)

Таблица 13.3. *Формат регистра состояния для DMA-1 и DMA-2*

Бит	Описание
0	Завершена работа DMA режима для канала 0 (4), если значение равно 1
1	Завершена работа DMA режима для канала 1 (5), если значение равно 1
2	Завершена работа DMA режима для канала 2 (6), если значение равно 1
3	Завершена работа DMA режима для канала 3 (7), если значение равно 1
4	Получен запрос для канала 0 (4), если значение равно 1
5	Получен запрос для канала 1 (5), если значение равно 1
6	Получен запрос для канала 2 (6), если значение равно 1
7	Получен запрос для канала 3 (7), если значение равно 1

Порты 09h (DMA-1) и D2h (DMA-2) используются только в режиме записи и позволяют установить сигнал запроса на указанном канале. Размер обоих регистров составляет 4 бита. Формат регистра запроса показан в табл. 13.4.

Таблица 13.4. *Формат регистра запроса для DMA-1 и DMA-2*

Бит	Описание
0—1	Выбор канала для сигнала запроса (00b — канал 0 или 4, 01b — канал 1 или 5, 10b — канал 2 или 6, 11b — канал 3 или 7)
2	Управление сигналом запроса (1 — установить, 0 — сбросить)
3—7	Игнорируются

Порты 0Ah (DMA-1) и D4h (DMA-2) используются только в режиме записи и позволяют установить бит маски на указанном канале для блокировки сигнала DREQ. Размер обоих регистров составляет 4 бита. Формат регистра маски показан в табл. 13.5.

Таблица 13.5. *Формат регистра маски для DMA-1 и DMA-2*

Бит	Описание
0—1	Выбор канала для сигнала запроса (00b — канал 0 или 4, 01b — канал 1 или 5, 10b — канал 2 или 6, 11b — канал 3 или 7)
2	Управление битом маски (1 — установить, 0 — сбросить)
3—7	Игнорируются

Порты 0Bh (DMA-1) и D6h (DMA-2) используются только в режиме записи и позволяют настроить режим работы контроллера для указанного канала. Номер канала устанавливается в самом регистре (биты 0 и 1). Размер обоих регистров составляет 6 бит. Формат регистра режима показан в табл. 13.6.

Таблица 13.6. *Формат регистра режима для DMA-1 и DMA-2*

Бит	Описание
0—1	Выбор канала для сигнала запроса (00b — канал 0 или 4, 01b — канал 1 или 5, 10b — канал 2 или 6, 11b — канал 3 или 7)
2—3	Режим передачи данных (00b — с проверкой, 01b — запись, 10b — чтение, 11b — недопустимый). Если биты 6 и 7 установлены в 1, данное поле неиспользуется
4	Автоинициализация (1 — включить, 0 — выключить)
5	Изменение значения адреса (1 — уменьшение, 0 — увеличение)
6–7	Выбор режима работы (00b — передача по запросу, 01b — одиночная передача, 10b — передача блоками, 11b — каскадный режим)

Порты 0Ch (DMA-1) и D8h (DMA-2) используются только в режиме записи и позволяют включить использование портов 00—07h для работы с 16-разрядными значениями. Вначале нужно будет прочитать младший байт, а затем старший.

Порты 0Dh (DMA-1) и DAh (DMA-2) позволяют выполнить сброс контроллера DMA. Для этого следует записать в эти порты любое значение.

Порты 0Eh (DMA-1) и DCh (DMA-2) позволяют выполнить сброс битов маски для всех каналов. Для этого следует записать в эти порты любое значение.

Порты 0Fh (DMA-1) и DEh (DMA-2) позволяют выполнить установки битов маски для всех каналов. Формат регистра установки маски показан в табл. 13.7.

Вот и все базовые сведения о контроллере DMA. Рассмотрим пример, позволяющий выполнить сброс контроллера DMA-2.

Таблица 13.7. *Формат регистра установки бита маски для всех каналов*

Бит	Описание
0	Бит маски для канала 0 (4) установлен, если значение равно 1, иначе сброшен
1	Бит маски для канала 1 (5) установлен, если значение равно 1, иначе сброшен
2	Бит маски для канала 2 (6) установлен, если значение равно 1, иначе сброшен
3	Бит маски для канала 3 (7) установлен, если значение равно 1, иначе сброшен
4—7	Зарезервированы и должны быть установлены в 0

Листинг 13.1. Сброс контроллера DMA

```
// пишем функцию для сброса контроллера DMA
void ResetDMA ( unsigned int uDMA )
{
   if (  ( uDMA == 0 ) || ( uDMA > 2 ) return;
   if ( uDMA == 1 ) // DMA-1
     outPort ( 0x0D, 0x01, 1 ); // выполняем сброс
   else // DMA-2
     outPort ( 0xDA, 0x01, 1 ); // выполняем сброс
}
// сбрасываем контроллер DMA-1
ResetDMA ( 1 );
```

Рассмотрим еще один пример для определения доступности канала 3 на контроллере DMA-1.

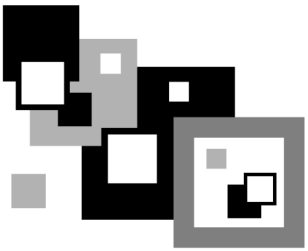
Листинг 13.2. Проверка готовности канала контроллера DMA-1

```
bool IsReadyDMA_1 ( unsigned int uChannel )
{
   DWORD dwResult = 0;
   if ( uChannel > 3 ) return;
   // получаем байт состояния
   inport ( 0x08, &dwResult, 1 ); // DMA-1
   // проверяем указанный канал
   switch ( uChannel )
    {
    case 0:
      if ( ( dwResult & 0x10 ) == 0x01 ) return true;
     break;
```

```
    case 1:
      if ( ( dwResult & 0x20 ) == 0x01 ) return true;
     break;
    case 2:
      if ( ( dwResult & 0x40 ) == 0x01 ) return true;
     break;
    case 3:
      if ( ( dwResult & 0x80 ) == 0x01 ) return true;
     break;
    }
   return false;
}
```

Вот и все, что мне хотелось рассказать о контроллере прямого доступа к памяти. Несомненно, его использование существенно увеличивает скорость обмена данными между устройством и компьютером, но следует помнить, что далеко не все даже современные устройства поддерживают такой режим работы. Кроме того, перед использованием контроллера необходимо убедиться в том, что выбранный канал не занят другим устройством и готов к работе.

ГЛАВА 14



Контроллер прерываний

Любой компьютер имеет в своем составе разнообразные внешние и внутренние устройства. Каждое из них так или иначе управляется центральным процессором, который в свою очередь должен успевать обрабатывать все поступающие данные. Поскольку процессор не может одновременно работать более чем с одним устройством, пришлось придумывать какие-то способы для разделения обработки данных, поступающих от конкретных устройств в отдельно взятой временной фазе. Вполне работоспособным показал себя *метод циклического опроса* каждого устройства с последующей обработкой данных, однако в результате получилась система с очень низкой производительностью, поскольку большая часть процессорного времени тратилась на банальное повторение одних и тех же операций (опрос, опознавание и последующий обмен данными). Наиболее приемлемым выходом из такой ситуации было бы использовать процессор только тогда, когда он будет нужен определенному устройству. В результате решения этой задачи появилось новое устройство, называемое контроллером прерываний (например, 8259 фирмы Intel). Общий принцип его работы достаточно прост: каждое устройство, когда возникает необходимость в использовании процессора посылает сигнал запроса в контроллер прерываний. В контроллере полученный запрос "рассматривается" и в зависимости от назначенного устройству приоритета посылается на выполнение центральному процессору. Процессор, получив запрос от контроллера прерываний, запоминает свое текущее состояние и переключается на выполнение запрошенной операции, по выполнении которой возвращается к решению сохраненной задачи. Такой режим работы называется прерыванием и позволяет достаточно эффективно обслуживать все имеющиеся в системе устройства.

Контроллер прерываний состоит из двух микросхем 8259 (в настоящее время они интегрируются в общий набор микросхем — *чипсет* (chipset)), подклю-

ченных по каскадной схеме. Каждая из них имеет 8 линий прерываний, закрепленных за определенным устройством. Первая обслуживает прерывания от IRQ0 до IRQ7, а вторая — от IRQ8 до IRQ15. Все прерывания имеют свой уровень приоритета (обслуживаются процессором в первую очередь). Самый высокий приоритет назначен прерыванию IRQ0, а самый низкий — IRQ15. В табл. 14.1 показано стандартное назначение каждого прерывания. В скобках после номера прерывания указан приоритет в числовом выражении.

Таблица 14.1. *Список аппаратных прерываний*

Номер прерывания	Назначенное устройство
IRQ0 (0)	Таймер
IRQ1 (1)	Клавиатура
IRQ2	Каскадное подключение второго контроллера
IRQ3 (10)	Последовательный порт COM2
IRQ4 (11)	Последовательный порт COM1
IRQ5 (12)	Параллельный порт LPT2
IRQ6 (13)	Контроллер флоппи-дисководов
IRQ7 (14)	Параллельный порт LPT1
IRQ8 (2)	Часы реального времени (CRT)
IRQ9 (3)	Свободен
IRQ10 (4)	Контроллер видеоадаптера
IRQ11 (5)	Свободно
IRQ12 (6)	Мышь PS/2
IRQ13 (7)	Математический сопроцессор
IRQ14 (8)	Первый контроллер жесткого диска
IRQ15 (9)	Второй контроллер жесткого диска

Чтобы получить доступ к контроллеру прерываний, следует использовать следующие порты ввода-вывода: 20h, 21h, A0h и A1h. Порты 20h и 21h обрабатывают прерывания IRQ0—IRQ7 для первого контроллера, а порты A0h и A1h — прерывания IRQ8—IRQ15 для второго контроллера прерываний.

Для организации работы контроллера прерываний используются специальные команды: управляющая команда инициализации (ICW — Initialization Command Word), управляющая команда операции (OCW — Operation

Command Word). Существуют четыре команды инициализации (от 1 до 4): `ICW1`, `ICW2`, `ICW3` и `ICW4`. Рассмотрим их подробнее.

14.1. Команда *ICW1*

Начальная команда инициализации контроллера прерываний. Для записи данной команды в регистры контроллеров (ведущего и ведомого) используются соответственно порты `20h` и `A0h`. Формат команды `ICW1` представлен в табл. 14.2.

Таблица 14.2. *Формат команды* `ICW1`

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	0	0	0	1	PT	Размер	Режим	ICW4

Приведем краткое описание таблицы.

- Бит 0 управляет использованием команды `ICW4`. Если бит установлен в 1, то команда будет вызвана.
- Бит 1 определяет использование ведомого контроллера (1 — не используется, 0 — используется).
- Бит 2 определяет размер вектора прерывания (1 — 4 байта, 0 — 8 байтов).
- Бит 3 определяет режим срабатывания триггера (1 — по фронту, 0 — по уровню).
- Бит 4 должен быть установлен в 1.
- Биты 5—7 должны быть установлены в 0.

14.2. Команда *ICW2*

Позволяет установить адрес вектора прерывания для `IRQ0` или `IRQ8`. Используются только биты с 3 по 7. Для первого контроллера нужно записать значение `08h`, а для второго — `70h`.

14.3. Команда *ICW3*

Команда имеет различное значение в зависимости от типа контроллера. Для ведущего устройства используются биты 0—7. Если значение равно 1, то, значит, подключен ведомый контроллер. Для ведомого устройства используются только биты 0—2. Они определяют номер выхода ведущего контроллера, к которому подключен ведомый и могут принимать одно из следующих

значений: 000b — выход 0, 001b — выход 1, 010b — выход 2, 011b — выход 3, 100b — выход 4, 101b — выход 5, 110b выход 6 и 111b — выход 7. Биты 3—7 должны быть установлены в 0.

14.4. Команда *ICW4*

Команда позволяет настроить дополнительные режимы работы. Формат регистра представлен в табл. 14.3.

Таблица 14.3. *Формат команды* `ICW4`

Биты	7	6	5	4	3	2	1	0
Описание	0	0	0	CP	Режим		Авто	CPU

Приведем краткий комментарий к таблице.

- Бит 0 определяет поддержку типа процессора (1 — 8086, 0 — 8085).
- Бит 1, установленный в 1, означает, что используется режим автоматического завершения прерывания EOI.
- Биты 2—3 определяют режим работы: 00b или 01b для не буферизированного режима, 10b — буферизация для ведомого контроллера, 11b — буферизация для ведущего контроллера.
- Бит 4 определяет специальный вложенный режим (1 — использовать, 0 — не использовать).
- Биты 5—7 зарезервированы и должны быть установлены в 0.

Кроме команд инициализации, существуют три команды управления: OCW1, OCW2 и OCW3. Они позволяют изменять параметры работы уже инициализированного контроллера прерываний.

14.5. Команда *OCW1*

Команда управляет маскированием различных прерываний. Установка бита в 1 позволяет запретить соответствующее прерывание, а сброс бита в 0 разрешает его. В табл. 14.4 показаны соотношения битов регистра и номеров прерываний для обоих контроллеров.

Прежде чем изменить значение маски прерываний, необходимо прочитать текущее значение из порта, а затем, установив нужный разряд с помощью операции ИЛИ, записать полученный результат обратно в порт.

Таблица 14.4. *Соотношения битов регистра и номеров прерываний*

Порт	Бит							
	7	6	5	4	3	2	1	0
21h	IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	IRQ0
A0h	IRQ15	IRQ14	IRQ13	IRQ12	IRQ11	IRQ10	IRQ9	IRQ8

14.6. Команда *OCW2*

Команда позволяет управлять использованием команды EOI (окончанием прерывания), а также изменяет приоритет выполнения конечной фазы прерывания. Формат команды представлен в табл. 14.5. Команда (сигнал) EOI необходима контроллеру прерываний для завершения операции обработки прерывания и сброса внутренних регистров для последующего использования, поэтому практически всегда после окончания работы с прерыванием необходимо записывать в регистр 20h (A0h) значение 20h.

Таблица 14.5. *Формат команды OCW2*

Биты	7	6	5	4	3	2	1	0
Описание	Использование EOI			0	0	Приоритет		

Приведем краткое описание таблицы.

- Биты 0—2 определяют номер линии (IRQ), которая будет использована, если бит 6 установлен в 1. Возможны следующие значения: 000b — выход 0, 001b — выход 1, 010b — выход 2, 011b — выход 3, 100b — выход 4, 101b — выход 5, 110b выход 6 и 111b — выход 7. Биты 3—7 должны быть установлены в 0.
- Биты 3—4 определяют код команды OCW2 (00b). Должны быть установлены в 0.
- Биты 5—7 определяют вариант использования команды окончания прерывания (EOI). Возможны следующие значения:
  - 000b — автоматический сдвиг приоритетов запрещен;
  - 100b — автоматический сдвиг приоритетов разрешен;
  - 001b — неспецифичная команда EOI;
  - 011b — специфичная команда EOI;
  - 101b — сдвиг приоритетов для неспецифичной команды EOI;
  - 111b — сдвиг приоритетов для специфичной команды EOI;

- 110b — использовать сдвиг приоритетов;
- 010b — нет никаких операций.

14.7. Команда *OCW3*

Команда позволяет прочитать текущее состояние контроллера прерываний. Формат команды представлен в табл. 14.6.

***Таблица 14.6.** Формат команды OCW3*

Биты	7	6	5	4	3	2	1	0
Описание	0	Маска		0	1	Опрос	Статус	

Приведем краткий комментарий к таблице.

- Биты 0—1 позволяют установить режим доступа к состоянию контроллера. Возможны следующие значения: 00b и 01b — не читать состояние, 10b — читать регистр состояния на следующем прерывании, 11b — читать регистр состояния.
- Бит 2 используется для метода поллинга (опроса). Если бит установлен в 1, режим поллинга используется.
- Биты 3—4 определяют код команды OCW2 (01b).
- Биты 5—6 определяют состояние специального режима маски (00b и 01b — не использовать, 10b — выключить специальный режим маски, 11b — включить специальный режим маски).
- Бит 7 зарезервирован и должен быть установлен в 0.

Вот и вся необходимая теория. А теперь рассмотрим практические примеры программирования контроллера прерываний. Сначала напишем код для запрещения прерывания IRQ15 (второй контроллер IDE) так, как показано в листинге 14.1.

Листинг 14.1. Запрещение прерывания IRQ15

```
// пишем функцию для запрещения прерывания от контроллера IDE
void DisableIRQ_IDE ( unsigned int uIRQ )
{
   DWORD dwResult = 0;
   if ( ( uIRQ < 14 ) && ( uIRQ > 15 ) ) return;
   if ( uIRQ == 14 ) // первый контроллер
   {
     // получаем текущее значение
     inPort ( 0xA0, &dwResult, 1 );
```

```
    dwResult |= 0x40; // запрещаем IRQ14
    // записываем обновленное значение в порт
    outPort ( 0xA0, dwResult, 1 );
  }
  else // второй контроллер
  {
    // получаем текущее значение
    inPort ( 0xA0, &dwResult, 1 );
    dwResult |= 0x80; // запрещаем IRQ15
    // записываем обновленное значение в порт
    outPort ( 0xA0, dwResult, 1 );
  }
}
```

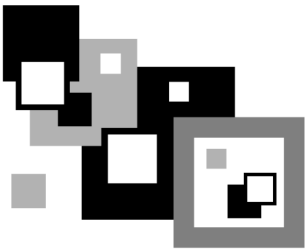
Между чтением и записью одноименного порта можно вставлять небольшую паузу. Попробуем запретить прерывания клавиатуры (листинг 14.2).

Листинг 14.2. Запрещение прерывания `IRQ1`

```
// пишем функцию для управления прерыванием от клавиатуры
void DisableIRQ_Kbr ( bool bEnable )
{
  DWORD dwResult = 0;
  if ( bEnable) // запретить прерывание
  {
    // получаем текущее значение
    inPort ( 0x21, &dwResult, 1 );
    dwResult |= 0x02; // запрещаем IRQ1
    Sleep ( 1 ); // добавляем маленькую задержку
    // записываем обновленное значение в порт
    outPort ( 0x21, dwResult, 1 );
  }
  else // разрешить прерывание
  {
    // получаем текущее значение
    inPort ( 0x21, &dwResult, 1 );
    dwResult &= 0xFD; // разрешаем IRQ1
    // записываем обновленное значение в порт
    outPort ( 0x21, dwResult, 1 );
  }
}
```

Вот и все, что может потребоваться в Windows для программирования контроллера прерываний.

ГЛАВА 15



Процессор

Думаю, нет смысла говорить о роли процессора в компьютерной системе. Эта небольшая по размерам микросхема является сердцем и мозгом компьютера, позволяя собрать в одно целое разнообразные типы устройств. В результате мы получаем мощный вычислительный комплекс, позволяющий решать не только и не столько научные задачи, но позволяющий нам окунуться в мир музыки или кино, игрушек и других всевозможных развлечений. Правда, на работе иногда приходится возиться с текстовыми процессорами и бухгалтерскими программами, рисовать однообразные логотипы и обрабатывать базы данных, но главное, что компьютер дает нам уникальную возможность самообразования и выражения своих способностей. И все это благодаря в первую очередь небольшой микросхеме, установленной внутри корпуса и практически бесшумно (если выкинуть медный кулер) выполняющей свой тяжкий труд.

Каждый программист должен иметь базовые сведения о центральном процессоре. К сожалению, прогресс так быстро создает новые и новые модификации, что угнаться за всем просто невозможно. В этой главе мы рассмотрим стандартные вопросы программирования процессора или иначе *CPU* (Central Processing Unit — центральный процессор).

Для получения информации о процессоре в языке ассемблера существует специальная команда `CPUID`. Она позволяет получить как версию процессора, так и поддерживаемые им возможности. Данную команду можно использовать для процессоров Intel (80486), AMD (80486DX4) и Cyrix M1. Прежде чем начать работу с этой командой, следует убедиться, можно ли ее использовать в данной системе. Для этого необходимо установить бит AC в регистре `EFLAGS`. Если операция пройдет успешно, значит, команда `CPUID` поддерживается. Пример кода для проверки команды `CPUID` представлен в листинге 15.1. Если команда `CPUID` не поддерживается вашим компилятором, то примените

вместо нее строку "db 0x0F, 0xA2". Кроме того, для проверки можно выполнить ту же последовательность действий, но для бита ID (21 бит в EFLAGS).

Листинг 15.1. Проверка поддержки процессором команды CPUID

```
// макрос для замены команды CPUID
#define CPUID _asm _emit 0x0F _asm _emit 0xA2
// пишем функцию проверки
bool IsCPUID ( )
{
   __int32 i32Result = 0;
   // блока кода на ассемблере
   __asm
    {
     pushfd
     pop EAX
     mov ECX, EAX
     xor EAX, 40000h ; (для ID 20000h)
     push EAX
     popfd
     pushfd
     pop EAX
     xor EAX, ECX
     je exit_proc
     mov i32Result, 1
     exit_proc:
    }
   if ( i32Result ) return true; // CPUID поддерживается
   // CPUID не поддерживается
   return false;
}
```

Теперь, когда мы убедились в поддержке команды CPUID, можно получить доступную информацию об установленном процессоре. Перед выполнением команды CPUID в регистр EAX следует записать определенное числовое значение, от которого будет зависеть возвращаемые данные. Например, для процессоров Intel формат данных показан в табл. 15.1.

Приведем краткое описание табл. 15.1.

- Если входное числовое значение равно 00h, в регистр EAX записывается максимальное значение, которое может использоваться (в EAX) с командой CPUID (в данном случае 2). Регистры EBX, ECX и EDX содержат составляющие имени производителя в ASCII-кодах (12 символов), причем первое значе-

Таблица 15.1. *Формат данных для процессора Intel*

Значение регистра EAX	Тип получаемой информации
00h	Регистр EAX: максимальное входное значение
	Регистр EBX: 756E6547h ("uneG")
	Регистр ECX: 49656E69h ("leni")
	Регистр EDX: 6C65746Eh ("letn")
01h	Регистр EAX: версия (модель, тип, семейство)
	Регистр EBX: зарезервирован
	Регистр ECX: зарезервирован
	Регистр EDX: информация о дополнительных свойствах
02h	Регистр EAX: информация о встроенном кэше
	Регистр EBX: информация о встроенном кэше
	Регистр ECX: информация о встроенном кэше
	Регистр EDX: информация о встроенном кэше

ние всегда расположено в младшем регистре (BL, CL и DL). Для процессоров фирмы AMD используется строка "AuthenticAMD", а для Cyrix — "CyrixInstead".

- Если числовое значение равно 01h, в регистр EAX записывается информация о версии процессора. Формат этого регистра показан в табл. 15.2. Регистры EBX ECX и зарезервированы, и не используются. В регистр EDX записывается информация о дополнительных свойствах процессора. Формат этого регистра представлен в табл. 15.3.
- Входное числовое значение 02h позволяет получить информацию о кэшах. В табл. 15.4 перечислены некоторые значения для определения размера внутреннего кэша процессора. При этом в регистр EAX записывается количество вызовов CPUID (02h), необходимое для получения всей информации.

Таблица 15.2. *Формат регистра с версией CPU*

Биты	31—14	13—12	11—8	7—4	3—0
Описание	Резерв	Тип	Family ID	Model ID	Stepping ID

Приведем некоторые пояснения к табл. 15.2.

- Биты 3—0 определяют модификацию процессора. Для Intel это значение будет больше 3, если установленный процессор выше 80486.
- Биты 7—4 определяют модель процессора. Значение этого поля зависит от соседнего поля Family ID и применяется совместно с ним, чтобы идентифицировать тип установленного процессора. Известные мне комбинации значений для процессоров Intel перечислены в табл. 15.5.
- Биты 11—8 определяют номер семейства процессора.
- Биты 13—12 для старых процессоров указывают один из следующих типов: `00b` — OEM, `01b` — Overdrive, `10b` — Dual.
- Биты 31—14 зарезервированы и не используются (должны быть равны 0).

Таблица 15.3. *Формат регистра свойств процессора*

Бит	Обозначение	Описание
0	FPU	Наличие математического сопроцессора
1	VME	Поддержка виртуального режима V86
2	DE	Поддержка точки останова для отладчика
3	PSE	Поддержка расширенных размеров страниц (до 4 Мбайт)
4	TSC	Поддержка команды `RDTSC` и `CR4`
5	MSR	Поддержка команд `RDMSR` и `WRMSR`
6	PAE	Поддержка расширенных адресов (более 32 бит)
7	MCE	Поддержка проверки машинных ошибок
8	CX8	Поддержка команды `CMPXCHG8B`
9	APIC	Поддержка встроенного контроллера прерываний APIC
10	—	Резерв
11	SEP	Поддержка быстрых системных вызовов (команды `SYSENTER` и `SYSEXIT`)
12	MTRR	Поддержка регистров диапазона памяти MTRR
13	PGE	Поддержка PGE в `CR4`
14	MCA	Поддержка проверки машинной архитектуры
15	CMOV	Поддержка расширенных команд `CMOV`
16	PAT	Поддержка таблицы атрибутов, позволяющей увеличить адресуемую память
17	PSE-36	Поддержка 36-разрядных расширенных страниц (страницы размером 4 Мбайт позволяют использовать физическую адресацию памяти свыше 4 Гбайт)

Таблица 15.3 *(окончание)*

Бит	Обозначение	Описание
18—22	—	Резерв
23	MMX	Поддержка технологии MMX
24	FXSR	Поддержка быстрых команд для чисел с плавающей точкой (FXSAVE и FXRSTOR)
25—31	—	Зарезервированы и должны быть равны 0

Таблица 15.4. *Значения второго кэша процессоров*

Код	Размер кэша, Кбайт
40h	Кэш отсутствует
41h	128
42h	256
43h	512
44h	1024
45h	2048
79h	128
7Ah	512
7Bh	1024
7Ch	2048
82h	256
83h	512
84h	1024
85h	2048

Таблица 15.5. *Возможные значения для полей Family ID и Model ID (Intel)*

Family ID	Model ID	Тип процессора
0000b	—	8086/8088
0001b	—	80186/80188
0010b	—	80286
0011b	0000b	Intel386 DX
	0010b	Intel386 SX (CX, EX)
	0100b	Intel386 SL
	1111b	Неизвестный тип

Таблица 15.5 *(продолжение)*

Family ID	Model ID	Тип процессора
0100b	0000b	Intel486 DX
	0001b	Intel486 DX
	0010b	Intel486 SX
	0011b	Intel487 SX (Intel486 DX)
	0100b	Intel486 SL
	0101b	IntelSX2
	0111b	IntelDX2
	1000b	IntelDX4
	1111b	Неизвестный тип
0101b	0001b	Pentium
	0010b	Pentium P54C
	0011b	Pentium overdrive
	0101b	IntelDX4 разогнанный до Pentium
	0100b	Intel Pentium MMX
	0111b	Intel Pentium (Mobile)
	1000b	Intel Pentium MMX (Mobile)
	1111b	Неизвестный тип
0110b	0000b	Intel Pentium Pro
	0001b	Intel Pentium Pro
	0011b	Intel Pentium II
	0101b	Intel Celeron (если кэш 2 равен 0x40 или 0x41)
	0101b	Intel Pentium II (если кэш 2 равен 0x43)
	0101b	Intel Pentium II Xeon (если кэш 2 равен 0x44 или 0x45)
	0110b	Intel Celeron A (если кэш 2 равен 0x40 или 0x41)
	0110b	Intel Pentium II (если кэш 2 равен 0x42 или 0x43)
	0110b	Intel Celeron Mobile (если кэш 2 равен 0x41 и Stepping ID = 10)
	0110b	Intel Pentium II Mobile (если кэш 2 равен 0x42 или 0x82 и Stepping ID = 10)
	0111b	Intel Pentium III (если кэш 2 не равен 0x44 и 0x45)
	0111b	Intel Pentium III Xeon (если кэш 2 равен 0x44 или 0x45)
	1000b	Intel Pentium III E Coppermine (если кэш 2 не равен 0x41)

Таблица 15.5 *(продолжение)*

Family ID	Model ID	Тип процессора
	1000b	Intel Celeron 2 (если кэш 2 равен 0x41)
	1010b	Intel Pentium III Xeon
	1111b	Неизвестный тип
1111b	0000b	Intel Pentium 4
	0001b	
	0010b	

Рассмотрим пример получения информации о процессоре (листинг 15.2).

Листинг 15.2. Получение стандартной информации о CPU

```
// макрос для замены команды CPUID
#define CPUID _asm _emit 0x0F _asm _emit 0xA2
// пишем функцию для получения стандартных сведений о процессоре
void GetCPU_Info ( )
{
   BYTE Vendor_Name[13];
   BYTE Stepping_ID;
   BYTE Model_ID;
   BYTE Family_ID;
   DWORD Feature_CPU;
   DWORD CacheCPU[4];
   unsigned int uCache_1 = 0, uCache_2 = 0;
   // получаем информацию о процессоре
   __asm
   {
     xor EAX, EAX; устанавливаем EAX в 0
     cpuid; вызываем команду CPUID
     ; сохраняем полученные данные
     mov dword ptr Vendor_Name, EBX
     mov dword ptr Vendor_Name[+ 4], EDX
     mov dword ptr Vendor_Name[+ 8], ECX
     ; получаем данные о свойствах и версии
     xor EAX, EAX ; устанавливаем EAX в 0
     inc EAX ; определяем значение 1
     cpuid ; вызываем команду CPUID
     mov Stepping_ID, AL ; копируем первый байт регистра EAX
     and Stepping_ID, 0Fh ; выделяем значение Stepping ID
```

```
    and AL, 0F0h ; восстанавливаем значение регистра
    shr AL, 4 ; сдвигаем на 4 разряда вправо
    mov Model_ID, AL ; получаем значение Model ID
    and EAX, 0FF0h ; выделяем биты 8-11
    shr EAX, 8 ; сдвигаем на 8 разрядов вправо
    and EAX, 0Fh; выделяем значение Family_ID
    ; получаем дополнительные свойства процессора
    mov Feature_CPU, ECX
    ; получаем размер встроенного кэша
    xor EAX, EAX ; устанавливаем EAX в 0
    mov EAX, 02h ; определяем значение 02h
    cpuid ; вызываем команду CPUID
    mov dword ptr [CacheCPU + 0], EAX
    mov dword ptr [CacheCPU + 4], EBX
    mov dword ptr [CacheCPU + 8], ECX
    mov dword ptr [CacheCPU + 12], EDX
  }
   // обрабатываем значение кэша для Intel
   DWORD cacheTemp = CacheCPU[3];
   BYTE cache_L1 = ( BYTE ) ( cacheTemp >> 8 ); // размер первого кэша
   BYTE cache_Tmp = ( BYTE ) ( cacheTemp >> 24 );
   BYTE cache_L2 = ( BYTE ) ( cacheTemp >> 0 ); // размер второго кэша
   // получаем размер первого кэша
   if ( ( cache_Tmp == 0x0A ) && ( cache_L1 == 0x06 ) )
     uCache_1 = 16; // 16 Кбайт
   else if ( ( cache_Tmp == 0x0C ) && ( cache_L1 == 0x08 ) )
     uCache_1 = 32; // 32 Кбайт
   // получаем размер второго кэша
   switch ( cache_L2 )
   {
    case 0x40: // нет второго кэша
     break;
    case 0x41: // 128 Кбайт
    case 0x79:
     uCache_2 = 128;
     break;
    case 0x42: // 256 Кбайт
    case 0x82:
     uCache_2 = 256;
     break;
    case 0x43: // 512 Кбайт
    case 0x7A:
    case 0x83:
```

```
      uCache_2 = 512;
      break;
     case 0x44: // 1024 Кбайт
     case 0x7B:
     case 0x84:
      uCache_2 = 1024;
      break;
     case 0x45: // 2048 Кбайт
     case 0x7C:
     case 0x85:
      uCache_2 = 2048;
      break;
    }
}
```

Кроме стандартных аргументов, процессор может поддерживать расширенные значения для команды `CPUID`, перечисленные в табл. 15.6. Эти значения следует использовать для получения данных о совместимых с Intel процессоров AMD и Cyrix.

Таблица 15.6. *Расширенные значения для команды* `CPUID`

Код	Описание
03h	Позволяет получить серийный номер процессора (начиная с Intel Pentium III), закодированный в регистрах ECX и EDX
80000000h	Максимальное расширенное значение, поддерживаемое процессором
80000001h	Возвращает версию и свойства для процессора
80000002h	Название процессора
80000003h	Название процессора
80000004h	Название процессора
80000005h	Возвращает информацию о размерах первого кэша
80000006h	Возвращает информацию о размерах второго кэша

В листинге 15.3 приводится пример использования расширенного значения команды `CPUID` для получения серийного номера процессора.

Листинг 15.3. Получение серийного номера процессора

```
// макрос для замены команды CPUID
#define CPUID _asm _emit 0x0F _asm _emit 0xA2
```

```
// выделяем переменную для хранения серийного номера
DWORD dwSerialNumber[3];
// пишем функцию
void GetSN_Intel ( )
{
   __asm
   {
     xor EAX, EAX ; обнуляем регистр
     cpuid ; вызываем команду CPUID
     mov dword ptr [dwSerialNumber + 0], EAX
     mov EAX, 03h ; получить номер CPU
     cpuid ; вызываем команду CPUID
     ; получаем серийный номер
     mov dword ptr [dwSerialNumber + 4], EDX
     mov dword ptr [dwSerialNumber + 8], ECX
   }
}
// сохраняем серийный номер в строку
char szSerial[30];
sprintf ( szSerial, "Intel Serial Number: %08lx-%08lx-%08lx",
          dwSerialNumber[0], dwSerialNumber[1], dwSerialNumber[2] );
```

И еще попробуем получить название процессора, используя расширенные значения команды `CPUID` (листинг 15.4).

Листинг 15.4. Получение названия установленного процессора

```
// макрос для замены команды CPUID
#define CPUID _asm _emit 0x0F _asm _emit 0xA2
// выделяем переменную для хранения серийного номера
char szCPU_Name[49];
szCPU_Name[48] = 0; // завершающий ноль
// пишем функцию
void GetCPU_Name ( )
{
   __asm
   {
     xor EAX, EAX; обнуляем регистр
     mov EAX, 80000002h
     cpuid ; вызываем команду CPUID
     mov dword ptr [CPU_Name + 0], EAX
     mov dword ptr [CPU_Name + 4], EBX
```

```
    mov dword ptr [CPU_Name + 8], ECX
    mov dword ptr [CPU_Name + 12], EDX
    mov EAX, 80000003h
    cpuid ; вызываем команду CPUID
    mov dword ptr [CPU_Name + 16], EAX
    mov dword ptr [CPU_Name + 20], EBX
    mov dword ptr [CPU_Name + 24], ECX
    mov dword ptr [CPU_Name + 28], EDX
    mov EAX, 80000004h
    cpuid ; вызываем команду CPUID
    mov dword ptr [CPU_Name + 32], EAX
    mov dword ptr [CPU_Name + 36], EBX
    mov dword ptr [CPU_Name + 40], ECX
    mov dword ptr [CPU_Name + 44], EDX
  }
}
```

Формат данных для процессоров AMD показан в табл. 15.7.

Таблица 15.7. *Формат данных для процессора AMD*

Значение регистра `EAX`	**Тип получаемой информации**
`00h`	Регистр `EAX`: максимальное входное значение
	Регистр `EBX`: `68747541h` ("htuA")
	Регистр `ECX`: `444D4163h` ("DMAc")
	Регистр `EDX`: `69746E65h` ("itne")
`01h`	Регистр `EAX`: версия (модель, тип, семейство)
	Регистр `EBX`: (марка, число потоков на канал)
	Регистр `ECX`: (свойства)
	Регистр `EDX`: информация о дополнительных свойствах

Формат регистра EAX для функции 01h представлен в табл. 15.8.

Таблица 15.8. *Формат регистра `EAX` для AMD*

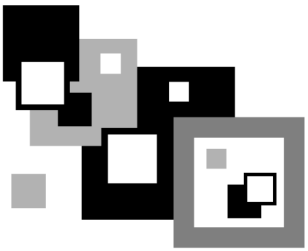
Биты	**31—28**	**27—20**	**19—16**	**15—12**	**11—8**	**7—4**	**3—0**
Описание	Резерв	Extended Family	Extended Model	Резерв	Base Family	Base Model	Stepping ID

Приведем некоторые пояснения к табл. 15.8.

- Биты 31—28 зарезервированы и не используются (должны быть равны 0).
- Биты 27—20 определяют расширенный номер семейства процессоров (Extended Family).
- Биты 19—16 определяют расширенный номер модели процессора (Extended Model).
- Биты 15—12 зарезервированы и должны быть равны 0.
- Биты 11—8 определяют базовый номер семейства процессоров (Base Family).
- Биты 7—4 определяют базовую модель процессора (Base Model).
- Биты 3—0 определяют модификацию процессора (Stepping ID).

Вот и все, что мне хотелось рассказать о программировании процессора.

ГЛАВА 16



Аппаратный мониторинг системы

В конце 90-х годов производители материнских плат стали активно использовать различные датчики температуры, питания и управления вентиляторами, благодаря чему появилась возможность контролировать критические компоненты системы в реальном времени. В настройки BIOS были добавлены дополнительные возможности мониторинга напряжений блока питания, температуры центрального процессора и внутреннего пространства корпуса, а также вывода информации о текущем состоянии вентиляторов (до трех). Важность этого трудно переоценить. Кроме контроля за состоянием системы, пользователь может самостоятельно устанавливать ограничения на температуру или питающие напряжения, что при умелом использовании помогает в ранней диагностике аппаратных сбоев оборудования. Например, установив максимальное значение температуры разогрева для процессора, можно защитить его от перегрева и возможного выхода из строя (особенно актуально для процессоров фирмы AMD). Контроль за вентилятором процессора позволяет вовремя выключить компьютер, если он вдруг заклинил или вовсе сгорел. А продвинутые пользователи, например, знают, что импульсный блок питания имеет довольно короткий срок эксплуатации (примерно 5 лет) и требует особого внимания. Необходимо постоянно (хотя бы раз в месяц) проверять значения питающих напряжений и при отклонении последних более чем на 10 % в ту или иную сторону, сразу же заменять блок питания целиком. Как правило, ремонту импульсные блоки питания не подлежат (не верьте никому, кто уверяет вас в обратном), и несоблюдение этих правил может привести к выходу из строя всех (!) компонентов системы. Но даже потеря "железа" не так важна, как потеря информации, хранящейся на компьютере.

Из всего сказанного следует, что программист должен знать и уметь работать с перечисленными выше возможностями для повышения не только безопасности компьютерной системы, но и для повышения своего профессионального мастерства.

В этой главе мы поговорим о том, как в современных компьютерах реализуется аппаратный мониторинг оборудования.

Для поддержки мониторинга системы, на материнскую плату добавляют самостоятельный модуль (микросхему) управления, позволяющий эффективно отслеживать напряжения питания, обороты вентиляторов и температуру. Существуют различные модули контроля оборудования, выпускаемые ведущими производителями, но здесь мы рассмотрим наиболее популярные микросхемы фирмы National Semiconductor. Изучив использование чипов данной фирмы, вы без особого труда сможете разобраться с остальными.

Итак, имеется несколько основных микросхем аппаратного мониторинга системы: LM78, LM79 и LM80. Они практически идентичны, различаясь в основном диапазоном и точностью измеряемых параметров. Мы рассмотрим только вторую и третью микросхемы, поскольку LM78 мало чем отличается от LM79. Основные характеристики этих микросхем представлены в табл. 16.1.

Таблица 16.1. *Основные характеристики микросхем LM79 и LM80*

Описание параметра	Поддерживаемое значение	
	LM79	**LM80**
Напряжение питания, В	5	2,8—5,75
Потребляемый ток (работа/простой)	1 мА/10 мкА	0,2 мА/15 мкА
Разрядность АЦП, бит	8	8
Максимальная погрешность измерения напряжения, %	± 1	± 1
Погрешность измерения температуры	Для диапазона от 10 до 100 ºС равна ± 3 ºС	Для диапазона от 25 ºС до 125 ºС равна ± 3 ºС
Точность измерения температуры, ºС	—	0,5

Микросхема LM79 работает с шинами ISA и Serial Bus и поддерживает следующие возможности:

- ❒ контроль температуры;
- ❒ управление тремя вентиляторами;
- ❒ пять положительных входных напряжений;
- ❒ два отрицательных входных напряжения;
- ❒ проверку на сравнение получаемых значений;
- ❒ дополнительный температурный датчик (например, LM75 или LM99);
- ❒ контроль вскрытия системного блока или изъятия какого-либо модуля.

Данная микросхема считывает обороты для первого и второго вентилятора в диапазоне от 1100 до 8800 оборот/мин, а для третьего фиксирует только среднее значение 4400.

Микросхема LM80 работает только с шиной Serial Bus и поддерживает такие же возможности, как и предыдущая.

Чтобы получить доступ к микросхеме мониторинга можно применить порт `290h`. Однако на несовместимой с Intel платформе номер порта может быть другим (следует обратиться на сайт производителя материнской платы). Есть еще один нюанс, о котором следует знать. Официально в документации Intel указан порт номер `80h`, хотя реально работает `290h`. Для определения порта шины Serial Bus (если стоит микросхема отличная от LM78 и LM79) необходимо применить процедуру поиска на шине PCI (см. *главу 10*) или попытаться обратиться через порты `90h` (базовый регистр адреса) и `D2h` (управляющий регистр).

Микросхемы мониторинга используют для управления и настройки определенные внутренние регистры, представленные в табл. 16.2. В скобках указаны адреса регистров для LM80.

Таблица 16.2. *Внутренние регистры управления*

Адрес регистра	Значение по умолчанию	Название	Описание
`40h` (`00h`)	`00001000b`	Регистр конфигурации	Позволяет настроить параметры работы
`41h` (`01h`)	`00000000b`	Регистр состояния 1	Отслеживают превышения допустимых заданных пределов или события прерываний (после чтения первого регистра будет автоматически выбран второй)
`42h` (`02h`)	`00000000b`	Регистр состояния 2	
`43h` (`03h`)	`00000000b`	Регистр маски 1 (SMI)	Позволяют маскировать различные источники прерываний (после чтения первого регистра будет автоматически выбран второй). Регистры SMI (System Management Interrupt) управляют системными прерываниями, а регистры NMI (Non-Maskable Interrupt) — не маскируемыми прерываниями
`44h` (`04h`)	`00000000b`	Регистр маски 2 (SMI)	
`45h` (—)	`00000000b`	Регистр маски 1 (NMI)	
`46h` (—)	`01000000b`	Регистр маски 2 (NMI)	
`47h` (`05h`)	`0101xxxxb` (`00010100b`)	Регистр делителя	Позволяет читать значение делителя для установки оборотов первого и второго вентиляторов (младшие 4 бита хранят состояние делителя, а старшие 4 бита позволяют установить новое значение). Для LM80 хранит одно значение делителя, равное 2 (4 400).

Таблица 16.2 *(окончание)*

Адрес регистра	Значение по умолчанию	Название	Описание
48h (—)	00101101b	Регистр адреса шины Serial Bus	Содержит адрес шины Serial Bus (по умолчанию 2Dh) и может быть изменен
49h (—)	1100000xb	Регистр сброса	Позволяет выполнить сброс микросхемы, установив для регистров значения по умолчанию
06h (LM80)	—	Регистр температуры	Используется только в LM80 и управляет представлением значений температуры (если бит 3 равен 1, используется 12-разрядное представление температуры и 4 младших бита значения расположены в битах 4–7)
20h—3Fh	—	Регистры данных	Позволяют получить или установить все поддерживаемые параметры

После подачи питания на микросхемы начинает работать непрерывный цикл измерения параметров системы с частотой один раз в секунду. Если микросхема содержит настройки для предельно-допустимых ограничений, происходит сравнение полученных и заданных значений. Если полученное значение превышает заданное, микросхема устанавливает сигнал прерывания в соответствующий регистр состояния. С помощью регистров маски устанавливается генерация тех или иных прерываний во время мониторинга оборудования, а регистр конфигурации позволяет заблокировать те или иные прерывания.

Поскольку микросхема LM79 поддерживает шину ISA, имеются еще четыре внешних регистра, необходимых для управления всеми внутренними регистрами LM79 (табл. 16.3).

Таблица 16.3. *Внешние регистры управления для LM79*

Адрес	Описание
x0h	Самотестирование микросхемы
x4h	Самотестирование микросхемы
x5h	Регистр адреса для выбора внутреннего регистра (чтение и запись)
x6h	Регистр данных (чтение и запись)

Работа с LM79 для ISA организуется следующим образом:

1. В регистр x5h записывается номер внутреннего регистра (см. табл. 16.2).
2. После этого можно читать или писать данные в регистр x6h.

Если на компьютере присутствуют обе шины (ISA и Serial Bus), следует вначале работы прочитать регистр `x5h`. Пока бит 7 равен 0, можно работать с шиной ISA, иначе используется шина Serial Bus. Переход на шину ISA требует примерно 10 мкс времени, а обратно всего 1 мкс. Бит 7 можно только читать.

Работа с LM79 для Serial Bus организуется следующим образом:

1. В регистр `x5h` записывается адрес шины Serial Bus (`48h`).
2. В этот же регистр пишется номер внутреннего регистра микросхемы (см. табл. 16.2).
3. В этот же регистр пишется байт данных.

Чтение данных происходит так:

1. В регистр `x5h` записывается адрес шины Serial Bus (`48h`). Этот пункт можно пропустить, если порт был уже выбран предыдущей операцией.
2. Из этого же регистра читается номер внутреннего регистра микросхемы (см. табл. 16.2).
3. Из этого же регистра читается байт данных.

Как уже говорилось ранее, адрес регистра для шины Serial Bus можно изменить, записав новое значение во внутренний регистр `48h`.

Регистр конфигурации (`40h`) управляет сбросом устройства и блокированием прерываний. Установка бита 0 в 0 позволяет выполнить сброс микросхемы и перевести ее в энергосберегающий режим. Установка бита 1 в 1 выполняет инициализацию регистра. Бит 2 управляет блокировкой немаскируемых прерываний (1 — разрешить NMI). Бит 3 управляет работой микросхемы (0 — включить мониторинг оборудования). При начале опроса параметров, бит 3 устанавливается в 0, а бит 0 в 1. Опрос параметров ведется в следующем порядке: температура, положительные напряжения питания (линии от 0 до 4), отрицательные напряжения питания (линии 5 и 6), первый, второй и третий вентиляторы. Из-за задержки в работе микросхемы следует при чтении каждого последующего параметра делать паузу не менее 120 мс. Если за один раз вы читаете все параметры, перед следующей операцией чтения необходимо выполнить паузу не менее 1,5 сек.

Для управления вентиляторами используется внутренний генератор (22,5 кГц для выдачи сигнала за время одного полного оборота) и 8-разрядный счетчик (максимум 255 отсчетов), определяющий количество сигналов. Например, если значение счетчика равно 153, обороты вентилятора равны 4400. Управление оборотами сводится к установке значения делителя (1, 2, 4 и 8) в регистре делителя (`47h`). Это будет работать только для первого и второго вентилятора. Третий имеет неизменяемые значения: делитель равен 2, а обороты

4400. По умолчанию начальные значения делителя и числа оборотов равны 2 и 4400 соответственно. Для подсчета текущего значения счетчика используется следующая формула:

$$\text{Значение счетчика} = (1{,}35 \times 10^6) / (\text{число оборотов} \times \text{делитель}).$$

Значения температуры кодируются 8-битным значением (LM79), где младший бит определяет 1 °C. В табл. 16.4 представлены стандартные значения регистра температуры.

Таблица 16.4. *Стандартные значения для 8-разрядного представления температуры*

Значение регистра	Температура, °C
01111101b (7Dh)	+125
00011001b (19h)	+25
00000001b (01h)	+1,0
00000000b (00h)	+0
11111111b (FFh)	–1
11100111b (E7h)	–25
11001001b (C9h)	–55

Если температура превышает заданную (или ниже ее), то генерируется прерывание и остается активным, пока не будет прочитан регистр состояния 1 (41h).

В микросхеме LM80 значения температуры могут кодироваться дополнительно 9- (табл. 16.5) и 12-разрядным (табл. 16.6) двоичным значением, что позволяет повысить точность измерений. Для 9-разрядного представления температуры значение младшего бита равно 0,5 °C, а для 12-разрядного — 0,0625 °C.

Таблица 16.5. *Стандартные значения для 9-разрядного представления температуры*

Значение регистра	Температура, °C
011111010b (FAh)	+125
000110010b (32h)	+25
000000011b (03h)	+1,5
00000000b (00h)	+0

Таблица 16.5 *(окончание)*

Значение регистра	Температура, °C
111111111b (1FFh)	–0,5
111001110b (1CEh)	–25
110010010b (192h)	–55

Таблица 16.6. *Стандартные значения для 12-разрядного представления температуры*

Значение регистра	Температура, °C
011111000000b (7D0h)	+125
000110010000b (190h)	+25
000000010000b (010h)	+1,0
000000000001b (01h)	+0,0625
00000000b (00h)	+0
111111111111b (FFFh)	–0,0625
111111110000 (FF0h)	–1,0
111001110000 (E70h)	–25
110010010000b (C90h)	–55

Младшие 8 бит значения (для 12-битного представления) температуры можно прочитать из внутреннего регистра 28h. Остаток будет записан в регистр 06h (биты 4—7). При 9-битном значении бит 9 будет записан в бит 7 регистра температуры.

И последнее, что мы рассмотрим здесь, будет структура внешних и некоторых внутренних регистров управления. В табл. 16.7 и 16.8 показаны форматы регистров x5h и x6h для LM79.

Таблица 16.7. *Формат адресного регистра x5h для LM79*

Биты	Доступ	Описание
0—6	Запись и чтение	Содержит адрес внутреннего регистра (см. табл.16.2)
7	Только чтение	Определяет готовность устройства (если бит равен 1, то можно записывать данные в регистр x6h для шины Serial Bus, иначе следует писать или читать регистр x6h после завершения работы на шине Serial Bus)

Таблица 16.8. *Формат адресного регистра x6h для LM79*

Биты	Доступ	Описание
0—7	Запись и чтение	Данные, которые надо передать или получить

В табл. 16.9 показан формат регистра выбора адреса для LM80.

Таблица 16.9. *Формат адресного регистра для LM80*

Биты	Доступ	Описание
0—7	Запись и чтение	Содержит адрес внутреннего регистра (см. табл.16.2) от 00h до 06h и от 20h до 3Fh

В табл. 16.10 показан формат регистров конфигурации для LM79 и LM80. Они позволяют инициализировать устройство мониторинга и установить режим работы. Кроме того, прочитав значение регистра после инициализации (оно должно быть равно 08h), можно сделать вывод о наличии микросхемы мониторинга на данной системе.

Таблица 16.10. *Формат регистра конфигурации для LM79 и LM80*

Биты	Доступ	Описание
0	Запись и чтение	Управляет запуском мониторинга системы (1 — включить контроль, 0 — перевести в дежурный режим). Перед установкой бита в 1 следует записать желаемые параметры в регистры данных 20h—3Fh
1	Запись и чтение	Установка бита в 1 позволяет включить прерывания (для LM79 SMI-прерывания)
2	Запись и чтение	Выбор активной выходной линии для LM80 и включение прерывания NMI для LM79
3	Запись и чтение	Блокировать прерывания SMI и NMI для LM79 и IRQ для LM80 (1 — блокировать, 0 — разрешить)
4	Запись и чтение	Сброс устройства
5	Запись и чтение	Выбор NMI-прерываний (1 — NMI, 0 — IRQ)
6	Запись и чтение	Управляет питанием
7	Запись и чтение	Восстанавливает питание на шине и инициализирует устройство

В табл. 16.11 и 16.12 показан формат первых регистров состояния (41h или 01h) для LM79 и LM80. Регистры состояния позволяют отслеживать изменение значений на заданных линиях.

Таблица 16.11. *Формат первого регистра состояния для LM79*

Биты	Доступ	Описание
0	Только чтение	Линия 0. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
1	Только чтение	Линия 1. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
2	Только чтение	Линия 2. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
3	Только чтение	Линия 3. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
4	Только чтение	Температура. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
5	Только чтение	Установка бита в 1 указывает то, что произошло прерывание от датчика температуры (LM75)
6	Только чтение	Первый вентилятор. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
7	Только чтение	Второй вентилятор. Установка бита в 1 указывает на выход текущего значения из заданного диапазона

Таблица 16.12. *Формат первого регистра состояния для LM80*

Биты	Доступ	Описание
0	Только чтение	Линия 0. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
1	Только чтение	Линия 1. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
2	Только чтение	Линия 2. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
3	Только чтение	Линия 3. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
4	Только чтение	Линия 4. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
5	Только чтение	Линия 5. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
6	Только чтение	Линия 6. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
7	Только чтение	Установка бита в 1 указывает, что на вход поступил сигнал прерывания

В табл. 16.13 и 16.14 показан формат вторых регистров состояния (42h или 02h) для LM79 и LM80. Выполняют те же функции, что и первые регистры состояния.

Таблица 16.13. *Формат второго регистра состояния для LM79*

Биты	Доступ	Описание
0	Только чтение	Линия 4. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
1	Только чтение	Линия 5. Установка бита в 1 указывает на выход текущего значения из заданного диапазона (отрицательные напряжения)
2	Только чтение	Линия 6. Установка бита в 1 указывает на выход текущего значения из заданного диапазона (отрицательные напряжения)
3	Только чтение	Третий вентилятор. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
4	Только чтение	Установка бита в 1 указывает на вскрытие корпуса или изъятие из разъема заданного модуля
5	Только чтение	Переполнение FIFO (через порты x0h и x4h)
6	Только чтение	Дополнительный цифровой вход. Установка бита в 1 указывает, что сигнал прерывания NMI на входе отсутствует
7	Только чтение	Резерв

Таблица 16.14. *Формат второго регистра состояния для LM80*

Биты	Доступ	Описание
0	Только чтение	Температура. Установка бита в 1 указывает на выход текущего значения из заданного диапазона (режим установлен во втором регистре маски битом 6)
1	Только чтение	Температура. Установка бита в 1 указывает, что произошло прерывание на входе для дополнительного датчика температуры (например, LM75 или LM99)
2	Только чтение	Первый вентилятор. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
3	Только чтение	Второй вентилятор. Установка бита в 1 указывает на выход текущего значения из заданного диапазона
4	Только чтение	Установка бита в 1 указывает на вскрытие корпуса или изъятие из разъема заданного модуля
5	Только чтение	Температура. Установка бита в 1 указывает на выход текущего значения из заданного диапазона (режим установлен во втором регистре маски битом 7)

Таблица 16.14 *(окончание)*

Биты	Доступ	Описание
6	Только чтение	Резерв
7	Только чтение	Резерв

В табл. 16.15 и 16.16 показан формат первых регистров маски для SMI (43h или 03h) для LM79 и LM80. Позволяют отключить генерацию прерываний для заданной линии. Эти регистры удобно использовать для временного выключения мониторинга за отдельными линиями (например, температуры или напряжения питания), а также для получения текущего состояния блокировок.

Таблица 16.15. *Формат первого регистра маски для LM79*

Биты	Доступ	Описание
0	Запись и чтение	Линия 0. Установка бита в 1 блокирует прерывание для данной линии
1	Запись и чтение	Линия 1. Установка бита в 1 блокирует прерывание для данной линии
2	Запись и чтение	Линия 2. Установка бита в 1 блокирует прерывание для данной линии
3	Запись и чтение	Линия 3. Установка бита в 1 блокирует прерывание для данной линии
4	Запись и чтение	Температура. Установка бита в 1 блокирует прерывание для данной линии
5	Запись и чтение	Первый вентилятор. Установка бита в 1 блокирует прерывание для данной линии
6	Запись и чтение	Второй вентилятор. Установка бита в 1 блокирует прерывание для данной линии
7	Запись и чтение	Резерв

Таблица 16.16. *Формат первого регистра маски для LM80*

Биты	Доступ	Описание
0	Запись и чтение	Линия 0. Установка бита в 1 блокирует прерывание для данной линии
1	Запись и чтение	Линия 1. Установка бита в 1 блокирует прерывание для данной линии

Таблица 16.16 *(окончание)*

Биты	Доступ	Описание
2	Запись и чтение	Линия 2. Установка бита в 1 блокирует прерывание для данной линии
3	Запись и чтение	Линия 3. Установка бита в 1 блокирует прерывание для данной линии
4	Запись и чтение	Линия 4. Установка бита в 1 блокирует прерывание для данной линии
5	Запись и чтение	Линия 5. Установка бита в 1 блокирует прерывание для данной линии
6	Запись и чтение	Линия 6. Установка бита в 1 блокирует прерывание для данной линии
7	Запись и чтение	Установка бита в 1 блокирует прерывание на цифровом входе для любого подключенного устройства

В табл. 16.17 и 16.18 показан формат вторых регистров маски для SMI (`44h` или `04h`) для LM79 и LM80. Выполняют те же функции, что и первые два регистра маски.

Таблица 16.17. *Формат второго регистра маски для LM79*

Биты	Доступ	Описание
0	Запись и чтение	Линия 4. Установка бита в 1 блокирует прерывание для данной линии
1	Запись и чтение	Линия 5. Установка бита в 1 блокирует прерывание для данной линии (отрицательные напряжения)
2	Запись и чтение	Линия 6. Установка бита в 1 блокирует прерывание для данной линии (отрицательные напряжения)
3	Запись и чтение	Третий вентилятор. Установка бита в 1 блокирует прерывание для данной линии
4	Запись и чтение	Вскрытие корпуса или изъятие из разъема заданного модуля. Установка бита в 1 блокирует прерывание для данной линии
5	Запись и чтение	Переполнение FIFO. Установка бита в 1 блокирует прерывание для данной линии
6	Запись и чтение	Дополнительный цифровой вход. Установка бита в 1 блокирует прерывание для данной линии
7	Запись и чтение	Сброс. Установка этого бита в 1 позволяет сбросить настройки для регистра конфигурации

Таблица 16.18. *Формат второго регистра маски для LM80*

Биты	Доступ	Описание
0	Запись и чтение	Температура. Установка бита в 1 блокирует прерывание для данной линии
1	Запись и чтение	Температура. Установка бита в 1 блокирует прерывание на входной линии для дополнительного датчика температуры (например, LM75 или LM99)
2	Запись и чтение	Первый вентилятор. Установка бита в 1 блокирует прерывание для данной линии
3	Запись и чтение	Второй вентилятор. Установка бита в 1 блокирует прерывание для данной линии
4	Запись и чтение	Вскрытие корпуса или изъятие из разъема заданного модуля. Установка бита в 1 блокирует прерывание для данной линии
5	Запись и чтение	Температура (12-разрядное представление). Установка бита в 1 блокирует прерывание для данной линии
6	Запись и чтение	Режим прерывания по температуре. При установке в ноль прерывание генерируется, если нарушен заданный диапазон, а сигнал прерывания будет сброшен. Если значение температуры остается вне диапазона, прерывание будет выдано повторно. Если бит установлен в 1, то прерывание будет выдано только один раз при выходе значения температуры из заданного диапазона
7	Запись и чтение	То же самое, но для 12-разрядного представления температуры

В табл. 16.19 и 16.20 показан формат регистров маски NMI (`45h` и `46h`) для LM79. Выполняет те же функции, что и первый регистр маски SMI.

Таблица 16.19. *Формат первого регистра маски NMI для LM79*

Биты	Доступ	Описание
0	Запись и чтение	Линия 0. Установка бита в 1 блокирует прерывание для данной линии
1	Запись и чтение	Линия 1. Установка бита в 1 блокирует прерывание для данной линии
2	Запись и чтение	Линия 2. Установка бита в 1 блокирует прерывание для данной линии
3	Запись и чтение	Линия 3. Установка бита в 1 блокирует прерывание для данной линии
4	Запись и чтение	Температура. Установка бита в 1 блокирует прерывание для данной линии

Таблица 16.19 *(окончание)*

Биты	Доступ	Описание
5	Запись и чтение	Температура. Установка бита в 1 блокирует прерывание на входной линии для дополнительного датчика температуры (например, LM75 или LM99)
6	Запись и чтение	Первый вентилятор. Установка бита в 1 блокирует прерывание для данной линии
7	Запись и чтение	Второй вентилятор. Установка бита в 1 блокирует прерывание для данной линии

Таблица 16.20. *Формат второго регистра маски NMI для LM79*

Биты	Доступ	Описание
0	Запись и чтение	Линия 4. Установка бита в 1 блокирует прерывание для данной линии
1	Запись и чтение	Линия 5. Установка бита в 1 блокирует прерывание для данной линии (отрицательные напряжения)
2	Запись и чтение	Линия 6. Установка бита в 1 блокирует прерывание для данной линии (отрицательные напряжения)
3	Запись и чтение	Третий вентилятор. Установка бита в 1 блокирует прерывание для данной линии
4	Запись и чтение	Вскрытие корпуса или изъятие из разъема заданного модуля. Установка бита в 1 блокирует прерывание для данной линии
5	Запись и чтение	Переполнение FIFO. Установка бита в 1 блокирует прерывание для данной линии
6	Запись и чтение	Дополнительный цифровой вход. Установка бита в 1 блокирует прерывание для данной линии
7	Запись и чтение	Сброс. Установка этого бита в 1 на время более 20 мс позволяет сбросить блокировку для вскрытия корпуса

В табл. 16.21 показан формат регистра делителя (`47h`) для LM79. Позволяет установить значения делителя для управления оборотами вентиляторов.

Таблица 16.21. *Формат регистра делителя для LM79*

Биты	Доступ	Описание
0—3	Только чтение	Для некоторых процессоров Pentium сюда записывается напряжение питания (младшие 4 бита)
4—5	Запись и чтение	Первый вентилятор. Возможны следующие значения: `00b` — 1, `01b` — 2, `10b` — 4 и `11b` — 8

Таблица 16.21 *(окончание)*

Биты	Доступ	Описание
6—7	Запись и чтение	Второй вентилятор. Возможны следующие значения: 00b — 1, 01b — 2, 10b — 4 и 11b — 8

В табл. 16.22 показан формат регистра делителя (05h) для LM80. Регистр предназначен для настройки оборотов вентиляторов и установки 12-разрядного представления для температуры.

Таблица 16.22. *Формат регистра делителя для LM80*

Биты	Доступ	Описание
0	Запись и чтение	Выбор первого вентилятора. При установке бита в 1 позволяет выбрать уровень чувствительности вентилятора, а при записи 0 — прочитать значение делителя
1	Запись и чтение	Выбор второго вентилятора. При установке бита в 1 позволяет выбрать уровень чувствительности вентилятора, а при записи 0 — прочитать значение делителя
2—3	Запись и чтение	Установка делителя для первого вентилятора. Возможны следующие значения: 00b — 1, 01b — 2, 10b — 4 и 11b — 8
4—5	Запись и чтение	Установка делителя для второго вентилятора. Возможны следующие значения: 00b — 1, 01b — 2, 10b — 4 и 11b — 8
6	Запись и чтение	Выбор режима для 12-разрядного представления температуры (бит должен быть установлен в 1)
7	Запись и чтение	Резерв

В табл. 16.23 показан формат регистра Serial Bus (48h) для LM79. Позволяет указать адрес регистра на шине Serial Bus.

Таблица 16.23. *Формат регистра Serial Bus для LM79*

Биты	Доступ	Описание
0—6	Запись и чтение	Адрес регистра Serial Bus
7	Только чтение	Резерв

В табл. 16.24 показан формат регистра сброса (49h) для LM79. Позволяет выполнить сброс микросхемы. После включения питания в регистре будет записано значение 1100000xb.

В табл. 16.25 показан формат регистра температуры (06h) для LM80. Управляет чувствительностью измерения температуры. После включения питания в регистре записано значение 00000001b.

Таблица 16.24. *Формат регистра Serial Bus для LM79*

Биты	Доступ	Описание
0	Только чтение	Старший бит (5) для напряжения питания процессора
1—4	Только чтение	Резерв
5	Запись и чтение	Установка бита в 1 позволит выполнить сброс микросхемы
6	Только чтение	Резерв
7	Только чтение	Идентификатор LM79 (как и для LM78 равен 0)

Таблица 16.25. *Формат регистра температуры для LM80*

Биты	Доступ	Описание
0	Только чтение	Состояние режима 12-разрядного представления температуры
1	Запись и чтение	Полярность температуры (1 — выше нуля, 0 — ниже нуля)
2	Запись и чтение	При установке в 0 используется режим генерации одного прерывания при нарушении заданных границ, иначе (1) прерывание генерируется постоянно, пока текущее значение температуры выходит за установленные пределы
3	Запись и чтение	Установка режима обработки температуры (0 — 8-разрядные преобразования, 0 — 11-разрядные преобразования)
4—7	Запись и чтение	Сюда записываются 4 младших бита для 11-разрядного положительного значения температуры, а для 8-разрядных значений температуры в бит 7 записывается знак плюса

Теперь осталось рассмотреть регистры данных, которые позволяют считывать и устанавливать параметры всех линий мониторинга оборудования. Список и назначение регистров данных для LM79 показан в табл. 16.26, а для LM80 — в табл. 16.27. Линии 0—6 применяются для считывания напряжений блока питания компьютера. Обычно они распределены следующим образом: линия 0 — +2,5 В, линия 1 — +2,5 В, линия 2 — +3,3 В, линия 3 — +5 В, линия 4 — +12 В, линия 5 — –12 В и линия 6 — –5 В.

Таблица 16.26. *Список регистров данных для LM79*

Номер регистра	Назначение
20h	Чтение линии 0
21h	Чтение линии 1

Таблица 16.26 *(окончание)*

Номер регистра	Назначение
22h	Чтение линии 2
23h	Чтение линии 3
24h	Чтение линии 4
25h	Чтение линии 5 (отрицательное напряжение)
26h	Чтение линии 6 (отрицательное напряжение)
27h	Чтение температуры
28h	Чтение текущего значения счетчика первого вентилятора
29h	Чтение текущего значения счетчика второго вентилятора
2Ah	Чтение текущего значения счетчика третьего вентилятора
2Bh	Установка максимального значения для линии 0
2Ch	Установка минимального значения для линии 0
2Dh	Установка максимального значения для линии 1
2Eh	Установка минимального значения для линии 1
2Fh	Установка максимального значения для линии 2
30h	Установка минимального значения для линии 2
31h	Установка максимального значения для линии 3
32h	Установка минимального значения для линии 3
33h	Установка максимального значения для линии 4
34h	Установка минимального значения для линии 4
35h	Установка максимального значения для линии 5
36h	Установка минимального значения для линии 5
37h	Установка максимального значения для линии 6
38h	Установка минимального значения для линии 6
39h	Установка предельного значения температуры
3Ah	Установка минимального значения температуры
3Bh	Установка предельного значения счетчика первого вентилятора
3Ch	Установка предельного значения счетчика второго вентилятора
3Dh	Установка предельного значения счетчика третьего вентилятора
3Eh—3Fh	Резерв

Таблица 16.27. *Список регистров данных для LM80*

Номер регистра	Назначение
20h	Чтение линии 0
21h	Чтение линии 1
22h	Чтение линии 2
23h	Чтение линии 3
24h	Чтение линии 4
25h	Чтение линии 5 (отрицательное напряжение)
26h	Чтение линии 6 (отрицательное напряжение)
27h	Чтение температуры
28h	Чтение текущего значения счетчика первого вентилятора
29h	Чтение текущего значения счетчика второго вентилятора
2Ah	Установка максимального значения для линии 0
2Bh	Установка минимального значения для линии 0
2Ch	Установка максимального значения для линии 1
2Dh	Установка минимального значения для линии 1
2Eh	Установка максимального значения для линии 2
2Fh	Установка минимального значения для линии 2
30h	Установка максимального значения для линии 3
31h	Установка минимального значения для линии 3
32h	Установка максимального значения для линии 4
33h	Установка минимального значения для линии 4
34h	Установка максимального значения для линии 5
35h	Установка минимального значения для линии 5
36h	Установка максимального значения для линии 6
37h	Установка минимального значения для линии 6
38h	Установка предельного значения температуры (8-разрядное значение)
39h	Установка минимального значения температуры (8-разрядное значение)
3Ah	Установка предельного значения температуры (12-разрядное значение)
3Bh	Установка минимального значения температуры (12-разрядное значение)
3Ch	Установка предельного значения счетчика первого вентилятора
3Dh	Установка предельного значения счетчика второго вентилятора
3Eh—3Fh	Резерв

Внешний датчик температуры может быть выполнен (в нашем контексте) на базе микросхем LM75, LM83, LM87, LM90, LM92 или LM99. Поскольку их параметры (чувствительность и время срабатывания) различаются, я не стану здесь рассматривать каждый, вы самостоятельно можете получить нужную информацию в Интернете. Здесь же я расскажу только о температурном датчике LM75. Он представляет собой отдельную микросхему для измерения температуры в диапазоне от –55 ºС до +125 ºС. Микросхема содержит общий управляющий регистр (чтение и запись), через который можно выбирать дополнительные регистры для чтения или записи. Формат общего регистра показан в табл. 16.28.

Таблица 16.28. *Формат общего регистра для LM75*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	0	0	0	0	0	0	Выбор регистра	

Описание табл. 16.28.

- Биты 2—7 не используются и должны быть установлены в 0.
- Биты 0—1 определяют номер дополнительного регистра. Возможны следующие значения: `00b` — регистр температуры, `01b` — регистр конфигурации, `10b` — регистр предельного значения гистерезиса температуры, `11b` — регистр стандартного значения температуры.

Регистр температуры доступен только для чтения и имеет размер 16 бит. Используются только старшие 8 бит (7—15). Установка 0 бита в 1 определяет значение температуры в 0,5 ºС.

Регистр конфигурации доступен для чтения и записи. Имеет размер 8 бит. Формат регистра конфигурации показан в табл. 16.29.

Таблица 16.29. *Формат регистра конфигурации для LM75*

Биты	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Описание	0	0	0	Дефекты		П	Режим	Питание

Описание табл. 16.29.

- Бит 0, установленный в 1, позволяет выключить подачу напряжения к микросхеме LM75.
- Бит 1 определяет режим работы микросхемы: 1 — генерировать прерывания, 0 — режим сравнения значений.
- Бит 2 позволяет установить полярность измерений (1 — выше нуля, 0 — ниже нуля).

- Биты 3—4 позволяют установить количество обнаруженных дефектов перед выходом в рабочий режим, которые будут игнорироваться. Для завершения процесса инициализации микросхемы достаточно 100 мс.
- Биты 5—7 должны быть установлены в 0.

Регистры установки предельного значения гистерезиса температуры и стандартного значения температуры доступны для чтения и записи. Имеют размер 16 бит, хотя используются только старшие 8 бит (7—15). По умолчанию значение предельного гистерезиса температуры равно 80 ºС, а стандартное значение 75 ºС.

Вот и все основные сведения о температурном датчике. Теперь рассмотрим примеры программирования функций мониторинга системы. Вначале выполним сброс и последующую инициализацию микросхемы, как показано в листинге 16.1. Хочу заметить, что для считывания текущих значений температуры, напряжений и оборотов вентилятора выполнять инициализацию не нужно. Это делается только в том случае, если вы хотите перенастроить параметры микросхемы по-новому или, когда микросхема не отвечает.

Листинг 16.1. Инициализация микросхемы мониторинга

```
// пишем функцию инициализации микросхемы LM79
bool Init_LM79 ( )
{
   unsigned int uBasePort = 0x290; // номер базового порта
   DWORD dwResult = 0;
   // выполняем сброс микросхемы
   // выбираем регистр конфигурации 40h
   outPort ( uBasePort + 5, 0x40, 1 );
   // записываем значение инициализации в регистр данных x6h
   outPort ( uBasePort + 6, 0x80, 1 );
   // проверяем наличие микросхемы мониторинга
   // выбираем регистр конфигурации 40h
   outPort ( uBasePort + 5, 0x40, 1 );
   // читаем значение регистра конфигурации 40h из порта данных x6h
   inPort ( uBasePort + 6, &dwResult, 1 );
   // если полученное значение не равно 0x08, микросхема отсутствует
   if ( dwResult != 0x08 ) return false;
   // выполняем инициализацию микросхемы
   // значение по умолчанию для регистра SMI 1
   // выбираем регистр SMI 1
   outPort ( uBasePort + 5, 0x43, 1 );
   // записываем значение инициализации в регистр данных x6h
   outPort ( uBasePort + 6, 0xFF, 1 );
```

```
   // выбираем регистр SMI 2
   outPort ( uBasePort + 5, 0x44, 1 );
   // записываем значение инициализации в регистр данных x6h
   outPort ( uBasePort + 6, 0xFF, 1 );
   // выбираем регистр NMI 1
   outPort ( uBasePort + 5, 0x45, 1 );
   // записываем значение инициализации в регистр данных x6h
   outPort ( uBasePort + 6, 0xFF, 1 );
   // выбираем регистр NMI 2
   outPort ( uBasePort + 5, 0x46, 1 );
   // записываем значение инициализации в регистр данных x6h
   outPort ( uBasePort + 6, 0xFF, 1 );
   // запускаем мониторинг системы
   // выбираем регистр конфигурации 40h
   outPort ( uBasePort + 5, 0x40, 1 );
   // читаем значение регистра конфигурации 40h из порта данных x6h
   inPort ( uBasePort + 6, &dwResult, 1 );
   // устанавливаем бит 1 для включения мониторинга
   dwResult |= 0x01;
   // разрешаем прерывания
   dwResult & = ~(0x08);
   // определяем генерацию NMI-прерываний
   dwResult |= 0x04;
   // выбираем прерывания IRQ
   dwResult & = ~(0x20);
   // записываем значение в регистр конфигурации
   // выбираем регистр конфигурации 40h
   outPort ( uBasePort + 5, 0x40, 1 );
   // записываем значение инициализации в регистр данных x6h
   outPort ( uBasePort + 6, dwResult, 1 );
   return true; // выходим из функции
}
```

Для того чтобы остановить процесс мониторинга, можно применить код из листинга 16.2.

Листинг 16.2. Остановка процесса мониторинга

```
// пишем функцию для остановки процесса мониторинга
void Stop_LM79 ( )
{
   unsigned int uBasePort = 0x290; // номер базового порта
   DWORD dwResult = 0;
```

```
   // выбираем регистр конфигурации 40h
   outPort ( uBasePort + 5, 0x40, 1 );
   // читаем значение регистра конфигурации 40h из порта данных x6h
   inPort ( uBasePort + 6, &dwResult, 1 );
   // бит 0 регистра конфигурации сбрасываем в 0
   dwResult & = ~(0x01);
   // сохраняем новое значение байта конфигурации
   outPort ( uBasePort + 5, 0x40, 1 );
   outPort ( uBasePort + 6, dwResult, 1 );
}
```

А теперь рассмотрим пример, позволяющий получить текущее значение оборотов вентилятора (листинг 16.3).

Листинг 16.3. Получение текущего значения оборотов вентилятора

```
// пишем функцию для определения числа оборотов вентилятора
unsigned int Get_CoolerSpeed ( unsigned int uCooler )
{
   unsigned int uBasePort = 0x290; // номер базового порта
   unsigned int uCurrentCounter = 1; // счетчик
   unsigned int uCurrentDiv = 0; // делитель
   DWORD dwResult = 0;
   if ( ( uCooler < 1 ) || ( uCooler > 3 ) ) return 0;
   // получаем текущее значение счетчика из регистра 28h
   outPort ( uBasePort + 5, 0x28, 1 );
   // читаем значение регистра конфигурации 28h из порта данных x6h
   inPort ( uBasePort + 6, &dwResult, 1 );
   // сохраняем его в переменную
   uCurrentCounter = dwResult;
   // проверяем, крутится ли вообще вентилятор
   if ( uCurrentCounter == 0xFF )
        return 0; // вентилятор не крутится или сгорел
   // получаем значение делителя из регистра 47h
   outPort ( uBasePort + 5, 0x47, 1 );
   // читаем значение регистра делителя 47h из порта данных x6h
   inPort ( uBasePort + 6, &dwResult, 1 );
   // выделяем значение делителя для указанного вентилятора
   switch ( uCooler )
   {
   case 1:
     uCurrentDiv = 1 << ( ( dwResult >> 4 ) & 0x03 );
     break;
```

```
    case 2:
      uCurrentDiv = 1 << ( ( dwResult >> 6 ) & 0x03 );
      break;
    case 3:
      uCurrentDiv = 2; // всегда постоянное значение
      break;
    }
    // считаем количество оборотов в минуту для вентилятора
    // и выходим из функции
    return ( 1 350 000 / ( uCurrentCounter * uCurrentDiv ) );
}
```

Для установки нового значения оборотов вентилятора можно применить код из листинга 16.4.

Листинг 16.4. Установка нового значения оборотов для первого вентилятора

```
void Set_CoolerSpeed_4 ( )
{
    unsigned int uBasePort = 0x290; // номер базового порта
    unsigned int uCurrentDiv = 2; // делитель
    DWORD dwResult = 0;
    // получаем текущее значение из регистра 47h
    outPort ( uBasePort + 5, 0x47, 1 );
    // читаем значение регистра делителя 47h из порта данных x6h
    inPort ( uBasePort + 6, &dwResult, 1 );
    // выделяем значение делителя для указанного вентилятора
    dwResult &= ~ ( 3 << 4 );
    uCurrentDiv = uCurrentDiv << 4;
    dwResult |= uCurrentDiv;
    // записываем результат в регистр делителя
    outPort ( uBasePort + 5, 0x47, 1 );
    // записываем новое значение в порт данных
    outPort ( uBasePort + 6, dwResult, 1 );
}
```

В листинге 16.5 показан способ считывания текущего значения температуры материнской платы.

Листинг 16.5. Получение текущей температуры материнской платы

```
int Get_MB_Temp ( )
{
    unsigned int uBasePort = 0x290; // номер базового порта
    DWORD dwResult = 0;
```

```
   // получаем текущее значение из регистра 27h
   outPort ( uBasePort + 5, 0x27, 1 );
   // читаем значение температуры из порта данных x6h
   inPort ( uBasePort + 6, &dwResult, 1 );
   return dwResult;
}
// пример использования функции
int iTemperature = 0;
char sxText[30];
// получаем текущее значение температуры
iTemperature = Get_MB_Temp ( );
sprintf ( szText, "Температура: %2d.00 ", iTemperature );
```

Ну и последний пример для получения текущих напряжений питания показан в листинге 16.6.

Листинг 16.6. Получение текущих напряжений питания

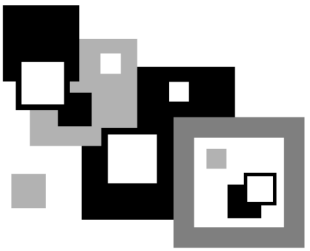
```
float Get_Volt ( unsigned int OffsetReg )
{
   unsigned int uBasePort = 0x290; // номер базового порта
   DWORD dwResult = 0;
   float fResult = 0.0000, fV_1, fV_2;
   // в зависимости от номера регистра, считаем базовое значение
   switch ( OffsetReg )
   {
   case 0: // регистр 20h для нулевой линии
   case 1: // регистр 21h для первой линии
   case 2: // регистр 22h для второй линии
     fV_1 = 0.00;
     fV_2 = 0.0;
     break;
   case 3: // регистр 23h для третьей линии
     fV_1 = 2.98;
     fV_2 = 5.0;

     break;
   case 4: // регистр 24h для четвертой линии
     fV_1 = 3.00;
     fV_2 = 12.0;
     break;
   case 5: // регистр 25h для пятой линии
     fV_1 = 3.00;
```

```
      fV_2 = -12.0;
      break;
    case 5: // регистр 26h для шестой линии
      fV_1 = 3.50;
      fV_2 = -5.0;
      break;
    }
    // получаем текущее значение из регистра
    outPort ( uBasePort + 5, OffsetReg, 1 );
    // читаем значение напряжения из порта данных x6h
    inPort ( uBasePort + 6, &dwResult, 1 );
    // вычисляем значение напряжения
    fResult = ( 0.016 * ( float ) dwResult ) — fV_1 + fV_2;
    return fResult;
}
// применим функцию на практике
char buffer[300];
float fCore, fn5, fp5, fn12, fp12, fc33, f33;
// получаем значения напряжений
fCore = Get_Volt ( 0 );
fn5 = Get_Volt ( 6 );
fp5 = Get_Volt ( 3 );
fn12 = Get_Volt ( 5 );
fp12 = Get_Volt ( 4 );
fc33 = Get_Volt ( 1 );
f33 = Get_Volt ( 2 );
// форматируем значения в удобочитаемый вид
sprintf ( buffer, "Vcore: %2.2f V \n -5 V: %2.2f V \n \
+5 V: %2.2f V \n -12 V: %2.2f V \n +12 V: %2.2f V \n \
3,3 V: %2.2f V \n 3,3 V: %2.2f V \n", fn5, fp5, fn12, fp12, fc33, f33 );
```

На этом можно завершить тему аппаратного мониторинга системы. Существует еще много различных модификаций датчиков и обо всех рассказать просто невозможно. Однако материал этой главы поможет вам самостоятельно разобраться со всеми другими существующими модулями и сенсорами. Главное, не забывайте обращаться за обновленной информацией на сайты производителей оборудования.

ГЛАВА 17



Параллельный и последовательный порты

Последовательный и параллельный порты так прочно вошли в конфигурацию компьютера, что, несмотря на появление новых высокоскоростных интерфейсов, продолжают оставаться без изменений и дополнений. Почему же так происходит и что в них такого особенного? В первую очередь, это простота исполнения и отработанный годами протокол передачи данных. Во-вторых, эти интерфейсы идеально подходят для подключения определенных низкоскоростных устройств: модема и принтера. В-третьих, данные порты (особенно последовательный) широко применяются для работы со специфическим промышленным и научным оборудованием. В-четвертых, в мире так много накопилось устройств, использующих эти интерфейсы, что еще не скоро производители современных компьютерных систем откажутся от них.

В любом случае, рассмотрим вопросы программирования параллельных и последовательных портов в следующем порядке:

- с использованием аппаратных портов;
- с помощью интерфейса Win32 API.

17.1. Общие сведения

В случае *параллельного порта* параллельная передача данных построена по принципу одновременной передачи 8 битов информации по 8 отдельным линиям (проводам). К тому же, данные могут передаваться лишь в одном направлении. Для организации двунаправленной передачи потребуется специальный кабель и соответствующее программное обеспечение.

В процессе передачи данных (1 байт за один цикл) приемная и передающая стороны сообщают по специальной линии о своем состоянии. Для этого применяются специальные сигналы. Когда устройство (принтер) занято, оно вы-

дает сигнал BUSY, а когда программа подготовила байт данных для передачи, она передает устройству сигнал STROBE. Существуют еще специальные линии для контроля передаваемых данных, для сообщения об отсутствии бумаги, для передачи сообщений об ошибках и готовности устройства.

Параллельный интерфейс (LPT — Line Printer) в компьютере состоит из трех независимых портов: LPT1, LPT2 и LPT3. Кроме того, имеется три основных типа параллельных портов:

- *стандартный*;
- *EPP* (Enhanced Parallel Port);
- *ECP* (Extended Capability Port).

Стандартный порт поддерживает только однонаправленную передачу данных (от компьютера к устройству). Максимальная скорость передачи может составлять от 120 до 200 Кбайт/с. Порт EPP является двунаправленным и поддерживает скорость передачи до 2 Мбайт/с. Данный порт позволяет использовать канал прямого доступа к памяти (DMA). Порт ECP аналогичен EPP, но дополнительно позволяет сжимать данные, что еще больше увеличивает общую скорость обмена данными. Сжатие реализуется как программно, так и аппаратно.

Последовательный порт реализован на базе интерфейса RS-232. Как правило, компьютер поддерживает до четырех портов: COM1, COM2, COM3 и COM4. Термин COM является сокращением от английского слова communications (связь). Передача данных выполняется побитно в двух направлениях и с одинаковой частотой. Передаваемый блок данных в асинхронном режиме состоит из стартового бита, битов данных (8) и стоп-бита. Скорость передачи данных измеряется в бодах (бит в секунду вместе со служебными битами) и может иметь следующие значения: 110, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600 и 115200 бод. Если два устройства (модема) имеют разную скорость передачи, будет использована наименьшая из двух. Стартовый и стоп-бит определяют начало и окончание блока данных. Дополнительно, для выявления ошибок передачи, добавляется бит четности. Его использование имеет три режима: без бита четности (No Parity), нечетный бит четности (Odd Parity) и четный бит четности (Even Parity). Перед началом передачи данных следует выполнить настройку и инициализацию обоих устройств.

17.2. Использование портов

Как мы уже знаем, стандартное количество параллельных портов ограничено тремя: LPT1, LPT2 и LPT3. Порты ввода-вывода распределены согласно табл. 17.1.

Таблица 17.1. *Распределение параллельных портов ввода-вывода*

LPT1 или LPT2	LPT2 или LPT3	Описание порта
378h	278h	Порт данных
379h	279h	Порт состояния
37Ah	27Ah	Порт управления

Можно напрямую обращаться к этим портам или же получить их значения из области памяти BIOS (0040h:0008h и 0040h:000Ah). Рассмотрим порты ввода-вывода подробнее.

Порт данных (378h или 278h) служит для передачи данных между компьютером и устройством (например, принтером) и доступен как для записи, так и для чтения.

Порт состояния (379h или 279h) позволяет получить информацию о текущем состоянии подключенного устройства, и доступен только для чтения. Его формат представлен в табл. 17.2.

Таблица 17.2. *Формат порта состояния*

Бит	Описание
0—2	Не используются и должны быть установлены в 0
3	Ошибка ввода-вывода (1 — нет, 0 — есть)
4	Выбор принтера (1 — принтер доступен, 0 — принтер не доступен)
5	Закончилась бумага (1 — нет бумаги, 0 — есть)
6	Подтверждение приема данных (0 — подтверждение есть, 1 — принтер не готов)
7	Готовность принтера (1 — принтер не занят, 0 — принтер занят)

Порт управления (37Ah или 27Ah) позволяет инициализировать принтер и установить различные параметры работы. Формат этого порта показан в табл. 17.3.

Таблица 17.3. *Формат порта управления*

Бит	Описание
0	Состояние линии STROBE (1 — можно передать данные на принтер, 0 — стандартное состояние)
1	Использование автоматического перевода строки LF (Ah) после возврата каретки CR (Dh): 1 — использовать

Таблица 17.3 *(окончание)*

Бит	Описание
2	Инициализация порта принтера (0 — выполнить начальную инициализацию)
3	Выбор принтера (1 — принтер доступен, 0 — принтер не доступен)
4	Использовать прерывание от принтера (1 — использовать, 0 — нет)
5—7	Не используются и должны быть установлены в 0

А теперь рассмотрим практические примеры работы с параллельными портами. Перед началом работы необходимо инициализировать порт, как это сделано в листинге 17.1. Кроме того, инициализацию нужно выполнять повторно, если произошла ошибка передачи данных.

Листинг 17.1. Инициализация параллельного порта

```
// пишем функцию инициализации
void InitLPT ( unsigned int Port )
{
   outPort ( Port, 0x0C, 1 ); // подготавливаем порт к работе
   Sleep ( 1 ); // небольшая задержка
   outPort ( Port, 0x08, 1 ); // инициализируем порт
}
// используем функцию для инициализации порта 27Ah
InitLPT ( 0x27A );
```

Для проверки готовности принтера можно использовать код, представленный в листинге 17.2.

Листинг 17.2. Проверка готовности порта LPT

```
// пишем функцию проверки подключения принтера
bool IsReady_LPT ( unsigned int Port )
{
   DWORD dwResult = 0;
   inPort ( Port, &dwResult, 1 ); // читаем байт из порта
   if ( ( dwResult & 0x10 ) == 0x01 )
     return true; // принтер подключен
   return false; // принтер не подключен
}
```

Для того чтобы послать на принтер байт данных, необходимо выполнить следующие действия:

1. Послать байт данных в порт данных.
2. Затем следует установить бит 0 управляющего порта в 0, а после небольшой паузы в 1.
3. После этого следует ожидать готовности принтера к приему следующего байта данных. Для этого постоянно опрашивается бит 7 в регистре статуса. Когда он будет установлен в 1, можно переходить к пункту 2.

Пример записи данных в порт принтера показан в листинге 17.3.

Листинг 17.3. Передача данных в порт LPT

```
// пишем функцию для передачи байта на принтер
int SetByte_LPT ( unsigned int BasePort, BYTE bData )
{
   DWORD dwResult = 0;
   int iTimeWait = 5000;
   // проверяем готовность порта
   while ( -- iTimeWait > 0 )
   {
       // читаем порт состояния
       inPort ( BasePort + 1, &dwResult, 1 );
       if ( (dwResult & 0x80 ) == 0x00 ) break;
       // закончилось время ожидания
       if ( iTimeWait < 1 ) return 1; // вышло время ожидания
   }
   // записываем значение байта данных в порт
   outPort ( BasePort, bData, 1 );
   // устанавливаем сигнал STROBE
   outPort ( BasePort + 2, 0x0D, 1 );
   // сбрасываем сигнал STROBE
   outPort ( BasePort + 2, 0x0C, 1 );
   // проверяем готовность принтера к приему следующего символа
   iTimeWait = 10000;
   while ( -- iTimeWait > 0 )
   {
       // читаем порт состояния
       inPort ( BasePort + 1, &dwResult, 1 );
       if ( (dwResult & 0x08 ) == 0x00 ) return 2; // произошла ошибка
       if ( (dwResult & 0x80 ) == 0x01 ) break;
```

```
        // закончилось время ожидания
        if ( iTimeWait < 1 ) return 1; // вышло время ожидания
    }
    return 0;
}
```

Работа с последовательным портом лишь немного сложнее параллельного. Стандартное распределение портов показано в табл. 17.4.

Таблица 17.4. *Распределения последовательных портов ввода-вывода*

Номер порта	Регистры ввода-вывода
COM1	3F8h—3FFh
COM2	2F8h—2FFh
COM3	3E8h—3EFh
COM4	2E8h—2EFh

Из таблицы видно, что для доступа к последовательному порту выделены 10 регистров, каждый из которых имеет свое назначение. Все регистры имеют размер 8 бит.

Регистр 3F8h выполняет несколько задач и доступен для записи или чтения. Если бит 7 в регистре управления (3FBh) равен 0, то через регистр 3F8h можно читать или записывать данные. Если бит 7 в регистре управления (3FBh) равен 1, то регистр 3F8h обрабатывает младший байт делителя частоты для скорости передачи данных.

Регистр 3F9h также выполняет несколько задач и доступен для записи или чтения. Если бит 7 в регистре управления (3FBh) равен 0, то регистр 3F9h управляет прерыванием (см. табл. 17.5). Если бит 7 в регистре управления (3FBh) равен 1, то регистр 3F9h обрабатывает старший байт делителя частоты для скорости передачи данных.

Таблица 17.5. *Формат регистра 3F9h в режиме управления прерыванием*

Бит	Описание
0	Состояние сигнала прерывания при наличии входных данных (1 — включен, 0 — выключен)
1	Состояние сигнала прерывания при опустошении выходного буфера (1 — включен, 0 — выключен)
2	Состояние сигнала прерывания при появлении ошибки или перерыва в передаче данных (1 — включен, 0 — выключен)

Таблица 17.5 *(окончание)*

Бит	Описание
3	Состояние сигнала прерывания при изменении состояния модема (1 — включен, 0 — выключен)
4—7	Не используются и должны быть установлены в 0

Регистр `3FAh` доступен для записи или чтения. Позволяет определить причину прерывания в режиме чтения (табл. 17.6). В режиме записи управляет FIFO (табл. 17.7).

Таблица 17.6. *Формат регистра 3F9h в режиме чтения*

Бит	Описание
0	Состояние прерывания (1 — нет прерывания, 0 — есть прерывание)
1—2	Причина прерывания: `00b` — изменение состояния модема, `01b` — опустошен буфер передачи, `10b` — получены данные, `11b` — произошла ошибка или перерыв в передаче данных
3	Закончилось время ожидания (тайм-аут) для приемника FIFO
4—5	Не используются и должны быть установлены в 0
6—7	Наличие FIFO (`11b` — присутствует, `00b` — отсутствует)

Таблица 17.7. *Формат регистра 3F9h в режиме записи*

Бит	Описание
0	Управление работой режима FIFO (1 — включен, 0 — выключен)
1	Очистка приемника FIFO (1 — очистить)
2	Очистка передатчика FIFO (1 — очистить)
3—5	Не используются и должны быть установлены в 0
6—7	Порог срабатывания для чтения данных (`00b` — 1 байт, `01b` — 4 байта, `10b` — 8 байт, `11b` — 14 байт)

Регистр `3FBh` доступен для записи или чтения. Регистр предназначен для управления линией связи. Формат регистра показан в табл. 17.8.

Таблица 17.8. *Формат регистра управления*

Бит	Описание
0—1	Длина данных (`00b` — 5 бит, `01b` — 6 бит, `10b` — 7 бит, `11b` — 8 бит)
2	Количество стоповых битов (0 — 1 бит, 1 — 2 бита)

Таблица 17.8 *(окончание)*

Бит	Описание
3—5	Бит четности (000b — нет, 001b — нечетный, 011b — четный)
6	Состояние перерыва передачи, при котором порты выдают нули (1 — включить, 0 — выключить)
7	Управление делителем частоты для выбора скорости передачи данных (1 — установить значение делителя частоты через регистры 3F8h и 3F9h)

Регистр 3FCh доступен для записи или чтения. Он служит для управления работой модема. Формат регистра показан в табл. 17.9.

Таблица 17.9. *Формат регистра управления модемом*

Бит	Описание
0	Управление линией DTR
1	Управление линией RTS
2	Управление линией OUT1 (0)
3	Управление линией OUT2 (1)
4	Самотестирование модема (1 — включить), при котором выход замыкается на вход
5—7	Не используются и должны быть установлены в 0

Регистр 3FDh доступен только для чтения. Позволяет получить текущее состояние линии связи. Формат регистра показан в табл. 17.10.

Регистр 3FEh доступен только для чтения. Позволяет получить текущее состояние модема. Формат регистра показан в табл. 17.11.

Таблица 17.10. *Формат байта состояния порта*

Бит	Описание
0	Готовность данных (1 — готовы, 0 — не готовы)
1	Ошибка переполнения данных (1 — есть, 0 — нет)
2	Ошибка четности (1 — есть, 0 — нет)
3	Ошибка синхронизации данных (1 — есть, 0 — нет)
4	Обнаружен перерыв в передаче данных (1 — да, 0 — нет)
5	Регистр хранения данных пустой (1 — нет данных, 0 — есть данные)
6	Регистр сдвига пустой (1 — нет данных, 0 — есть данные)
7	Тайм-аут или ошибка по времени (1 — ошибка, 0 — нет ошибки)

Таблица 17.11. *Формат байта состояния модема*

Бит	Описание
0	Изменилось состояние на входной линии CTS (Clear To Send): 1 — есть
1	Изменилось состояние на входной линии DSR (Data Set Ready): 1 — есть
2	Изменилось состояние на входной линии RI (Ring Indicator): 1 — есть
3	Изменилось состояние на входной линии DCD (Data Carrier Detect) 1 — есть
4	Произошел сброс для передачи на входной линии CTS (Clear To Send): 1 — сброс
5	Готовность модема для чтения данных на входной линии DSR (Data Set Ready): 1 — готов, 0 — не готов
6	Состояние индикатора звонка на входной линии RI (Ring Indicator): 1 — есть сигнал
7	Сигнал обнаружения несущей на входной линии DCD (Data Carrier Detect): 1 — обнаружен

Регистр `3FCh` доступен для записи или чтения. Он является резервным и не используется.

Перед началом работы следует инициализировать последовательный порт. Для этого надо вначале установить бит 7 в 1 для регистра `3FBh`, а затем записать желаемую скорость передачи (старший и младший байты) в регистры `3F8h` и `3F9h`. После этого в регистр `3FBh` можно записать режим работы линии, обнулив бит 7. Затем в регистр `3F9h` надо записать 0, если прерывания не будут использованы, или установить биты 0—3 в соответствии с требуемыми прерываниями. Возможные значения для скорости передачи данных представлены в табл. 17.12.

Таблица 17.12. *Коды значений для установки скорости передачи данных*

Код значения		Скорость, бод
3F9h	3F8h	
04h	17h	110
01h	80h	300
00h	C0h	600
00h	60h	1 200
00h	30h	2 400
00h	20h	3 600
00h	18h	4 800

Таблица 17.12 *(окончание)*

Код значения		Скорость, бод
3F9h	3F8h	
00h	10h	7 200
00h	0Ch	9 600
00h	06h	19 200
00h	03h	38 400
00h	02h	57 600
00h	01h	115 200

В листинге 17.4 показано, как выполняется инициализация последовательного порта без использования прерываний.

Листинг 17.4. Инициализация последовательного порта

```
// пишем функцию инициализации последовательного порта
void Init_COM ( unsigned int BasePort )
{
   // устанавливаем бит 7 в 1 в регистре управления линией
   outPort ( BasePort + 3, 0x80, 1 );
   // устанавливаем скорость передачи данных 9 600 бод
   outPort ( BasePort, 0x00, 1 ); // младший байт делителя
   outPort ( BasePort + 1, 0x0C, 1 ); // старший байт делителя
   // настраиваем регистр управления линией
   // 8 бит, 1 стоповый бит, без четности
   outPort ( BasePort + 3, 0x03, 1 );
   // настраиваем регистр управления модемом
   // DTR, RTS, OUT1, OUT2
   outPort ( BasePort + 4, 0x0B, 1 );
   // настраиваем регистр управления прерываниями
   outPort ( BasePort + 1, 0x00, 1 ); // запрещаем все прерывания
}
```

Для проверки текущего состояния линии можно использовать код, представленный в листинге 17.5.

Листинг 17.5. Проверка состояния последовательного порта

```
// проверка состояния линии перед записью данных в порт
bool IsRead_Write_COM ( unsigned int BasePort )
```

```
{
   DWORD dwResult = 0;
   inPort ( BasePort + 5, &dwResult, 1 );
   if ( ( dwResult & 0x10 ) == 0x01 )
     return false; // линия занята
   return true;
}
// проверка состояния линии перед чтением данных из порта
bool IsRead_Read_COM ( unsigned int BasePort )
{
   DWORD dwResult = 0;
   inPort ( BasePort + 5, &dwResult, 1 );
   if ( ( dwResult & 0x02 ) == 0x01 )
     return false; // в линии остались непрочитанные данные
   return true;
}
```

Теперь, когда последовательный порт готов к работе, можно записать в него данные (листинг 17.6).

Листинг 17.6. Запись байта данных в последовательный порт

```
// пишем функцию для записи байта в последовательный порт
void SendByte_COM ( unsigned int BasePort, char* data )
{
   // записываем байт в порт
   outPort ( BasePort, ( DWORD ) data, 1 );
}
```

Пример чтения байта из последовательного порта представлен в листинге 17.7.

Листинг 17.7. Чтение байта данных из последовательного порта

```
// пишем функцию для чтения байта из последовательного порта
BYTE ReadByte_COM ( unsigned int BasePort )
{
   DWORD dwResult = 0;
   // читаем байт из порта
   inPort ( BasePort, &dwResult, 1 );
   return ( BYTE ) dwResult;
}
```

На этом я хотел бы завершить программирование портов ввода-вывода и немного рассказать о поддержке последовательных и параллельных портов в интерфейсе Win32.

17.3. Использование Win32 API

Работа с LPT- и COM-портами в Windows упрощена до минимума. Чтобы инициализировать порт, достаточно вызвать функцию `CreateFile` с соответствующими аргументами. После этого можно писать и читать данные в порт посредством стандартных функций `WriteFile` и `ReadFile`. Кроме того, для настройки портов и последующего управления ими предназначен целый ряд дополнительных функций. Рассмотрим основные моменты программирования портов подробнее.

Перед началом работы необходимо открыть порт и настроить его на требуемый режим работы (для COM). В листинге 17.8 показано, как можно это сделать.

Листинг 17.8. Начальная инициализация последовательного порта COM1

```
#include <windows.h>
// объявляем структуру для конфигурации последовательного порта
DCB dcb;
ZeroMemory ( &dcb, sizeof ( DCB ) );
// дескриптор порта
HANDLE hCom_1 = NULL;
// пишем функцию инициализации порта COM1
bool Init_COM1 ( )
{
   // открываем порт COM1 ( для COM2 "COM2", для LPT1 "LPT1" и т. д. )
   hCom_1 = CreateFile ( "COM1", GENERIC_READ | GENERIC_WRITE, 0, NULL,
                         OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL );
   if ( hCom_1 == INVALID_HANDLE_VALUE ) // если порт не удалось открыть
      return false; // выходим из функции
   // если порт успешно открыт, получаем его состояние
   if ( !GetCommState ( hCom_1, &dcb ) )
   {
    // если не удалось получить статус порта, выходим из функции
    CloseHandle ( hCom_1 );
    return false; // выходим из функции
   }
   // настраиваем параметры порта
   dcb.BaudRate  = CBR_19200; // скорость передачи 19 200 бод
   dcb.ByteSize = 8; // размер байта данных
```

```
   dcb.StopBits = ONESTOPBIT; // один стоповый бит
   dcb.Parity = NOPATITY; // контроля четности нет
   // сохраняем новые параметры конфигурации для порта
   if ( !SetCommState ( hCom_1, &dcb ) )
   {
    // если не удалось настроить порт, выходим из функции
    CloseHandle ( hCom_1 );
    return false; // выходим из функции
   }
   return true;
}
```

Теперь, когда наш порт открыт и сконфигурирован для работы, нужно настроить обработку сигналов DSR и CTS. Пример кода для активизации уведомлений для последовательного порта показан в листинге 17.9.

Листинг 17.9. Настройка уведомляющих событий для сигналов DSR и CTS

```
// объявляем структуру для асинхронного ввода-вывода данных
// и помещаем ее в глобальную область видимости
OVERLAPPED over;
ZeroMemory ( &over, sizeof ( OVERLAPPED ) );
// пишем функцию настройки событий
bool SetEventCOM ( )
{
   // настраиваем мониторинг за определенными событиями порта
   if ( !SetCommMask ( hCom_1, EV_DSR | EV_CTS ) )
   {
    // если не удалось настроить порт, выходим из функции
    CloseHandle ( hCom_1 );
    return false; // выходим из функции
    }
   // создаем объект события
   over.hEvent = CreateEvent ( NULL, FALSE, FALSE, NULL );
   if ( !over.hEvent )
   {
    // если не удалось создать событие, выходим из функции
    CloseHandle ( hCom_1 );
    return false; // выходим из функции
   }
   return true;
}
```

На данный момент вся начальная подготовка для работы с портом закончена. Осталось только написать функции чтения и записи данных и функцию

управления для последовательного порта. Примеры функций чтения и записи показаны в листинге 17.10.

Листинг 17.10. Функции чтения и записи для работы с последовательным портом

```
// функция чтения одного байта данных
BYTE ReadByteCOM ( )
{
   BYTE read = 0;
   DWORD dwByteRead = 0;
   do
   { // читаем байт из порта
     if ( !ReadFile ( hCom_1,  &read, sizeof ( BYTE ), &dwByteRead,
                      NULL ) )
       return 0;
   }  while ( !dwByteRead );
return read; // возвращаем прочитанный байт
}
// функция чтения массива данных
DWORD ReadData_COM ( void* Data, unsigned int uNumBytes )
{
   DWORD dwBytesRead = 0;
   if ( !ReadFile ( hCom_1, Data, uNumBytes, &dwBytesRead, NULL ) )
      return dwBytesRead; // сколько удалось прочитать
   return dwBytesRead; // возвращаем полное число прочитанных байтов
}
// функция для записи одного байта
bool WriteByteCOM ( BYTE bByte )
{
   BYTE write = 0;
   DWORD dwByteWrite = 0;
   if ( !WriteFile ( hCom_1, &write, sizeof ( BYTE ), &dwByteWrite,
                     NULL ) )
       return false;
   return true;
}
// функция для записи массива данных
DWORD WriteDataCOM ( void* Data, unsigned int uNumBytes )
{
   DWORD dwBytesWrite = 0;
   if ( !WriteFile ( hCom_1, Data, uNumBytes, &dwByteWrite,
                     NULL ) )
```

```
        return dwBytesWrite; // сколько удалось записать
    return dwBytesWrite; // возвращаем полное число записанных байтов
}
```

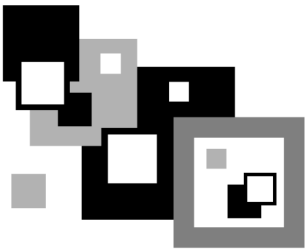
Теперь пишем общую функцию для работы с портом (листинг 17.11).

Листинг 17.11. Общая функция для работы с последовательным портом

```
// добавляем в глобальную область данных значение для выделения сигнала
DWORD dwSignal;
void GeneralCOM ( )
{
   // проверяем сигнал в линии
   if ( WaitCommEvent ( hCom_1, &dwSignal, &over ) )
   {
     if ( dwSignal & EV_DSR ) // данные готовы для чтения
     {
       // читаем байт из порта
       BYTE data = ReadByteCOM ( );
       // сохраняем полученный байт куда-либо
     }
     if ( dwSignal & EV_CTS ) // можно писать данные в порт
     {
       // передаем извне байт и пишем его в порт
       WriteByteCOM ( myByte );
     }
   }
}
// после завершения работы следует вызвать функцию очистки ресурсов
void CloseCOM ( )
{
   if ( over.hEvent )
   {
     CloseHandle ( over.hEvent ); // закрываем объект события
     over.hEvent = NULL;
   }
   if ( hCom_1 )
   {
     CloseHandle ( hCom_1 ); // закрываем порт COM1
     hCom = NULL;
   }
}
```

Вот, в принципе, и все. Дополнительные возможности, предоставляемые интерфейсом Win32, вы сможете добавить самостоятельно.

ГЛАВА 18



Современные интерфейсы

В последнее время развитие компьютерной индустрии все больше направлено не на создание принципиально новых решений, а скорее на поддержку и расширение существующих. Это в первую очередь касается увеличения пропускной способности, удобства использования и взаимодействия технически сложных устройств между собой.

Не секрет, что, например, интерфейс *USB* (Universal Serial Bus — универсальная последовательная шина обмена данными) появился достаточно давно. Однако наиболее широкое распространение он получил в последние годы, в первую очередь в связи с появлением всевозможных миниатюрных устройств хранения данных (внешних накопителей). Кроме того, получили широкое распространение и другие интерфейсы, такие как *IrDA* (Infra red Data Assotiation — интерфейс связи, основанный на открытом оптическом канале в инфракрасном диапазоне), *IEEE 1394* (цифровой интерфейс, использующий последовательную шину), *Bluetooth* (интерфейс, осуществляющий обмен данными посредством беспроводных сетей), *Wireless LAN* (или *Wi-Fi* — Wireless Fidelity, высокоскоростной беспроводный интерфейс обмена данными), *DECT* (Digital Enhanced Cordless Telecommunication — цифровой интерфейс беспроводной связи). Большая часть из перечисленных технологий позволяет подключать различные устройства к компьютеру не напрямую, а через платы сопряжения (переходники, контроллеры и т. д.). В свою очередь платы сопряжения используют стандартные интерфейсы, такие как USB, COM, PCI. Поэтому нетрудно предположить, что процесс разработки программного обеспечения для управления внешними устройствами может быть сведен к программированию базовых компьютерных интерфейсов. Однако это верно лишь в том случае, если программируются готовые устройства, выпущенные сторонними производителями. Если же необходимо настроить работу самостоятельно сконструированного оборудования, дополнительно при-

дется изучить спецификации выбранных интерфейсов, написать собственный драйвер и сопутствующее программное обеспечение.

Чтобы отчасти помочь читателям в решении этих задач, рассмотрим вопросы программирования некоторых из перечисленных выше интерфейсов, а также познакомимся со структурой каждого из них. Поскольку в одной главе трудно охватить все возможные аспекты, ограничимся только наиболее важными из них.

18.1. Интерфейс USB

Итак, USB представляет собой универсальный последовательный интерфейс для передачи данных. На данный момент существует две основные версии стандарта: 1.1 и 2.0. Главным отличием между ними является пропускная способность шины. Для первой версии максимальная скорость передачи данных не превышает 12 Мбит/с. Для второй эта планка составляет уже 480 Мбит/с. Здесь мы рассмотрим последнюю действующую версию, которая предоставляет следующие основные возможности:

- передачу в реальном времени аудио- и видеоданных;
- полноценный асинхронный обмен данными;
- поддержку различных конфигураций компьютера;
- полностью стандартизированную технологию для использования в продуктах различных производителей;
- полную совместимость со всеми устройствами, построенными на базе предыдущих версий.

В первую очередь, для работы с этим интерфейсом необходимо представлять его общую структуру, что мы сейчас и обсудим. Основой интерфейса служит так называемый *хост* (host) (базовое устройство, физически расположенное, как правило, на материнской плате компьютера), который связывает компьютер и любые устройства USB посредством последовательной шины. Поскольку топология интерфейса выполнена в виде звезды, дополнительно применяются *хабы* (hub) (узлы для разветвления шины на отдельные самостоятельные линии), через которые подключаются непосредственно сами устройства USB. Эту схему можно описать следующим образом: есть только один хост, он связан с корневым хабом (физически расположен на материнской плате компьютера), который в свою очередь может быть связан с дополнительным хабом или с конечными устройствами USB. Спецификация позволяет подключать последовательно до 5 хабов и до 127 конечных устройств USB.

Корневой хост включает в себя хост-контроллер, выполняющий следующие основные задачи:

- автоматически определяет подключение или удаление устройств USB;
- управляет обменом данными между хостом и устройствами USB;
- собирает статистические сведения о текущих состояниях подключенных устройств;
- обеспечивает и управляет питанием подключенные USB-устройства.

Кроме аппаратной, существует программная поддержка USB, которая обеспечивает следующие возможности:

- перечисление устройств USB на шине и получение их конфигурации;
- изохронную и асинхронную передачу данных;
- управление питанием;
- управление информацией для шины и устройств USB.

Стандартная связь между хостом и устройством USB формируется на основе достаточно сложной системы, в которую входят:

- *физическое устройство*, подключенное с одной стороны кабеля USB;
- *клиентское программное обеспечение*, разработанное для конкретного устройства USB и взаимодействующее с хостом;
- *системное программное обеспечение*, предназначенное для поддержки интерфейса USB в конкретной операционной системе;
- *хост-контроллер*, содержащий как аппаратные, так и программные средства для подключения устройства USB.

Мы не будем здесь подробно рассматривать все эти аспекты, поскольку для этого потребуется написать отдельную книгу. Все интересующие читателей детали и тонкости можно прочитать в спецификации USB, которая доступна любому на сайте **www.usb.org**.

Обмен данными по шине выполняется посредством пакетов данных. Структура каждого из них формируется на основе следующих компонентов:

- *поля синхронизации*, которое имеет ширину 8 бит для стандарта 1.1 и 32 бита для стандарта 2.0;
- *поля идентификатора пакета*, имеющего размер 8 бит, где младшие 4 бита определяют тип пакета, а старшие — содержат контрольную сумму для правильной расшифровки данных;
- *поля адреса*, служащего для определения конечной точки расположения устройства на шине;

- *дополнительного поля конечной точки*, которое представляет собой дополнительное 4-битовое поле адреса, если устройство требует более одной конечной точки;
- *поля нумерации*, представляющего собой 11-битовое поле счетчика хоста, которое принимает максимальное значение `7FFh`;
- *поля данных*, которое может иметь размер от 0 до 1024 байтов и содержит непосредственно передаваемые данные.

Для управления пакетами данных хост формирует *транзакции* (transactions) (последовательность различных действий, объединенная в общую группу), которые состоят из нескольких отдельных пакетов: пакет описания устройства USB (тип и адрес) и направления передачи (от хоста к устройству или наоборот), пакет данных и пакет подтверждения (результат приема или передачи данных). Более подробно о формате пакетов и структуре данных можно ознакомиться в спецификации. Главное необходимо понимать, что инициатором всех операций (запросы, обмен данными) является хост, а все устройства USB являются лишь "исполнителями".

18.1.1. Структура запроса

В начале работы хост посылает устройству запрос установки для идентификации и определения основных свойств. Размер установочного запроса составляет 8 байт и показан в табл. 18.1.

Таблица 18.1. *Структура запроса установки*

Смещение	Наименование поля	Размер поля, байт	Описание
0	`bmRequestType`	1	Описание запроса
1	`bRequest`	1	Дополнительное описание запроса
2	`wValue`	2	Изменяемое поле запроса
4	`wIndex`	2	Поле индекса или смещения
6	`wLength`	2	Количество байтов передачи

Описание табл. 18.1.

- `bmRequestType` — определяет тип запроса. Формат поля представлен в табл. 18.2.
- `bRequest` — стандартный тип запроса. Спецификацией определены общие типы запросов (табл. 18.3), которые должны поддерживаться всеми устройствами USB.

- `wValue` — содержит дополнительное описание параметра запроса, как правило, тип дескриптора. Стандартные типы дескрипторов приведены в табл. 18.4.
- `wIndex` — содержит дополнительное описание параметра запроса.
- `wLength` — определяет размер передаваемых данных и зависит от бита направления передачи данных поля `bmRequestType`.

Таблица 18.2. *Формат поля* `bmRequestType`

Биты	Описание
7	Направление передачи (0 — от хоста к устройству, 1 — от устройства к хосту)
6—5	Тип запроса (0 — обычный, 1 — особый для определенного класса устройств, 2 — определенный производителем устройства, 3 — резерв)
4—0	Получатель (0 — устройство, 1 — интерфейс, 2 — конечная точка, 3 — какой-либо другой получатель)

Таблица 18.3. *Стандартные типы запросов*

Код запроса	Тип запроса	Описание
00h	GET_STATUS	Получает состояние определенного устройства (получателя)
01h	CLEAR_FEATURE	Используется для очистки или отключения определенной настройки устройства
03h	SET_FEATURE	Используется для установки или включения определенной настройки устройства
05h	SET_ADDRESS	Устанавливает адрес устройства для дальнейшего доступа
06h	GET_DESCRIPTOR	Получает дескриптор устройства, если он существует
07h	SET_DESCRIPTOR	Дополнительный запрос для обновления информации о существующем или вновь добавленном дескрипторе устройства
08h	GET_CONFIGURATION	Получает значение текущей конфигурации устройства
09h	SET_CONFIGURATION	Устанавливает конфигурацию устройства
0Ah	GET_INTERFACE	Получает выбранную настройку для определенного интерфейса
0Bh	SET_INTERFACE	Позволяет хосту выбирать дополнительные настройки для определенного интерфейса
0Ch	SYNCH_FRAME	Предназначен для установки и сообщения структуры объекта синхронизации для конечной точки

Таблица 18.4. *Стандартные типы дескрипторов*

Значение	Тип дескриптора
01h	Дескриптор устройства
02h	Дескриптор конфигурации
03h	Строковый дескриптор
04h	Дескриптор интерфейса
05h	Дескриптор конечной точки
06h	Дополнительный дескриптор устройства
07h	Дескриптор конфигурации управления питанием
08h	Дескриптор интерфейса управления питанием

После того как мы познакомились с общей структурой запросов, перейдем к описанию их форматов. Рассмотрим только стандартные запросы, поддерживаемые всеми устройствами USB.

18.1.1.1. Запрос *CLEAR_FEATURE*

Данный запрос позволяет очистить или отключить определенную настройку (свойство) устройства USB, интерфейса или конечной точки. Формат запроса представлен в табл. 18.5.

Таблица 18.5. *Формат запроса* `CLEAR_FEATURE`

Поле запроса	Значение
bmRequestType	0h — устройство, 1h — интерфейс, 2h — конечная точка
bRequest	01h
wValue	Значение свойства получателя (табл. 18.6)
wIndex	0 для конечной точки или номер устройства (интерфейса)
wLength	0
Данные	Нет

Таблица 18.6. *Значение свойства получателя*

Свойство	Получатель	Значение
ENDPOINT_HALT	Конечная точка	0
DEVICE_REMOTE_WAKEUP	Устройство USB	1
TEST_MODE	Устройство USB	2

Будет возвращена ошибка, если не существует интерфейса или конечной точки, на которую ссылается запрос, или настройка не может быть очищена в виду ее отсутствия.

18.1.1.2. Запрос *GET_CONFIGURATION*

Запрос получает значение текущей конфигурации устройства. Формат запроса представлен в табл. 18.7.

Таблица 18.7. *Формат запроса* `GET_CONFIGURATION`

Поле запроса	Значение
`bmRequestType`	`80h`
`bRequest`	`08h`
`wValue`	0
`wIndex`	0
`wLength`	1
`Данные`	Значение конфигурации

Если поле данных равно 0, значит, устройство не сконфигурировано. Иначе будет возвращен 1 байт данных конфигурации.

18.1.1.3. Запрос *GET_DESCRIPTOR*

Позволяет получить определенный дескриптор, если он существует. Формат запроса представлен в табл. 18.8.

Таблица 18.8. *Формат запроса* `GET_DESCRIPTOR`

Поле запроса	Значение
`bmRequestType`	`80h`
`bRequest`	`06h`
`wValue`	Тип и индекс дескриптора
`wIndex`	0 или `Language ID`
`wLength`	Длина дескриптора
`Данные`	Дескриптор

В поле `wValue` следует записать тип дескриптора (старший байт) и индекс дескриптора (младший байт). Доступные типы дескриптора представлены

в табл. 18.4. Следует заметить, что дескриптор индекса используется только для конфигурации или строки, иначе он должен быть равен 0.

В поле wIndex следует передать идентификатор языка (только для дескриптора строки) или 0.

В поле wLength нужно записать размер дескриптора в байтах. Если в результате выполнения запроса реальная длина дескриптора окажется больше, то хост возвратит только данные, соответствующие заданному в поле wLength значению.

После успешного выполнения запроса будет возвращена информация о запрошенном дескрипторе или сгенерирован запрос ошибки.

18.1.1.4. Запрос *GET_INTERFACE*

Позволяет получить выбранную настройку (свойство) для определенного интерфейса. Формат запроса представлен в табл. 18.9.

Таблица 18.9. *Формат запроса* GET_INTERFACE

Поле запроса	Значение
bmRequestType	81h
bRequest	0Ah
wValue	0
wIndex	Интерфейс
wLength	1
Данные	Значение текущей настройки

После выполнения запроса хост возвратит 1 байт данных с текущей настройкой интерфейса. Если интерфейс не существует, будет сгенерирован запрос ошибки.

18.1.1.5. Запрос *GET_STATUS*

Позволяет получить текущее состояние определенного получателя. В качестве получателя может быть задано устройство USB, интерфейс или конечная точка. Формат запроса представлен в табл. 18.10.

Таблица 18.10. *Формат запроса* GET_STATUS

Поле запроса	Значение
bmRequestType	80h — устройство, 81h — интерфейс, 82h — конечная точка
bRequest	00h

Таблица 18.10 *(окончание)*

Поле запроса	Значение
wValue	0
wIndex	0, интерфейс, конечная точка
wLength	2
Данные	Текущее состояние

В поле wIndex для получения статуса устройства USB нужно записать 0, а для интерфейса и конечной точки — номер. После выполнения запроса хост возвратит 2 байта (слово), содержащие информацию о статусе.

Для устройства USB следует обработать следующие биты:

- бит 0 — определяет источник питания устройства. Если бит равен 0, то питание поступает с шины. Если бит равен 1 — устройство имеет собственный источник питания.
- Бит 1 — определяет, доступен ли устройству внешний сигнал на пробуждение. Если бит равен 0 — данная возможность выключена, если бит равен 1 — включена. По умолчанию и после перезагрузки устройства данный бит всегда устанавливается в 0.

Остальные биты (с 2 по 15) зарезервированы и не используются.

Для интерфейса все биты зарезервированы и равны 0. Для конечной точки следует обработать бит 0, который определяет блокировку и принимает следующие значения:

- 0 — конечная точка доступна;
- 1 — конечная точка выключена.

Остальные биты (с 1 по 15) зарезервированы и равны 0.

18.1.1.6. Запрос *SET_ADDRESS*

Позволяет установить адрес устройства для последующего доступа. Формат запроса представлен в табл. 18.11.

Таблица 18.11. *Формат запроса* SET_ADDRESS

Поле запроса	Значение
bmRequestType	00h
bRequest	05h
wValue	Адрес

Таблица 18.11 *(окончание)*

Поле запроса	Значение
wIndex	0
wLength	0
Данные	Не используется

В поле wValue следует записать значение адреса, которое будет использовано в дальнейшем для доступа к устройству на шине. При установке адреса следует помнить, что максимальное значение не может превышать 127, поскольку так определено стандартом USB. Передача значения 0 также приведет к неопределенному результату.

18.1.1.7. Запрос *SET_CONFIGURATION*

Данный запрос позволяет задать конфигурацию для устройства USB. Формат запроса представлен в табл. 18.12.

Таблица 18.12. *Формат запроса* *SET_CONFIGURATION*

Поле запроса	Значение
bmRequestType	00h
bRequest	09h
wValue	Значение конфигурации
wIndex	0
wLength	0
Данные	Не используется

В младший байт поля wValue следует записать значение конфигурации. Оно может быть равно 0 или равно значению дескриптора конфигурации.

18.1.1.8. Запрос *SET_DESCRIPTOR*

Данный запрос является дополнительным и позволяет обновить существующий дескриптор или добавить новый. Формат запроса представлен в табл. 18.13.

В поле wValue следует записать тип дескриптора (старший байт) и индекс дескриптора (младший байт). Доступные типы дескриптора представлены в табл. 18.4. Следует заметить, что дескриптор индекса используется только для конфигурации или строки, иначе он должен быть равен 0.

Таблица 18.13. *Формат запроса* `SET_DESCRIPTOR`

Поле запроса	Значение
bmRequestType	00h
bRequest	07h
wValue	Тип или индекс дескриптора
wIndex	0 или Language ID
wLength	Длина дескриптора
Данные	Дескриптор

В поле `wIndex` следует передать идентификатор языка (только для дескриптора строки) или 0.

В поле `wLength` нужно записать размер дескриптора в байтах.

После успешного выполнения запроса будет возвращена информация о запрошенном дескрипторе или сгенерирован запрос ошибки.

18.1.1.9. Запрос *SET_FEATURE*

Этот запрос дает возможность установить или включить определенную настройку (свойство) устройства, интерфейса или конечной точки. Формат запроса представлен в табл. 18.14.

Таблица 18.14. *Формат запроса* `SET_FEATURE`

Поле запроса	Значение
bmRequestType	00h — устройство, 01h — интерфейс, 02h — конечная точка
bRequest	03h
wValue	Значение настройки
wIndex	0 или настройка режима тестирования
wLength	0
Данные	Не используется

В поле `wValue` следует записать значение настройки, соответствующее коду, заданному в поле `bmRequestType`. Поле `wIndex` для интерфейса и конечной точки должно быть равно 0, а для устройства USB одному из стандартных значений из табл. 18.15.

Таблица 18.15. *Настройки режима тестирования*

Значение настройки	Описание
00h	Резерв
01h	Test_J (тестирование быстродействия)
02h	Test_K (тестирование быстродействия)
03h	Test_SE0_NAK (тестирование быстродействия)
04h	Test_Packet (тестирование волновых характеристик сигнала)
05h	Test_Force_Enable (тестирование подключений на линии)
06h—3Fh	Резерв
3Fh—BFh	Резерв
C0h—FFh	Зарезервировано для производителей

18.1.1.10. Запрос *SET_INTERFACE*

Данный запрос позволяет хосту выбирать настройки для определенного интерфейса. Формат запроса представлен в табл. 18.16.

Таблица 18.16. *Формат запроса* `SET_INTERFACE`

Поле запроса	Значение
bmRequestType	01h
bRequest	0Bh
wValue	Значение настройки
wIndex	Интерфейс
wLength	0
Данные	Не используется

В поле `wValue` следует записать значение настройки, а в поле `wIndex` — номер интерфейса. Если интерфейс не существует или отсутствует настройка, то будет сгенерирован запрос ошибки.

18.1.1.11. Запрос *SYNCH_FRAME*

Запрос предназначен для установки и сообщения структуры объекта синхронизации для конечной точки. Формат запроса представлен в табл. 18.17.

Таблица 18.17. *Формат запроса* `SYNCH_FRAME`

Поле запроса	Значение
bmRequestType	82h
bRequest	0Ch
wValue	0
wIndex	Конечная точка
wLength	2
Данные	Номер кадра

Данный запрос выполняется для конечной точки, которая поддерживает изохронный обмен данными и позволяет синхронизировать ее работу относительно хоста. В поле `wIndex` следует записать номер конечной точки. В результате выполнения запроса будет возвращено 2 байта (слово) с текущим номером кадра.

Вот мы и рассмотрели стандартные запросы, которые должны поддерживаться любыми устройствами, подключенными к шине USB.

18.1.2. Структура дескрипторов

Как правило, под дескриптором в данном контексте понимается заранее четко определенная структура данных, которая содержит всю необходимую информацию об устройстве USB (интерфейсе или конечной точке), необходимую хосту для взаимодействия с ним. Мы рассмотрим основной и дополнительный дескриптор устройства, а также дескрипторы конфигурации, интерфейса и конечной точки. Формат основного дескриптора устройства представлен в табл. 18.18.

Таблица 18.18. *Формат основного дескриптора устройства*

Смещение	Размер поля, байт	Наименование поля
00h	1	bLength
01h	1	bDescriptorType
02h	2	BcdUSB
04h	1	bDeviceClass
05h	1	bDeviceSubClass
06h	1	bDeviceProtocol
07h	1	bMaxPacketSize0

Таблица 18.18 *(окончание)*

Смещение	Размер поля, байт	Наименование поля
08h	2	idVendor
0Ah	2	idProduct
0Ch	2	bcdDevice
0Eh	1	iManufacturer
0Fh	1	iProduct
10h	1	iSerialNumber
11h	1	bNumConfigurations

Описание полей табл. 18.18.

- bLength — размер структуры дескриптора в байтах;
- bDescriptorType — тип дескриптора (USB_DEVICE_DESCRIPTOR_TYPE). Некоторые из существующих стандартных типов представлены в табл. 18.19;
- bcdUSB — номер версии текущего стандарта USB (на сегодня пока 2.0 или 200h), закодированный в двоично-десятичном формате (BCD);
- bDeviceClass — код класса устройства. Некоторые из стандартных значений перечислены в табл. 18.20. При использовании значения 00h каждый интерфейс в пределах конфигурации имеет собственный класс, а различные интерфейсы работают независимо;
- bDeviceSubClass — код подкласса устройства. Если поле кода класса равно 00h, то в это поле тоже должен передаваться 0;
- bDeviceProtocol — код протокола. Определяет протокол, используемый переданным классом устройства. Если значение равно 00h, то устройство не работает с протоколом, установленным для данного класса устройства. Если значение равно FFh, то устройство работает по протоколу, определенному производителем;
- bMaxPacketSize0 — максимальное значение пакета данных для нулевой конечной точки. Доступны только следующие значения: 8, 16, 32 и 64;
- idVendor — идентификатор производителя устройства. Является уникальным значением для каждого производителя и не может повторяться;
- idProduct — идентификатор продукта;
- bcdDevice — номер версии устройства в двоично-десятичном формате;
- iManufacturer — индекс смещения для строкового описания производителя;

- `iProduct` — индекс смещения для строкового описания продукта;
- `iSerialNumber` — индекс смещения для строкового описания серийного номера устройства;
- `bNumConfigurations` — число доступных конфигураций устройства.

Таблица 18.19. *Типы дескрипторов*

Тип	Значение	Описание
USB_DEVICE_DESCRIPTOR_TYPE	01h	Дескриптор устройства USB
USB_CONFIGURATION_DESCRIPTOR_TYPE	02h	Дескриптор конфигурации
USB_STRING_DESCRIPTOR_TYPE	03h	Дескриптор строки
USB_INTERFACE_DESCRIPTOR_TYPE	04h	Дескриптор интерфейса
USB_ENDPOINT_DESCRIPTOR_TYPE	05h	Дескриптор конечной точки

Таблица 18.20. *Классы устройств*

Класс	Значение	Описание
USB_DEVICE_CLASS_RESERVED	00h	Зарезервирован
USB_DEVICE_CLASS_AUDIO	01h	Устройство, работающее со звуком
USB_DEVICE_CLASS_COMMUNICATIONS	02h	Устройство связи
USB_DEVICE_CLASS_HUMAN_INTERFACE	03h	Устройство взаимодействия с человеком
USB_DEVICE_CLASS_MONITOR	04h	Устройство вывода изображения
USB_DEVICE_CLASS_PHYSICAL_INTERFACE	05h	Устройство с определенным физическим интерфейсом
USB_DEVICE_CLASS_POWER	06h	Устройство управления питанием
USB_DEVICE_CLASS_PRINTER	07h	Устройство печати
USB_DEVICE_CLASS_STORAGE	08h	Устройство хранения данных
USB_DEVICE_CLASS_HUB	09h	Разветвитель шины (хаб)
USB_DEVICE_CLASS_VENDOR_SPECIFIC	FFh	Класс устройства определяется изготовителем

Кроме основного дескриптора, существует дополнительный, который содержит информацию о быстродействии устройства. Например, если устройство работает на максимальной скорости, то дескриптор возвратит информацию

о том, как устройство могло бы работать на пониженной скорости и наоборот. Формат дополнительного дескриптора устройства представлен в табл. 18.21.

Таблица 18.21. *Формат дополнительного дескриптора устройства*

Смещение	Размер поля, байт	Наименование поля
00h	1	bLength
01h	1	bDescriptorType
02h	2	bcdUSB
04h	1	bDeviceClass
05h	1	bDeviceSubClass
06h	1	bDeviceProtocol
07h	1	bMaxPacketSize0
08h	1	bNumConfigurations
09h	1	bReserved

Описание полей табл. 18.21.

- `bLength` — размер структуры дескриптора в байтах;
- `bDescriptorType` — тип дополнительного дескриптора;
- `bcdUSB` — номер версии текущего стандарта USB (на сегодня пока 2.0 или `200h`), закодированный в двоично-десятичном формате (BCD);
- `bDeviceClass` — код класса;
- `bDeviceSubClass` — код подкласса;
- `bDeviceProtocol` — код протокола;
- `bMaxPacketSize0` — максимальный размер пакета данных для другой скорости;
- `bNumConfigurations` — число конфигураций для другой скорости;
- `bReserved` — значение поля зарезервировано и должно быть равно 0.

Как видите, большинство полей аналогичны полям основного дескриптора. Информация о производителе и продукте отсутствует, поскольку она одинакова для выбранного устройства USB, независимо от скорости обмена данными. Для получения дополнительного дескриптора следует выполнить запрос `GET_DESCRIPTOR`, передав значение `06h` (см. табл. 18.4).

Следующий тип дескриптора, о котором мы поговорим, описывает конфигурацию устройства USB. Может существовать более одного дескриптора кон-

фигурации для определенного устройства USB. В свою очередь, каждая конфигурация может состоять более чем из одного интерфейса, а интерфейс иметь несколько или ни одной конечной точки. Формат дескриптора конфигурации представлен в табл. 18.22.

Таблица 18.22. *Формат дескриптора конфигурации*

Смещение	Размер поля, байт	Наименование поля
00h	1	bLength
01h	1	bDescriptorType
02h	2	wTotalLength
04h	1	bNumInterfaces
05h	1	bConfigurationValue
06h	1	iConfiguration
07h	1	bmAttributes
08h	1	bMaxPower

Описание полей табл. 18.22.

- `bLength` — размер структуры дескриптора в байтах;
- `bDescriptorType` — тип дескриптора (`USB_CONFIGURATION_DESCRIPTOR_TYPE`);
- `wTotalLength` — содержит общую длину данных (в байтах), которая будет возвращена для дескриптора конфигурации. Сюда включена длина самой конфигурации, интерфейса, конечной точки и данных, определенных производителем устройства;
- `bNumInterfaces` — число интерфейсов, поддерживаемых конфигурацией;
- `bConfigurationValue` — код конфигурации, используемый в стандартном запросе `SET_CONFIGURATION` для установки данной конфигурации;
- `iConfiguration` — смещение для строкового описания конфигурации;
- `bmAttributes` — дополнительные свойства конфигурации. Биты 0—4 и 7 зарезервированы и равны 0. Бит 5 определяет, доступен ли устройству внешний сигнал на пробуждение (0 — нет, 1— доступен). Бит 6 определяет управление питанием (0 — по шине USB, 1— свой источник питания);
- `bMaxPower` — максимальное значение тока, потребляемое устройством USB в мА с шагом в 2 мА (например, значение 20 соответствует 40 мА).

А теперь немного поговорим о дескрипторе интерфейса. Стандартный дескриптор содержит описание интерфейса в пределах определенной конфигурации. Конфигурация поддерживает один или несколько интерфейсов,

каждый из которых включает в себя дескриптор конечной точки. Запрос GET_CONFIGURATION возвращает дескриптор интерфейса, а за ним дескриптор конечной точки. Следует помнить, что дескриптор интерфейса нельзя получить непосредственно запросом GET_DESCRIPTOR, а только через дескриптор конфигурации. Формат дескриптора интерфейса представлен в табл. 18.23.

Таблица 18.23. *Формат дескриптора интерфейса*

Смещение	Размер поля, байт	Наименование поля
00h	1	bLength
01h	1	bDescriptorType
02h	1	bInterfaceNumber
03h	1	bAlternateSetting
04h	1	bNumEndpoints
05h	1	bInterfaceClass
06h	1	bInterfaceSubClass
07h	1	bInterfaceProtocol
08h	1	iInterface

Описание полей табл. 18.23.

- bLength — размер структуры дескриптора в байтах;
- bDescriptorType — тип дескриптора (USB_INTERFACE_DESCRIPTOR_TYPE);
- bInterfaceNumber — номер интерфейса в пределах конфигурации. Нумерация начинается с 0;
- bAlternateSetting — значение данного поля содержит дополнительные настройки интерфейса, номер которого задан в поле bInterfaceNumber;
- bNumEndpoints — количество поддерживаемых интерфейсом конечных точек, исключая нулевую. При использовании интерфейсом нулевой конечной точки значение этого поля равно 0;
- bInterfaceClass — код класса интерфейса. Нулевое значение зарезервировано, а значение FFh определяет код класса, заданный производителем устройства;
- bInterfaceSubClass — код подкласса интерфейса. Если значение кода класса равно 0, то в данном поле также должен быть 0;
- bInterfaceProtocol — код протокола. Значение данного поля зависит от кодов класса и подкласса интерфейса. Значение 0 информирует о том, что

интерфейс не поддерживает протокол данного класса устройства, а значение FFh определяет работу интерфейса по протоколу, заданному производителем для данного класса устройства;

- iInterface — индекс смещения для строкового описания интерфейса.

Как уже говорилось ранее, интерфейс поддерживает от одной до нескольких конечных точек, каждая из которых имеет свой дескриптор. Он содержит информацию о пропускной способности конечной точки и является частью данных конфигурации, возвращаемых с помощью запроса GET_CONFIGURATION. Нельзя получить дескриптор конечной точки непосредственно через запрос GET_DESCRIPTOR. Кроме того, для нулевой конечной точки дескриптора не существует. Формат дескриптора конечной точки представлен в табл. 18.24.

Таблица 18.24. *Формат дескриптора конечной точки*

Смещение	Размер поля, байт	Наименование поля
00h	1	bLength
01h	1	bDescriptorType
02h	1	bEndpointAddress
03h	1	bmAttributes
04h	2	wMaxPacketSize
06h	1	bInterval

Описание полей табл. 18.24.

- bLength — размер структуры дескриптора в байтах;
- bDescriptorType — тип дескриптора (USB_ENDPOINT_DESCRIPTOR_TYPE);
- bEndpointAddress — адрес конечной точки для устройства USB. Биты 3—0 содержат номер конечной точки. Биты 6—4 зарезервированы и должны быть равны 0. Бит 7 определяет направление передачи данных (0 — от хоста к конечной точке, 1— от конечной точки к хосту). Значение бита 7 не используется для контрольной конечной точки;
- bmAttributes — описывает свойства конечной точки. Биты 7 и 6 не используются и должны быть равны 0. Биты 5—4 определяют вариант использования конечной точки (00h — конечная точка данных, 01h — конечная точка обратной связи, 02h — конечная точка для неявной обратной связи, 03h — резерв). Биты 3—2 определяют тип синхронизации (00h — без синхронизации, 01h — асинхронный, 02h — адаптивный, 03h — синхронный). Биты 1—0 задают свойства конечной точки (00h — контроль-

ная, 01h — изохронная, 02h — для передачи данных, 03h — для прерываний);

- wMaxPacketSize — максимальный размер пакета данных, поддерживаемый конечной точкой. Биты 15—13 зарезервированы и должны быть установлены в 0. Биты 12—11 определяют количество транзакций в одном фрейме (00h — одна транзакция, 01h — две, 02h — три, 03h — резерв). Биты 10—0 содержат максимальный размер пакета данных в байтах;
- bInterval — интервал опроса конечной точки (в миллисекундах) при передаче данных.

И последний дескриптор, который мы рассмотрим, описывает параметры строки. Он не является обязательным и в этом случае возвращаемые данные равны 0. Данные дескриптора строки заданы в формате Unicode и поддерживают все языки, определяемые этим форматом. Структура дескриптора строки представлена в табл. 18.25.

Таблица 18.25. *Формат дескриптора строки*

Смещение	Размер поля, байт	Наименование поля
00h	1	bLength
01h	1	bDescriptorType
02h	N	bString

Описание полей табл. 18.25.

- bLength — размер структуры дескриптора в байтах;
- bDescriptorType — тип дескриптора (USB_STRING_DESCRIPTOR_TYPE);
- bString — строка в формате Unicode.

Поскольку устройство USB может поддерживать несколько языков, для получения дескриптора строки следует в запрос передавать идентификатор языка (Language ID). Формат идентификатора языка показан в табл. 18.26.

Таблица 18.26. *Формат идентификатора языка*

Смещение	Размер поля, байт	Наименование поля
00h	1	Blength
01h	1	bDescriptorType
02h	2	wLANGID[0]
N	2	wLANGID[x]

Описание табл. 18.26.

- `bLength` — размер структуры дескриптора в байтах;
- `bDescriptorType` — тип дескриптора (`USB_STRING_DESCRIPTOR_TYPE`);
- `wLANGID` — идентификатор языка.

Нумерация идентификаторов языка начинается с 0. Структура идентификатора языка состоит из следующих данных: биты 9—0 определяют первичный идентификатор языка, а биты 15—10 — дополнительный идентификатор диалекта. Примеры некоторых идентификаторов языков представлены в табл. 18.27.

***Таблица 18.27.** Идентификаторы языков*

Идентификатор	Описание
0x042B	Армянский
0x0423	Белорусский
0x0409	Английский (США)
0x0407	Немецкий
0x0426	Латвийский
0x0415	Польский
0x0419	Русский
0x0422	Украинский

18.1.3. Использование запросов

Для программирования запросов нам понадобятся файлы определений (в частности hidsdi.h), входящие в состав пакета для разработки драйверов (*DDK* — Driver Developers Kits). Данный пакет можно свободно скачать с официального сайта фирмы Microsoft. Кроме того, необходимо будет добавить в опциях компоновщика ссылку на библиотеки Hid.lib и Setupapi.lib. Чтобы упростить код, вынесем в отдельный файл необходимые для поддержки USB константы и структуры и назовем его usbdefs.h. Полное описание этого файла представлено в листинге 18.1.

Листинг 18.1. Файл поддержки USB-устройств usbdefs.h

```
// подключаем файл определений из пакета DDK
extern "C"
{
   #include "hidsdi.h"
}
```

```
// базовые коды ввода-вывода
#include <winioctl.h>
// максимальная длина строки описания
#define MAXIMUM_USB_STRING_LENGTH 255
// типы дескрипторов
#define USB_CONFIGURATION_DESCRIPTOR_TYPE   0x02
#define USB_STRING_DESCRIPTOR_TYPE          0x03
#define USB_INTERFACE_DESCRIPTOR_TYPE       0x04
// типы классов для устройств USB
#define USB_DEVICE_CLASS_RESERVED           0x00
#define USB_DEVICE_CLASS_AUDIO              0x01
#define USB_DEVICE_CLASS_COMMUNICATIONS     0x02
#define USB_DEVICE_CLASS_HUMAN_INTERFACE    0x03
#define USB_DEVICE_CLASS_MONITOR            0x04
#define USB_DEVICE_CLASS_PHYSICAL_INTERFACE 0x05
#define USB_DEVICE_CLASS_POWER              0x06
#define USB_DEVICE_CLASS_PRINTER            0x07
#define USB_DEVICE_CLASS_STORAGE            0x08
#define USB_DEVICE_CLASS_HUB                0x09
#define USB_DEVICE_CLASS_VENDOR_SPECIFIC    0xFF
// коды ввода-вывода для функции DeviceIoControl
#define FILE_DEVICE_USB      FILE_DEVICE_UNKNOWN
#define USB_IOCTL_INDEX      0x00ff
#define IOCTL_USB_GET_NODE_INFORMATION   CTL_CODE ( FILE_DEVICE_USB,  \
                USB_IOCTL_INDEX + 3, METHOD_BUFFERED, FILE_ANY_ACCESS )
#define IOCTL_USB_GET_ROOT_HUB_NAME  CTL_CODE ( FILE_DEVICE_USB,  \
                USB_IOCTL_INDEX + 3, METHOD_BUFFERED, FILE_ANY_ACCESS )
#define IOCTL_USB_GET_NODE_CONNECTION_INFORMATION
    CTL_CODE ( FILE_DEVICE_USB, USB_IOCTL_INDEX + 4, METHOD_BUFFERED,  \
                                                      FILE_ANY_ACCESS )
#define IOCTL_USB_GET_DESCRIPTOR_FROM_NODE_CONNECTION
    CTL_CODE ( FILE_DEVICE_USB, USB_IOCTL_INDEX + 5, METHOD_BUFFERED,  \
                                                      FILE_ANY_ACCESS )
#define IOCTL_USB_GET_NODE_CONNECTION_NAME CTL_CODE ( FILE_DEVICE_USB,  \
                USB_IOCTL_INDEX + 6, METHOD_BUFFERED, FILE_ANY_ACCESS )
#define IOCTL_USB_GET_NODE_CONNECTION_DRIVERKEY_NAME
    CTL_CODE ( FILE_DEVICE_USB, USB_IOCTL_INDEX + 9, METHOD_BUFFERED,  \
                                                      FILE_ANY_ACCESS )
#define IOCTL_GET_HCD_DRIVERKEY_NAME CTL_CODE ( FILE_DEVICE_USB,  \
                USB_IOCTL_INDEX + 10, METHOD_BUFFERED, FILE_ANY_ACCESS )
// структуры данных
#pragma pack ( 1 )
```

```
// описатель дескриптора устройства
typedef struct _USB_INTERFACE_DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
   UCHAR bInterfaceNumber;
   UCHAR bAlternateSetting;
   UCHAR bNumEndpoints;
   UCHAR bInterfaceClass;
   UCHAR bInterfaceSubClass;
   UCHAR bInterfaceProtocol;
   UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR;
// дополнительный описатель дескриптора устройства
typedef struct _USB_INTERFACE_DESCRIPTOR2 {
   UCHAR  bLength;
   UCHAR  bDescriptorType;
   UCHAR  bInterfaceNumber;
   UCHAR  bAlternateSetting;
   UCHAR  bNumEndpoints;
   UCHAR  bInterfaceClass;
   UCHAR  bInterfaceSubClass;
   UCHAR  bInterfaceProtocol;
   UCHAR  iInterface;
   USHORT wNumClasses;
} USB_INTERFACE_DESCRIPTOR2, *PUSB_INTERFACE_DESCRIPTOR2;
// описание конфигурации
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
   USHORT wTotalLength;
   UCHAR bNumInterfaces;
   UCHAR bConfigurationValue;
   UCHAR iConfiguration;
   UCHAR bmAttributes;
   UCHAR MaxPower;
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR;
// описатель основного дескриптора устройства
typedef struct _USB_COMMON_DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
} USB_COMMON_DESCRIPTOR, *PUSB_COMMON_DESCRIPTOR;
// описатель корневого устройства
typedef struct _USB_ROOT_HUB_NAME {
```

```
   ULONG ActualLength;
   WCHAR RootHubName[1];
} USB_ROOT_HUB_NAME, *PUSB_ROOT_HUB_NAME;
// параметры подключенного к хабу устройства
typedef struct _USB_NODE_CONNECTION_NAME {
   ULONG ConnectionIndex;
   ULONG ActualLength;
   WCHAR NodeName[1];
} USB_NODE_CONNECTION_NAME, *PUSB_NODE_CONNECTION_NAME;
// описание драйвера устройства
typedef struct _USB_NODE_CONNECTION_DRIVERKEY_NAME {
   ULONG ConnectionIndex;
   ULONG ActualLength;
   WCHAR DriverKeyName[1];
} USB_NODE_CONNECTION_DRIVERKEY_NAME,
 *PUSB_NODE_CONNECTION_DRIVERKEY_NAME;
// описание дескриптора
typedef struct _USB_STRING_DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
   WCHAR bString[1];
} USB_STRING_DESCRIPTOR, *PUSB_STRING_DESCRIPTOR;
// описание подключенного устройства
typedef struct _STRING_DESCRIPTOR_NODE
{
   struct _STRING_DESCRIPTOR_NODE *Next;
   UCHAR DescriptorIndex;
   USHORT LanguageID;
   USB_STRING_DESCRIPTOR StringDescriptor[0];
} STRING_DESCRIPTOR_NODE, *PSTRING_DESCRIPTOR_NODE;
// описание дескриптора хаба
typedef struct _USB_HUB_DESCRIPTOR {
   UCHAR bDescriptorLength;
   UCHAR bDescriptorType;
   UCHAR bNumberOfPorts;
   USHORT wHubCharacteristics;
   UCHAR bPowerOnToPowerGood;
   UCHAR bHubControlCurrent;
   UCHAR bRemoveAndPowerMask[64];
} USB_HUB_DESCRIPTOR, *PUSB_HUB_DESCRIPTOR;
// дополнительная информация о хабе
typedef struct _USB_HUB_INFORMATION {
   USB_HUB_DESCRIPTOR HubDescriptor;
```

```
   BOOLEAN HubIsBusPowered;
} USB_HUB_INFORMATION, *PUSB_HUB_INFORMATION;
typedef struct _USB_MI_PARENT_INFORMATION {
   ULONG NumberOfInterfaces;
} USB_MI_PARENT_INFORMATION, *PUSB_MI_PARENT_INFORMATION;
// параметры драйвера устройства
typedef struct _USB_HCD_DRIVERKEY_NAME {
   ULONG ActualLength;
   WCHAR DriverKeyName[1];
} USB_HCD_DRIVERKEY_NAME, *PUSB_HCD_DRIVERKEY_NAME;
// описание хаба
typedef enum _USB_HUB_NODE
{
    UsbHub,
    UsbMIParent
} USB_HUB_NODE;

typedef struct _USB_NODE_INFORMATION
{
   USB_HUB_NODE NodeType;
   union {
     USB_HUB_INFORMATION HubInformation;
     USB_MI_PARENT_INFORMATION MiParentInformation;
   } u;
} USB_NODE_INFORMATION, *PUSB_NODE_INFORMATION;
// параметры устройства
typedef struct _USB_DEVICE_DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
   USHORT bcdUSB;
   UCHAR bDeviceClass;
   UCHAR bDeviceSubClass;
   UCHAR bDeviceProtocol;
   UCHAR bMaxPacketSize0;
   USHORT idVendor;
   USHORT idProduct;
   USHORT bcdDevice;
   UCHAR iManufacturer;
   UCHAR iProduct;
   UCHAR iSerialNumber;
   UCHAR bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
```

```
// состояние соединения
typedef enum _USB_CONNECTION_STATUS {
   NoDeviceConnected,
   DeviceConnected,
   DeviceFailedEnumeration,
   DeviceGeneralFailure,
   DeviceCausedOvercurrent,
   DeviceNotEnoughPower,
   DeviceNotEnoughBandwidth
} USB_CONNECTION_STATUS, *PUSB_CONNECTION_STATUS;
// дополнительная информация о подключенном устройстве
typedef struct _USB_ENDPOINT_DESCRIPTOR {
   UCHAR bLength;
   UCHAR bDescriptorType;
   UCHAR bEndpointAddress;
   UCHAR bmAttributes;
   USHORT wMaxPacketSize;
   UCHAR bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;
typedef struct _USB_PIPE_INFO {
   USB_ENDPOINT_DESCRIPTOR EndpointDescriptor;
   ULONG ScheduleOffset;
} USB_PIPE_INFO, *PUSB_PIPE_INFO;
typedef struct _USB_NODE_CONNECTION_INFORMATION {
   ULONG ConnectionIndex;
   USB_DEVICE_DESCRIPTOR DeviceDescriptor;
   UCHAR CurrentConfigurationValue;
   BOOLEAN LowSpeed;
   BOOLEAN DeviceIsHub;
   USHORT DeviceAddress;
   ULONG NumberOfOpenPipes;
   USB_CONNECTION_STATUS ConnectionStatus;
   USB_PIPE_INFO PipeList[0];
} USB_NODE_CONNECTION_INFORMATION, *PUSB_NODE_CONNECTION_INFORMATION;
// текущее состояние устройства
typedef struct _USB_DESCRIPTOR_REQUEST {
   ULONG ConnectionIndex;
   struct {
     UCHAR bmRequest;
     UCHAR bRequest;
     USHORT wValue;
     USHORT wIndex;
     USHORT wLength;
    } SetupPacket;
```

```
   UCHAR Data[0];
} USB_DESCRIPTOR_REQUEST, *PUSB_DESCRIPTOR_REQUEST;
// общие параметры устройства
typedef struct
{
   PCHAR HubName;
   PUSB_NODE_INFORMATION HubInfo;
   PUSB_NODE_CONNECTION_INFORMATION ConnectionInfo;
   PUSB_DESCRIPTOR_REQUEST ConfigDesc;
   PSTRING_DESCRIPTOR_NODE StringDescs;
} USBDEVICEINFO, *PUSBDEVICEINFO;
#pragma pack ( )
```

Теперь, когда все основные структуры данных определены, создадим новый класс для работы с устройствами на шине USB. Чтобы продемонстрировать основные принципы программирования данного типа оборудования, мы рассмотрим два базовых метода, совмещенных в одном классе. Назовем наш класс USB и создадим два файла USB.h (файл определений) и USB.cpp (файл реализации методов и функций класса). Поскольку листинги файлов довольно громоздки, в тексте книги они не приводятся. Вы можете найти их на прилагаемом к книге компакт-диске.

Реализация поддержки устройств USB в представленном классе осуществляется двумя способами: посредством набора функций менеджера установки устройств и с помощью универсальной функции ввода-вывода `DeviceIoControl`. Рассмотрим каждый из них более подробно.

В первом случае вызывается функция `_getDevicesParams`. Она находит все устройства определенного класса и выделяет их основные параметры (идентификаторы производителя и изделия) для дальнейшего доступа. Чтобы явно указать класс USB-устройств, мы применили специальную функцию `HidD_GetHidGuid`, которая предназначена для определения уникального глобального идентификатора (GUID), поддерживающего все устройства с интерфейсом USB. Далее мы передали GUID в функцию `SetupDiGetClassDevs` для получения информации об устройствах указанного класса (в нашем случае — USB). На следующем этапе, периодически вызывая функцию `SetupDiEnumDeviceInterfaces`, определяем параметры всех имеющихся в системе USB-устройств. Для этого дополнительно применяем функцию `SetupDiGetDeviceInterfaceDetail`. Главная ее задача состоит в возвращении пути к найденному устройству. Используя путь, открываем устройство (`CreateFile`) и получаем основные параметры посредством функции `HidD_GetAttributes`. Дополнительно обрабатываем информацию о производителе и серийном номере устройства (`HidD_GetManufacturerString`, `HidD_`

GetProductString и HidD_GetSerialNumberString). И в завершении закрываем текущее устройство и переходим к поиску следующего. Когда все устройства найдены, можем приступать к непосредственной работе с ними. Сначала следует проанализировать возможности устройства, особенно если вы планируете выполнять обмен данными. Эта задача решается в функции GetDeviceCaps. Вызываем HidD_GetPreparsedData (предварительная подготовка данных об устройстве) и в случае успешного завершения определяем непосредственно поддерживаемые устройством возможности (HidP_GetCaps). В данном случае нас интересуют только три параметра: размер данных для управления свойствами устройства (FeatureReportByteLength), минимальные размеры в байтах входного (InputReportByteLength) и выходного (OutputReportByteLength) буферов для обмена данными. Для получения и установки свойств используются соответственно функции HidD_GetFeature и HidD_SetFeature. Обмен данными с устройствами USB осуществляется с помощью стандартных файловых функций Win32 — ReadFile и WriteFile. Вот в принципе и все базовые сведения. Естественно, для организации передачи или приема данных необходимо иметь представление о применяемом в USB протоколе и правилах форматирования последовательности байтов.

Второй способ основан на использовании функции DeviceIoControl. Принцип работы этой функции базируется на прямом взаимодействии программы и устройства (драйвера) посредством специальных управляющих кодов. Например, для сброса порта USB служит код IOCTL_INTERNAL_USB_RESET_PORT. Полный список всех поддерживаемых кодов можно посмотреть в файле usbioctl.h, входящем в состав пакета DDK. Кроме того, посредством функции DeviceIoControl можно выполнять стандартные запросы для устройств USB. Посмотрите пример, представленный в листинге 18.2.

Листинг 18.2. Использование стандартных запросов

```
#define CTL_CODE( DeviceType, Function, Method, Access ) ( \
((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
#define METHOD_BUFFERED 0
#define FILE_ANY_ACCESS 0
#define FILE_DEVICE_UNKNOWN 0x00000022
#define USBIO_IOCTL_VENDOR 0x0800 // Vendor defined
#define IOCTL_USBIO_WRITE_PACKET CTL_CODE(FILE_DEVICE_UNKNOWN, \
                                          USBIO_IOCTL_VENDOR + 10, \
                                    METHOD_BUFFERED, \ FILE_ANY_ACCESS)
// структура для описания пакетной команды
struct ioPacket
{
       unsigned char Recipient;
       unsigned char DeviceModel;
```

```
        unsigned char MajorCmd;
        unsigned char MinorCmd;
        unsigned char DataMSB;
        unsigned char DataLSB;
        unsigned short Length;
} VENDORPACKET, *PVENDORPACKET;
// функция для обработки команды
int UsbCmd( PVENDORPACKET pPacket )
{
        ULONG nBytes;
        BOOLEAN bSuccess;
        // выполняем команду
        bSuccess = DeviceIoControl ( hUsb, IOCTL_USBIO_WRITE_PACKET,
                        pPacket, sizeof( VENDORPACKET ), pPacket,
                        sizeof( VENDORPACKET ), &nBytes, NULL) ;

        if(!bSuccess )
        {
                return NULL;
        }
        else
                return nBytes;
}
```

Все перечисленные способы программирования устройств USB используют специальный системный драйвер. Чтобы получить прямой доступ к устройствам посредством портов ввода-вывода, понадобится предварительно собрать информацию об адресах, номерах прерываний и других выделенных системой ресурсов. Сделать это можно с помощью кода, рассмотренного в *главе 1*, поскольку все необходимое находится в реестре.

В завершении мы рассмотрим универсальный хост-контроллер USB фирмы Intel (*UHCI* — Universal Host Controller Interface). Следует заметить, что работать с контроллером можно как напрямую посредством портов, так и через управляющие программные модули (драйверы). Второй вариант более предпочтительней, поскольку отвечает концепции работы с оборудованием, принятым в операционных системах Windows. Контроллер предоставляет базовый набор регистров. Они делятся на две группы: регистры ввода-вывода и регистры конфигурации для устройств на шине PCI. Первая группа отвечает за отображение адресов ввода-вывода и статуса в адресном пространстве шины PCI. Адреса регистров даны в виде смещения относительно базового адреса. Список регистров ввода-вывода представлен в табл. 18.28.

Таблица 18.28. *Список регистров ввода-вывода*

Адрес	Описание регистра
Базовый адрес + (00h—01h)	USB Command (регистр команд)
Базовый адрес + (02h—03h)	USB Status (регистр состояния)
Базовый адрес + (04h—05h)	USB Interrupt Enable (регистр управления прерываниями)
Базовый адрес + (06h—07h)	Frame Number (регистр счетчика фреймов)
Базовый адрес + (08h—0Bh)	Frame List Base Address (регистр адресов списка фреймов)
Базовый адрес + 0Ch	Start Of Frame Modify (регистр стартового фрейма)
Базовый адрес + (10h—11h)	Port 1 Status/Control (регистр управления и статуса для порта 1)
Базовый адрес + (12h—13h)	Port 2 Status/Control (регистр управления и статуса для порта 2)

Вторая группа необходима для представления в пространстве шины PCI адресных регистров для поддержки устройств USB. Следует заметить, что контроллер не взаимодействует с шиной PCI, а лишь предоставляет ей адреса для обработки их счетчиком PCI. Счетчик идентифицирует все доступные устройства в пределах пространства шины PCI и распределяет системные ресурсы для них. Список регистров конфигурации показан в табл. 18.29.

Таблица 18.29. *Список регистров конфигурации*

Адрес смещения PCI	Описание регистра
00h—08h	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI
09h—0Bh	Регистр кода класса
0Ch—1Fh	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI
20h—23h	Регистр адресов
24h—5Fh	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI
60h	Регистр номера версии последовательной шины
61h—FFh	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI

А теперь подробнее рассмотрим все эти регистры.

18.1.4. Регистры ввода-вывода

Основная часть адресов этих регистров выбирается в регистр конфигурации шины PCI. Часть регистров позволяет читать текущие состояния портов, но не обязательно записывают значения в порт. Программа должна проверить состояние регистра и определить момент, когда порт освободится и можно будет передать данные. Это сделано для того, чтобы не нарушить функционирование шины PCI и подключенных к ней устройств.

18.1.4.1. Регистр команд

В первую очередь этот регистр сообщает о том, что команда будет выполнена хост-контроллером. Запись в регистр значения инициирует выполнение команды. Регистр поддерживает дополнительные биты для выполнения, остановки команды, а также для отладки. Регистр доступен для чтения и записи. Значение по умолчанию равно `0000h`. Формат регистра представлен в табл. 18.30.

Таблица 18.30. *Формат регистра команд*

Биты	Описание
15—8	Зарезервированы
7	Размер пакета данных
6	Флаг конфигурации
5	Режим отладки
4	Общее управление
3	Suspend mode
2	Общий сброс
1	Сброс контроллера
0	Управление выполнением команды

Приведем подробное описание табл. 18.30.

- Биты 15—8 зарезервированы и не используются.
- Бит 7 определяет максимальный размер пакета данных в байтах (0 — 32 байта, 1 — 64 байта).
- Бит 6 задает флаг конфигурации. Данный флаг не влияет на контроллер, а лишь помогает программному обеспечению сообщить о завершении процесса конфигурации, установив бит в 1.

- Бит 5 управляет режимом отладки. Установка бита в 1 включает режим отладки, иначе (значение 0) контроллер переходит в нормальный режим работы.
- Бит 4 определяет общее управление устройствами USB. Установка бита в 1 вынуждает контроллер передать общий сигнал готовности для работы. Программное обеспечение должно по истечении 20 мс установить бит в 0, тем самым сообщив контроллеру, что устройства USB готовы к работе. Контроллер может устанавливать данный бит в различных ситуациях: подключение или отключение устройства, переход в режим Suspend mode.
- Бит 3 управляет общим сигналом режима Suspend mode. Установка бита в 1 включает режим Suspend mode для всех устройств USB. Для выхода из этого режима программное обеспечение должно установить бит в 0.
- Бит 2 управляет общим сбросом всех регистров и устройств. Установка бита в 1 инициирует сброс устройств USB и перезагружает логику, включая внутренние регистры хаба. Программное обеспечение устанавливает бит в 0 не ранее, чем через 10 мс.
- Бит 1 управляет сбросом самого хост-контроллера. Установка его в 1 выполняет сброс внутренних счетчиков, таймеров и другой логики контроллера. При этом обмен данными с устройствами USB немедленно прекращается.
- Бит 0 определяет выполнение команды. Установка бита в 1 указывает на выполнение команды. Установка бита в 0 заставляет хост завершить выполнение текущей команды и остановиться.

18.1.4.2. Регистр состояния

Регистр позволяет проследить текущие прерывания и различные состояния хост-контроллера. Он доступен для чтения и записи. Запись значения 1 очищает (устанавливает в 0) соответствующий бит. Значение по умолчанию равно `0000h`. Формат регистра представлен в табл. 18.31.

Таблица 18.31. *Формат регистра состояния*

Биты	Описание
15—6	Зарезервированы
5	Состояние выполнения команды
4	Признак ошибки в процессе обработки хостом дескриптора
3	Признак системной ошибки
2	Признак состояния Suspend mode

Таблица 18.31 *(окончание)*

Биты	Описание
1	Признак ошибки прерывания на устройстве USB
0	Состояние прерывания устройства USB

Приведем краткое описание табл. 18.31.

- Биты 15—6 зарезервированы и не используются.
- Бит 5 определяет состояние выполнения команды. Хост записывает сюда значение 1 после того, как выполнение команды было остановлено (записано значение 0 в бит 0 регистра команд) программным обеспечением или самим хост-контроллером, например, для отладки.
- Бит 4 устанавливается хостом в 1, когда возникает неустранимая ошибка. При этом хост сбрасывает бит 0 командного регистра, чтобы прекратить дальнейшую обработку команды. Генерируется аппаратное прерывание.
- Бит 3 устанавливается хостом в 1, если происходит серьезная ошибка, связанная с системным доступом к хост-контроллеру. При этом хост очищает бит 0 командного регистра. Генерируется аппаратное прерывание.
- Бит 2 устанавливается хостом в 1, когда получает сигнал Suspend mode от устройства USB. Значение бита корректно только в том случае, если хост находится в режиме Suspend mode (бит 3 в регистре команд установлен в 1).
- Бит 1 устанавливается хостом в 1, когда выполнение транзакции на устройстве USB приводит в ошибке (например, переполнение счетчика).
- Бит 0 устанавливается хостом в 1, когда прерывание вызвано завершением транзакции на устройстве USB или обнаружен неполный пакет данных.

18.1.4.3. Регистр управления прерываниями

Данный регистр позволяет включать или отключать запросы на прерывания для программного обеспечения. Когда прерывания разрешены, генерируется прерывание для хоста. Регистр доступен для чтения и записи. Значение по умолчанию равно `0000h`. Формат регистра представлен в табл. 18.32.

Таблица 18.32. *Формат регистра управления прерываниями*

Биты	Описание
15—4	Зарезервированы
3	Доступность прерывания для пакета данных

Таблица 18.32 *(окончание)*

Биты	Описание
2	Доступность завершения прерывания
1	Доступность состояния прерывания
0	Доступность прерывания по времени и контрольной сумме

Приведем краткое описание таблицы.

- Биты 15—4 зарезервированы и не используются.
- Бит 3 определает работу прерывания при передаче короткого пакета данных. Установка бита в 1 включает, а установка в 0 выключает прерывание.
- Бит 2 определяет выдачу прерывания на завершение передачи (1 — включено, 0 — выключено).
- Бит 1 управляет прерыванием для режима Suspend mode (1 — включено, 0 — выключено).
- Бит 0 управляет прерыванием для ситуаций, когда истекает время выполнения операции, и для результата проверки контрольной суммы пакета данных.

18.1.4.4. Регистр счетчика фреймов

Служит для анализа числа фреймов, передаваемых в пакете данных. Обновляется после каждого завершения передачи списка фреймов. Нельзя писать в регистр, когда хост остановлен. Регистр доступен для чтения и записи. В режиме записи следует передавать только 16-разрядное (2-байтовое) значение, иначе возникнет неопределенная ошибка. Значение по умолчанию равно `0000h`. Формат регистра представлен в табл. 18.33.

Таблица 18.33. *Формат регистра счетчика фреймов*

Биты	Описание
15—11	Зарезервированы
10—0	Список текущих индексов или номеров фреймов. Содержат число фреймов в списке. Значение увеличивается примерно через каждую 1 мс

18.1.4.5. Регистр адресов для списка фреймов

Данный регистр содержит базовый адрес списка фреймов в системной памяти. Является 32-разрядным. Регистр загружается перед началом выполнения команды хост-контроллером. При этом используются только старшие

20 битов, а оставшиеся 12 устанавливаются в 0 для выравнивания до 2 байтов. Содержание регистра объединено со счетчиком количества фреймов для использования возможности последовательной обработки списка фреймов. Поддерживаются до 1024 списков. Регистр доступен для чтения и записи. Формат регистра представлен в табл. 18.34.

Таблица 18.34. *Формат регистра адресов для списка фреймов*

Биты	Описание
31—12	Базовый адрес в памяти
11—0	Зарезервированы и должны быть установлены в 0

18.1.4.6. Регистр стартового фрейма

Данный регистр имеет размер 1 байт и служит для генерации стартового фрейма пакета данных. Каждое новое значение, записанное в регистр, позволяет синхронизировать следующий фрейм. Таким образом, регистр может применяться для корректировки счетчика фреймов относительно системных часов. Регистр доступен для чтения и записи. Формат регистра представлен в табл. 18.35.

Таблица 18.35. *Формат регистра стартового фрейма*

Биты	Описание
7	Зарезервирован
6—0	Значение для синхронизации стартового фрейма

18.1.4.7. Регистры управления и статуса

Два однотипных 16-разрядных регистра для 1 и 2 портов хост-контроллера. Служат для отражения состояния и управления хост-контроллером. Регистры доступны для чтения и записи. Запись следует производить только 2-байтовым (слово) значением. По умолчанию значение регистра равно `0080h`. Регистры отключены после выключения питания, общего сброса или сброса хост-контроллера. Формат регистра представлен в табл. 18.36.

Таблица 18.36. *Формат регистров управления и статуса*

Биты	Описание
15—13	Зарезервированы
12	Режим Suspend mode

Таблица 18.36 *(окончание)*

Биты	Описание
11—10	Зарезервированы и должны быть установлены в 0 при записи
9	Сброс порта
8	Тип подключенного устройства (по скорости)
7	Зарезервирован
6	Состояние для режима Suspend mode
5—4	Статус линии
3	Контроль изменений порта
2	Контроль включения порта
1	Состояние соединения
0	Текущее состояние соединения

Рассмотрим краткое описание табл. 18.36.

- Биты 15—13 зарезервированы и должны быть установлены в 0 при записи.
- Бит 12 доступен для чтения и записи и определяет состояние режима Suspend mode (1 — работа порта приостановлена, 0 — порт работает).
- Биты 11—10 зарезервированы.
- Бит 9 управляет сбросом порта (1 — порт был сброшен, 0 — порт не был сброшен).
- Бит 8 определяет тип подключенного устройства USB (1 — подключено низкоскоростное устройство, 0 — подключено высокоскоростное устройство). Бит используется только для чтения.
- Бит 7 служит только для чтения. На данный момент зарезервирован и всегда возвращает значение 1.
- Бит 6 определяет состояние порта для режима приостановки (1 — есть, 0 — нет). Запись программой значения 1 в этот бит позволяет получать состояние режима приостановки для порта.
- Биты 5—4 доступны только для чтения и определяют логический уровень сигнала на линии.
- Бит 3 позволяет определить состояние порта (1 — порт был включен, выключен и состояние изменилось, 0 — состояние не изменилось). Этот бит доступен для чтения и для записи значения сброса текущего состояния.
- Бит 2 управляет работой порта (1 — порт включен, 0 — порт выключен). Данный бит доступен для чтения и записи.

- Бит 1 определяет состояние соединения. Доступен для чтения и записи значения очистки текущего состояния (1 — есть изменение, 0 — нет изменения).
- Бит 0 определяет текущее состояние соединения и доступен только для чтения (1 — устройство подключено, 0 — устройство не подключено).

18.1.5. Регистры конфигурации

Регистры конфигурации в первую очередь необходимы для связи с пространством адресов шины PCI. Подробнее о шине PCI и ее особенностях вы можете прочитать в *главе 12*. Замечу только, что каждое устройство на этой шине должно принадлежать к определенному классу, иметь свои адреса памяти. Для предоставления этих данных и взаимодействия с шиной PCI и предназначены регистры конфигурации.

18.1.5.1. Регистр кода класса

Данный регистр доступен только для чтения и предназначен для идентификации кода класса и подкласса устройства, требуемых шиной PCI для выделения системных ресурсов. Регистр 24-разрядный. Значение по умолчанию равно `010180h`. Формат регистра представлен в табл. 18.37.

Таблица 18.37. *Формат регистра кода класса*

Биты	Описание
23—16	Базовый код класса. Для контроллера последовательной шины равен `0Ch`
15—8	Код подкласса. Для хост-контроллера USB равен `03h`
7—0	Программируемый интерфейс. Всегда равен `00h`

18.1.5.2. Регистр базового адреса

Регистр позволяет устанавливать или получать базовый адрес ввода-вывода для обмена данными с устройством USB. Регистр является 32-разрядным и доступен для чтения и записи. По умолчанию значение регистра равно `00000001h`. Формат регистра представлен в табл. 18.38.

Таблица 18.38. *Формат регистра базового адреса*

Биты	Описание
31—16	Зарезервированы и должны быть установлены в 0
15—5	Значение базового адреса

Таблица 18.38 *(окончание)*

Биты	Описание
4—1	Зарезервированы и должны быть установлены в 0
0	Доступен только для чтения и всегда равен 1

18.1.5.3. Регистр номера версии

Данный регистр является 8-разрядным и определяет номер текущей версии USB, которую поддерживает хост-контроллер. Регистр доступен только для чтения. Формат регистра представлен в табл. 18.39.

Таблица 18.39. *Формат регистра номера версии*

Биты	Описание
7—0	Номер версии для последовательной шины (00h — предварительная версия 1.0, 01h — версия 1.0, 02h — версия 2.0)

Вот и все, с чем мне хотелось познакомить читателей. Более подробную информацию о контроллере можно получить на сайте производителя (**www.intel.com**).

18.2. Интерфейс IEEE 1394

Данный интерфейс является цифровым и для обмена данными использует последовательную высокоскоростную шину. Его еще называют *FireWire* или *iLink*. Впервые был использован фирмой SONY для видеокамер стандарта mini-DV. В данный момент получил широкое распространение как в видеокамерах, так и в ноутбуках и настольных компьютерах. Как правило, для последних необходимо дополнительно приобретать соответствующий контроллер. К преимуществам интерфейса IEEE 1394 можно отнести:

- высокую скорость обмена данными. Так для расширения IEEE 1394b скорость составляет 1600 Мбит/с, а для IEEE 1394a — до 400 Мбит/с;
- поддержка "горячего" подключения устройств к шине;
- отсутствие потерь информации при обмене данными;
- гибкая настройка конфигурации оборудования;
- возможность подключения до 63 устройств одновременно;
- наличие питания прямо на шине;
- поддержка асинхронной и изохронной передачи данных.

Функционирование интерфейса происходит по 3-уровневой схеме. Уровень транзакций позволяет управлять потоками данных и поддерживает асинхронный протокол при записи или чтении. Уровень компоновки определяет формирование пакетов данных и их передачу. И наконец, физический уровень выполняет преобразование цифрового сигнала в аналоговый и наоборот, а также контроль уровня сигнала и доступа к шине. Топология шины поддерживает древовидную и последовательную структуру, а также их взаимную комбинацию.

Обмен данными между компьютером и внешними устройствами происходит посредством контроллера, поддерживающего стандарт IEEE 1394 Open Host Controller Interface (*OHCI*). Данный стандарт определяет основные свойства протокола компоновки последовательной шины, регистры управления и обмена данными. Он также включает в себя поддержку прямого доступа к памяти (DMA) для повышения эффективности передачи данных по шине и управление энергопотреблением. Контроллер позволяет передавать данные в двух режимах: асинхронном и изохронном. Для программного доступа к хосту используются следующие средства: регистры, прямой доступ к памяти DMA и прерывания. Архитектура хоста дает возможность отображать регистры в собственном адресном пространстве для последующего обращения к ним. Прямой доступ к памяти формируется двумя способами: выделением резидентной памяти для хранения структуры дескрипторов, описывающих буферы данных, и непосредственным (физическим) доступом к памяти для чтения и записи. Более подробно о структуре хоста, выделении памяти можно познакомиться в спецификации, которая находится в свободном доступе в Интернете. Здесь же мы кратко познакомимся со структурой регистров и программированием интерфейса IEEE 1394 в Windows.

18.2.1. Описание регистров

Хост включает в себя два основных типа регистров: чтения-записи и установки-очистки. Кроме того, регистры чтения-записи могут обновляться автономно хостом. Регистры чтения-записи дают возможность обращаться к ним напрямую. Регистры установки-очистки имеют свои адреса в памяти, обращение к ним сводится к посылке хосту нужного адреса. А он в свою очередь возвращает текущие значения регистров. Для них доступны следующие операции: чтение, установка значения, очистка значения и автономное обновление значения хостом. Все регистры имеют 32-разрядный формат.

18.2.1.1. Регистр контекста

Регистр относится к регистрам установки-очистки. Служит для предоставления хосту информации о дескрипторах прямого доступа к памяти (DMA).

Каждый контекст имеет регистр управления (ContextControl) и регистр указателя (CommandPtr). Формат уравляющего регистра представлен в табл. 18.40.

Таблица 18.40. *Формат управляющего регистра контекста*

Биты	Описание
31—16	Не используются
15	Устанавливает (1) или сбрасывает (0) обработку информации о дескрипторе
14—13	Не используются
12	Определяет процесс обработки хостом дескриптора (приостановить или продолжить)
11	Если возникает фатальная ошибка, то хост устанавливает бит в 1 и очищает (записывает 0), когда программа очищает бит 15
10	Хост устанавливает бит в 1, когда выполняет обработку дескриптора
9—8	Не используются
7—5	Данное битовое поле определяет текущую скорость получения пакета данных (0 — 100 Мбит/с, 1 — 200 Мбит/с, 3 — 400 Мбит/с) и неопределено, когда установлены биты 12 и 10
4—0	Биты содержат код успешной передачи пакета или код ошибки в обратном случае

Код ошибки иначе называют *кодом события* пакета данных. Некоторые из стандартных кодов представлены в табл. 18.41.

Таблица 18.41. *Коды ошибок*

Код	Наименование	Описание
00h	evt_no_status	Нет ошибки
02h	evt_long_packet	Длина принятых данных превышает длину выделенного буфера
03h	evt_missing_ack	Произошла ошибка передачи данных до подтверждения или код подтверждения вернул ошибку
04h	evt_underrun	Неполный пакет данных на линии FIFO (первым пришел, первым ушел)
05h	evt_overrun	Переполнение пакета данных на линии FIFO во время приема в изохронном режиме
06h	evt_descriptor_read	Невосстанавливаемая ошибка во время чтения хостом дескриптора
07h	evt_data_read	Произошла ошибка в момент считывания данных хостом из памяти

Таблица 18.41 *(окончание)*

Код	Наименование	Описание
08h	evt_data_write	Произошла ошибка в момент записи хостом данных в память
09h	evt_bus_reset	Пакет данных определился как пакет сброса шины
0Ah	evt_timeout	Истекло время ожидания при асинхронной передаче данных
1Dh	ack_data_error	Ошибка передачи данных (возможно связано с ошибкой в контрольной сумме)
1Eh	ack_type_error	В одно из полей заголовка пакета данных записано неверное или недопустимое значение

18.2.1.2. Регистр указателя

Регистр предназначен для получения адреса блока дескриптора. Формат уравляющего регистра представлен в табл. 18.42.

Таблица 18.42. *Формат регистра указателя*

Биты	Описание
31—4	Определяют дескриптор адреса
3—0	Определяет число непрерывных 16-байтных блоков для выравнивания значения адреса

18.2.1.3. Регистр версии

Данный 32-разрядный регистр доступен только для чтения и предоставляет информацию о текущей версии интерфейса. Формат регистра показан в табл. 18.43.

Таблица 18.43. *Формат регистра версии*

Биты	Описание
31—25	Резерв
24	GUID
23—16	Номер основной версии
15—8	Резерв
7—0	Номер подверсии

Рассмотрим краткое описание таблицы.

- Биты 31—25 зарезервированы и не используются.
- Бит 24 определяет наличие глобального идентификатора, доступного через регистр `GUID_ROM` (1 — идентификатор присутствует, 0 — идентификатор недоступен).
- Биты 23—16 определяют основной номер версии в двоично-десятичном формате. Например, для версии 1.0 значение будет равно `01h`.
- Биты 15—8 зарезервированы и не используются.
- Биты 7—0 определяют номер подверсии в двоично-десятичном формате. Например, для версии 1.0 значение будет равно 10h.

18.2.1.4. Регистр *GUID_ROM*

Данный регистр является необязательным и содержит адреса доступа к ROM GUID. Формат регистра показан в табл. 18.44.

Таблица 18.44. *Формат регистра* `GUID_ROM`

Биты	Описание
31	Сброс регистра
30—26	Резерв
25	Байт статуса
24	Резерв
23—16	Данные адреса
15—8	Резерв
7—0	Адрес первого байта

Приведем краткое описание таблицы.

- Бит 31 отвечает за сброс регистра. Для этого программа должна записать сюда значение 1.
- Биты 30—26 зарезервированы и не используются.
- Бит 25 определяет состояние чтения регистра. Когда хост завершает чтение, бит устанавливается в 0.
- Бит 24 зарезервирован и не используется.
- Биты 23—16 определяют считанные данные из регистра.
- Биты 15—8 зарезервированы и не используются.

- Биты 7—0 содержат адрес первого байта, связанного с ROM GUID с этим полем.

18.2.1.5. Регистр управления хост-контроллером

Данный регистр предоставляет определенные возможности по управлению хост-контроллером шины. Формат регистра показан в табл. 18.45.

Таблица 18.45. *Формат регистра управления хостом*

Биты	Описание
31	Позволяет хосту блокировать доступ к конфигурации ROM
30	Управляет служебным байтом подкачки во время обмена данными
29	Позволяет управлять запросом для проверки конфигурации ROM
28—24	Резерв
23	Если значение равно 1, то программное обеспечение имеет доступ к конфигурации, иначе оно не может менять конфигурацию
22	Управляет расширениями интерфейса (1 — расширения доступны, 0 — расширения не доступны)
21—20	Резерв
19	Служит для управления питанием (1 — разрешено, 0 — запрещено)
18	Включает (1) или отключает (0) физическую запись посылок
17	Программа устанавливает бит в 1 перед началом выполнения операции, бит может быть сброшен в 0 только при сбросе устройства или программы
16	Управляет программным сбросом (1 — сброс, 0 — состояние отсутствия программного сброса)
15—0	Резерв

Вообще, регистров у контроллера OHCI достаточно много и описание их всех потребует отдельной книги. Поэтому мы остановимся на представленных выше регистрах, по которым можно составить общее представление о контроллере. Более подробную информацию читатель может найти в спецификации OHCI, доступной для свободного доступа в Интернете.

А сейчас немного поговорим о практическом использовании полученных сведений. Сразу отмечу, что получить информацию об устройствах с интерфейсом IEEE 1394 можно, как и для устройств USB, посредством функций менеджера установки устройств, а также с помощью универсальной функции ввода-вывода `DeviceIoControl`.

Рассмотрим пример получения списка устройств с интерфейсом IEEE 1394 (листинг 18.4), но перед этим определим необходимые константы и структуры данных (листинг 18.3).

Листинг 18.3. Файл определений 1394.h

```
#ifdef __cplusplus
extern "C" {
#endif
// флаги для асинхронной обработки
#define ASYNC_LOOPBACK_READ                    1
#define ASYNC_LOOPBACK_WRITE                   2
#define ASYNC_LOOPBACK_LOCK                    4
#define ASYNC_LOOPBACK_RANDOM_LENGTH           8
// специфические флаги
#define ASYNC_ALLOC_USE_MDL                    1
#define ASYNC_ALLOC_USE_FIFO                   2
#define ASYNC_ALLOC_USE_NONE                   3

#define BIG_ENDIAN_ADDRESS_RANGE               1
// флаги доступа
#define ACCESS_FLAGS_TYPE_READ                 1
#define ACCESS_FLAGS_TYPE_WRITE                2
#define ACCESS_FLAGS_TYPE_LOCK                 4
// флаги событий
#define NOTIFY_FLAGS_NEVER                     0
#define NOTIFY_FLAGS_AFTER_READ                1
#define NOTIFY_FLAGS_AFTER_WRITE               2
#define NOTIFY_FLAGS_AFTER_LOCK                4
// типы блокировок транзакций
#define LOCK_TRANSACTION_MASK_SWAP             1
#define LOCK_TRANSACTION_COMPARE_SWAP          2
#define LOCK_TRANSACTION_FETCH_ADD             3
#define LOCK_TRANSACTION_LITTLE_ADD            4
#define LOCK_TRANSACTION_BOUNDED_ADD           5
#define LOCK_TRANSACTION_WRAP_ADD              6

// флаг для асинхронного чтения, записи, блокировки
#define ASYNC_FLAGS_NONINCREMENTING            1

// флаг статуса
#define ASYNC_FLAGS_NO_STATUS              0x00000002
```

```
// получение информации о хост-контроллере
#define GET_HOST_UNIQUE_ID                      1
#define GET_HOST_CAPABILITIES                   2
#define GET_POWER_SUPPLIED                      3
#define GET_PHYS_ADDR_ROUTINE                   4
#define GET_HOST_CONFIG_ROM                     5
#define GET_HOST_CSR_CONTENTS                   6
#define HOST_INFO_PACKET_BASED                  1
#define HOST_INFO_STREAM_BASED                  2
#define HOST_INFO_SUPPORTS_ISOCH_STRIPPING      4
#define HOST_INFO_SUPPORTS_START_ON_CYCLE       8
#define HOST_INFO_SUPPORTS_RETURNING_ISO_HDR    16
#define HOST_INFO_SUPPORTS_ISO_HDR_INSERTION    32

// флаги сброса
#define BUS_RESET_FLAGS_PERFORM_RESET           1
#define BUS_RESET_FLAGS_FORCE_ROOT              2

// поддерживаемые скорости шины
#define SPEED_FLAGS_100                         0x01
#define SPEED_FLAGS_200                         0x02
#define SPEED_FLAGS_400                         0x04
#define SPEED_FLAGS_800                         0x08
#define SPEED_FLAGS_1600                        0x10
#define SPEED_FLAGS_3200                        0x20
#define SPEED_FLAGS_FASTEST                     0x80000000

// флаги для изохронного обмена данных
#define RESOURCE_USED_IN_LISTENING              1
#define RESOURCE_USED_IN_TALKING                2
#define RESOURCE_BUFFERS_CIRCULAR               4
#define RESOURCE_STRIP_ADDITIONAL_QUADLETS      8
#define RESOURCE_TIME_STAMP_ON_COMPLETION       16
#define RESOURCE_SYNCH_ON_TIME                  32
#define RESOURCE_USE_PACKET_BASED               64
#define DESCRIPTOR_SYNCH_ON_SY                  0x00000001
#define DESCRIPTOR_SYNCH_ON_TAG                 0x00000002
#define DESCRIPTOR_SYNCH_ON_TIME                0x00000004
#define DESCRIPTOR_USE_SY_TAG_IN_FIRST          0x00000008
#define DESCRIPTOR_TIME_STAMP_ON_COMPLETION     0x00000010
#define DESCRIPTOR_PRIORITY_TIME_DELIVERY       0x00000020
#define DESCRIPTOR_HEADER_SCATTER_GATHER        0x00000040
#define SYNCH_ON_SY                             DESCRIPTOR_SYNCH_ON_SY
```

```
#define SYNCH_ON_TAG                                DESCRIPTOR_SYNCH_ON_TAG
#define SYNCH_ON_TIME                               DESCRIPTOR_SYNCH_ON_TIME

// флаги для установки свойств порта
#define SET_LOCAL_HOST_PROPERTIES_NO_CYCLE_STARTS       0x00000001
#define SET_LOCAL_HOST_PROPERTIES_GAP_COUNT             0x00000002
#define SET_LOCAL_HOST_PROPERTIES_MODIFY_CROM           0x00000003

// флаги обработки данных в ROM
#define SLHP_FLAG_ADD_CROM_DATA         0x01
#define SLHP_FLAG_REMOVE_CROM_DATA      0x02

// дополнительные флаги
#define INITIAL_REGISTER_SPACE_HI               0xffff
#define TOPOLOGY_MAP_LOCATION                   0xf0001000
#define SPEED_MAP_LOCATION                      0xf0002000


// структура временных отсчетов
typedef struct _CYCLE_TIME
{
    ULONG               CL_CycleOffset:12;      // Bits 0-11
    ULONG               CL_CycleCount:13;       // Bits 12-24
    ULONG               CL_SecondCount:7;       // Bits 25-31
} CYCLE_TIME, *PCYCLE_TIME;

// структура формата адреса узла
typedef struct _NODE_ADDRESS
{
    USHORT              NA_Node_Number:6;       // Bits 10-15
    USHORT              NA_Bus_Number:10;       // Bits 0-9
} NODE_ADDRESS, *PNODE_ADDRESS;

// формат адреса (48-разрядная адресация)
typedef struct _ADDRESS_OFFSET
{
    USHORT              Off_High;
    ULONG               Off_Low;
} ADDRESS_OFFSET, *PADDRESS_OFFSET;

// формат адреса ввода-вывода
typedef struct _IO_ADDRESS
{
    NODE_ADDRESS        IA_Destination_ID;
```

```
    ADDRESS_OFFSET      IA_Destination_Offset;
} IO_ADDRESS, *PIO_ADDRESS;

// формат адреса собственного идентификатора пакета данных
typedef struct _SELF_ID
{
    ULONG               SID_Phys_ID:6;          // Byte 0 — Bits 0-5
    ULONG               SID_Packet_ID:2;        // Byte 0 — Bits 6-7
    ULONG               SID_Gap_Count:6;        // Byte 1 — Bits 0-5
    ULONG               SID_Link_Active:1;      // Byte 1 — Bit 6
    ULONG               SID_Zero:1;             // Byte 1 — Bit 7
    ULONG               SID_Power_Class:3;      // Byte 2 — Bits 0-2
    ULONG               SID_Contender:1;        // Byte 2 — Bit 3
    ULONG               SID_Delay:2;            // Byte 2 — Bits 4-5
    ULONG               SID_Speed:2;            // Byte 2 — Bits 6-7
    ULONG               SID_More_Packets:1;     // Byte 3 — Bit 0
    ULONG               SID_Initiated_Rst:1;    // Byte 3 — Bit 1
    ULONG               SID_Port3:2;            // Byte 3 — Bits 2-3
    ULONG               SID_Port2:2;            // Byte 3 — Bits 4-5
    ULONG               SID_Port1:2;            // Byte 3 — Bits 6-7
} SELF_ID, *PSELF_ID;

// формат карты узла
typedef struct _TOPOLOGY_MAP
{
    USHORT              TOP_Length;      // размер структуры    USHORT
                       TOP_CRC;         // 16-битовое поле контроля ошибок
    ULONG               TOP_Generation;  // номер узла    USHORT
                   TOP_Node_Count;      // количество узлов
    USHORT              TOP_Self_ID_Count;   // смещение для
                                             идентификаторов
    SELF_ID             TOP_Self_ID_Array []; // смещение для массива
                                              идентификаторов
} TOPOLOGY_MAP, *PTOPOLOGY_MAP;

// формат карты скорости
typedef struct _SPEED_MAP
{
    USHORT              SPD_Length;     // размер структуры
    USHORT              SPD_CRC;        // 16-битовое поле контроля ошибок
    ULONG               SPD_Generation;  // номер узла
    UCHAR               SPD_Speed_Code [4032];
} SPEED_MAP, *PSPEED_MAP;
```

```
// структуры формата информационных уровней хоста
typedef struct _GET_LOCAL_HOST_INFO1
{
    LARGE_INTEGER       UniqueId;
} GET_LOCAL_HOST_INFO1, *PGET_LOCAL_HOST_INFO1;

typedef struct _GET_LOCAL_HOST_INFO2
{
    ULONG               HostCapabilities;
    ULONG               MaxAsyncReadRequest;
    ULONG               MaxAsyncWriteRequest;
} GET_LOCAL_HOST_INFO2, *PGET_LOCAL_HOST_INFO2;

typedef struct _GET_LOCAL_HOST_INFO3
{
    ULONG               deciWattsSupplied;
    ULONG               Voltage;
} GET_LOCAL_HOST_INFO3, *PGET_LOCAL_HOST_INFO3;

typedef struct _GET_LOCAL_HOST_INFO4
{
    PVOID               PhysAddrMappingRoutine;
    PVOID               Context;
} GET_LOCAL_HOST_INFO4, *PGET_LOCAL_HOST_INFO4;

typedef struct _GET_LOCAL_HOST_INFO5
{
    PVOID                   ConfigRom;
    ULONG                   ConfigRomLength;
} GET_LOCAL_HOST_INFO5, *PGET_LOCAL_HOST_INFO5;

typedef struct _GET_LOCAL_HOST_INFO6
{
    ADDRESS_OFFSET          CsrBaseAddress;
    ULONG                   CsrDataLength;
    UCHAR                   CsrDataBuffer [1];
} GET_LOCAL_HOST_INFO6, *PGET_LOCAL_HOST_INFO6;

typedef struct _SET_LOCAL_HOST_PROPS2
{
    ULONG       GapCountLowerBound;
} SET_LOCAL_HOST_PROPS2, *PSET_LOCAL_HOST_PROPS2;
```

```
typedef struct _SET_LOCAL_HOST_PROPS3
{
    ULONG       fulFlags;
    HANDLE      hCromData;
    ULONG       nLength;
    UCHAR       Buffer[1];
//    PMDL        Mdl;
} SET_LOCAL_HOST_PROPS3, *PSET_LOCAL_HOST_PROPS3;

// формат пакета данных конфигурации
typedef struct _PHY_CONFIGURATION_PACKET
{
    ULONG               PCP_Phys_ID:6;          // Byte 0 — Bits 0-5
    ULONG               PCP_Packet_ID:2;        // Byte 0 — Bits 6-7
    ULONG               PCP_Gap_Count:6;        // Byte 1 — Bits 0-5
    ULONG               PCP_Set_Gap_Count:1;    // Byte 1 — Bit 6
    ULONG               PCP_Force_Root:1;       // Byte 1 — Bit 7
    ULONG               PCP_Reserved1:8;        // Byte 2 — Bits 0-7
    ULONG               PCP_Reserved2:8;        // Byte 3 — Bits 0-7
    ULONG               PCP_Inverse;            // Inverse quadlet
} PHY_CONFIGURATION_PACKET, *PPHY_CONFIGURATION_PACKET;

// структура для выделенных адресов в памяти
typedef struct _ALLOCATE_ADDRESS_RANGE
{
    ULONG           fulAllocateFlags;
    ULONG           fulFlags;
    ULONG           nLength;
    ULONG           MaxSegmentSize;
    ULONG           fulAccessType;
    ULONG           fulNotificationOptions;
    ADDRESS_OFFSET  Required1394Offset;
    HANDLE          hAddressRange;
    UCHAR           Data[1];
} ALLOCATE_ADDRESS_RANGE, *PALLOCATE_ADDRESS_RANGE;

// структура для асинхронного чтения данных
typedef struct _ASYNC_READ
{
    ULONG           bRawMode;
    ULONG           bGetGeneration;
    IO_ADDRESS      DestinationAddress;
    ULONG           nNumberOfBytesToRead;
```

```
    ULONG           nBlockSize;
    ULONG           fulFlags;
    ULONG           ulGeneration;
    UCHAR           Data [1];
} ASYNC_READ, *PASYNC_READ;

// структура для установки адреса данных
typedef struct _SET_ADDRESS_DATA
{
    HANDLE          hAddressRange;
    ULONG           nLength;
    ULONG           ulOffset;
    UCHAR           Data[1];
} SET_ADDRESS_DATA, *PSET_ADDRESS_DATA;

// структура для асинхронной записи данных
typedef struct _ASYNC_WRITE
{
    ULONG           bRawMode;
    ULONG           bGetGeneration;
    IO_ADDRESS      DestinationAddress;
    ULONG           nNumberOfBytesToWrite;
    ULONG           nBlockSize;
    ULONG           fulFlags;
    ULONG           ulGeneration;
    UCHAR           Data [1];
} ASYNC_WRITE, *PASYNC_WRITE;

// структура для асинхронных блокировок данных
typedef struct _ASYNC_LOCK
{
    ULONG           bRawMode;
    ULONG           bGetGeneration;
    IO_ADDRESS      DestinationAddress;
    ULONG           nNumberOfArgBytes;
    ULONG           nNumberOfDataBytes;
    ULONG           fulTransactionType;
    ULONG           fulFlags;
    ULONG           Arguments [2];
    ULONG           DataValues [2];
    ULONG           ulGeneration;
    ULONG           Buffer [2];
} ASYNC_LOCK, *PASYNC_LOCK;
```

```
// структура для асинхронной потоковой обработки
typedef struct _ASYNC_STREAM
{
    ULONG           nNumberOfBytesToStream;
    ULONG           fulFlags;
    ULONG           ulTag;
    ULONG           nChannel;
    ULONG           ulSynch;
    ULONG           nSpeed;
    UCHAR           Data [1];
} ASYNC_STREAM, *PASYNC_STREAM;

// структуры для поддержки изохронной передачи данных
typedef struct _ISOCH_ALLOCATE_BANDWIDTH
{
    ULONG           nMaxBytesPerFrameRequested;
    ULONG           fulSpeed;
    HANDLE          hBandwidth;
    ULONG           BytesPerFrameAvailable;
    ULONG           SpeedSelected;
} ISOCH_ALLOCATE_BANDWIDTH, *PISOCH_ALLOCATE_BANDWIDTH;

typedef struct _ISOCH_ALLOCATE_CHANNEL
{
    ULONG           nRequestedChannel;
    ULONG           Channel;
    LARGE_INTEGER   ChannelsAvailable;
} ISOCH_ALLOCATE_CHANNEL, *PISOCH_ALLOCATE_CHANNEL;

typedef struct _ISOCH_ALLOCATE_RESOURCES
{
    ULONG           fulSpeed;
    ULONG           fulFlags;
    ULONG           nChannel;
    ULONG           nMaxBytesPerFrame;
    ULONG           nNumberOfBuffers;
    ULONG           nMaxBufferSize;
    ULONG           nQuadletsToStrip;
    HANDLE          hResource;
} ISOCH_ALLOCATE_RESOURCES, *PISOCH_ALLOCATE_RESOURCES;

typedef struct _RING3_ISOCH_DESCRIPTOR
{
    ULONG           fulFlags;
```

```
    ULONG           ulLength;
    ULONG           nMaxBytesPerFrame;
    ULONG           ulSynch;
    ULONG           ulTag;
    CYCLE_TIME      CycleTime;
    ULONG           bUseCallback;
    ULONG           bAutoDetach;
    UCHAR           Data[1];
} RING3_ISOCH_DESCRIPTOR, *PRING3_ISOCH_DESCRIPTOR;

typedef struct _ISOCH_ATTACH_BUFFERS
{
    HANDLE                  hResource;
    ULONG                   nNumberOfDescriptors;
    ULONG                   ulBufferSize;
    ULONG                   pIsochDescriptor;
    RING3_ISOCH_DESCRIPTOR  R3_IsochDescriptor[1];
} ISOCH_ATTACH_BUFFERS, *PISOCH_ATTACH_BUFFERS;

typedef struct _ISOCH_DETACH_BUFFERS
{
    HANDLE          hResource;
    ULONG           nNumberOfDescriptors;
    ULONG           pIsochDescriptor;
} ISOCH_DETACH_BUFFERS, *PISOCH_DETACH_BUFFERS;

typedef struct _ISOCH_LISTEN
{
    HANDLE          hResource;
    ULONG           fulFlags;
    CYCLE_TIME      StartTime;
} ISOCH_LISTEN, *PISOCH_LISTEN;

typedef struct _ISOCH_QUERY_RESOURCES
{
    ULONG           fulSpeed;
    ULONG           BytesPerFrameAvailable;
    LARGE_INTEGER   ChannelsAvailable;
} ISOCH_QUERY_RESOURCES, *PISOCH_QUERY_RESOURCES;

typedef struct _ISOCH_SET_CHANNEL_BANDWIDTH
{
    HANDLE          hBandwidth;
```

```
    ULONG           nMaxBytesPerFrame;
} ISOCH_SET_CHANNEL_BANDWIDTH, *PISOCH_SET_CHANNEL_BANDWIDTH;

typedef struct _ISOCH_STOP
{
    HANDLE          hResource;
    ULONG           fulFlags;
} ISOCH_STOP, *PISOCH_STOP;

typedef struct _ISOCH_TALK
{
    HANDLE          hResource;
    ULONG           fulFlags;
    CYCLE_TIME      StartTime;
} ISOCH_TALK, *PISOCH_TALK;

// структура для получения информации о хосте
typedef struct _GET_LOCAL_HOST_INFORMATION
{
    ULONG           Status;
    ULONG           nLevel;
    ULONG           ulBufferSize;
    UCHAR           Information [1];
} GET_LOCAL_HOST_INFORMATION, *PGET_LOCAL_HOST_INFORMATION;

// структура для получения адреса устройства
typedef struct _GET_1394_ADDRESS
{
    ULONG           fulFlags;
    NODE_ADDRESS    NodeAddress;
} GET_1394_ADDRESS, *PGET_1394_ADDRESS;

// получение максимальной скорости устройства
typedef struct _GET_MAX_SPEED_BETWEEN_DEVICES
{
    ULONG           fulFlags;
    ULONG           ulNumberOfDestinations;
    ULONG           hDestinationDeviceObjects [64];
    ULONG           fulSpeed;
} GET_MAX_SPEED_BETWEEN_DEVICES, *PGET_MAX_SPEED_BETWEEN_DEVICES;

// структура для установки данных для хоста
typedef struct _SET_LOCAL_HOST_INFORMATION
```

```
{
    ULONG           nLevel;
    ULONG           ulBufferSize;
    UCHAR           Information [1];
} SET_LOCAL_HOST_INFORMATION, *PSET_LOCAL_HOST_INFORMATION;

// структура для хранения версии
typedef struct _VERSION_DATA
{
    ULONG           ulVersion;
    ULONG           ulSubVersion;
} VERSION_DATA, *PVERSION_DATA;

// структура для асинхронного обмена данными
typedef struct _ASYNC_LOOPBACK_PARAMS
{
    HWND                            hWnd;
    PSTR                            szDeviceName;
    BOOLEAN                         bKill;
    HANDLE                          hThread;
    UINT                            ThreadId;
    ULONG                           ulLoopFlag;
    ULONG                           nIterations;
    ULONG                           ulPass;
    ULONG                           ulFail;
    ULONG                           ulIterations;
    ADDRESS_OFFSET                  AddressOffset;
    ULONG                           nMaxBytes;
} ASYNC_LOOPBACK_PARAMS, *PASYNC_LOOPBACK_PARAMS;

// структура для изохронного обмена данными
typedef struct _ISOCH_LOOPBACK_PARAMS
{
    HWND                            hWnd;
    PSTR                            szDeviceName;
    BOOLEAN                         bKill;
    BOOLEAN                         bLoopback;
    BOOLEAN                         bAutoFill;
    HANDLE                          hThread;
    UINT                            ThreadId;
    ULONG                           ulLoopFlag;
    ULONG                           nIterations;
    ULONG                           ulPass;
```

```
    ULONG                           ulFail;
    ULONG                           ulIterations;
    BOOLEAN                         bAutoAlloc;
    ISOCH_ATTACH_BUFFERS            isochAttachBuffers;
} ISOCH_LOOPBACK_PARAMS, *PISOCH_LOOPBACK_PARAMS;

// поддержка списка устройств
typedef struct _DEVICE_LIST
{
    CHAR            DeviceName [255];
} DEVICE_LIST, *PDEVICE_LIST;

typedef struct _DEVICE_DATA
{
    ULONG           numDevices;
    DEVICE_LIST     deviceList [10];
} DEVICE_DATA, *PDEVICE_DATA;

#define bswap(value)    (((ULONG) (value)) << 24 |\
                         (((ULONG) (value)) & 0x0000FF00) << 8 |\
                         (((ULONG) (value)) & 0x00FF0000) >> 8 |\
                         ((ULONG) (value)) >> 24)

#define bswapw(value)   (((USHORT) (value)) << 8 |\
                         (((USHORT) (value)) & 0xff00) >> 8)
#endif
#ifdef __cplusplus
}
#endif
```

Листинг 18.4. Получение информации об устройствах интерфейса IEEE 1394

```
#include "stdafx.h"
#include <setupapi.h>
#include "1394.h"
#define GUID_1394 "12345678-1234-1234-1234-000012345678"

void GetDeviceList()
{
    HDEVINFO                        hDevInfo;
    ULONG                           i = 0;
    SP_DEVICE_INTERFACE_DATA        deviceInterfaceData;
```

```
PSP_DEVICE_INTERFACE_DETAIL_DATA    DeviceInterfaceDetailData;
ULONG                               requiredSize;
GUID                                t1394GUID;
ULONG                               dwError;
BOOL                                bReturn;
PDEVICE_DATA                        DeviceData;

t1394GUID = GUID_1394;
DeviceData->numDevices = 0;

 hDevInfo = SetupDiGetClassDevs ( &t1394GUID, NULL, NULL,
                ( DIGCF_PRESENT | DIGCF_INTERFACEDEVICE ) );

if ( !hDevInfo )
{
   dwError = GetLastError ();
}
else
{
    while (TRUE)
    {

     deviceInterfaceData.cbSize = sizeof( SP_DEVICE_INTERFACE_DATA );

     if (SetupDiEnumDeviceInterfaces( hDevInfo, 0, &t1394DiagGUID, i,
                                      &deviceInterfaceData))
     {

          bReturn = SetupDiGetDeviceInterfaceDetail( hDevInfo,
                 &deviceInterfaceData, NULL, 0, &requiredSize, NULL);

          if ( !bReturn )
          {
               dwError = GetLastError ();
          }

          DeviceInterfaceDetailData = LocalAlloc(LPTR, requiredSize);
          DeviceInterfaceDetailData->cbSize =
                       sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

          bReturn = SetupDiGetDeviceInterfaceDetail( hDevInfo,
                 &deviceInterfaceData, DeviceInterfaceDetailData,
                 requiredSize, NULL, NULL );
```

```
            if ( !bReturn )
            {
                dwError = GetLastError ();

                LocalFree(DeviceInterfaceDetailData);
                break;
              }
              else
              {
                DeviceData->numDevices ++;
                lstrcpy(DeviceData->deviceList[i].DeviceName,
                        DeviceInterfaceDetailData->DevicePath);
                LocalFree(DeviceInterfaceDetailData);
              }
          }
          else
          {
              dwError = GetLastError ();
              break;
          }
          i++;
      }
      SetupDiDestroyDeviceInfoList(hDevInfo);
  }
}
```

В рассмотренном примере мы получили список всех доступных устройств IEEE 1394, установленных в системе. Для того чтобы передать или принять от устройства данные, необходимо его открыть. Делается это с помощью стандартной функции `CreateFile`, как показано в листинге 18.5.

Листинг 18.5. Открытие устройства IEEE 1394

```
HANDLE OpenDevice( LPCWSTR szDeviceName )
{
    HANDLE  hDevice;
    CHAR  tmpBuff[STRING_SIZE];

    hDevice = CreateFile( szDeviceName, GENERIC_WRITE | GENERIC_READ,
                          FILE_SHARE_WRITE | FILE_SHARE_READ,
                          NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED,
                          NULL );
```

```
        if (hDevice == INVALID_HANDLE_VALUE)
        {
        // ошибка открытия устройства
        }
    }

        return(hDevice);
}
```

После того как устройство открыто, можно считывать и записывать данные. Пример, демонстрирующий чтение данных в асинхронном режиме, представлен в листинге 18.6.

Листинг 18.6. Чтение данных из устройства IEEE 1394

```
#include <winioctl.h>
#define DIAG1394_IOCTL_INDEX  0x0800
#define IOCTL_ASYNC_READ CTL_CODE( FILE_DEVICE_UNKNOWN,  \
       DIAG1394_IOCTL_INDEX + 2, \ METHOD_BUFFERED, \ FILE_ANY_ACCESS)

DWORD ReadData_1394( LPCWSTR szDeviceName, PASYNC_READ asyncRead )
{
    HANDLE          hDevice;
    DWORD           dwRet, dwBytesRet;
    PASYNC_READ     pAsyncRead;
    ULONG           ulBufferSize;
    ULONG           i;

    ulBufferSize = sizeof(ASYNC_READ) + asyncRead->nNumberOfBytesToRead;

    pAsyncRead = (PASYNC_READ) LocalAlloc(LPTR, ulBufferSize);

    FillMemory(pAsyncRead, ulBufferSize, 0);

    *pAsyncRead = *asyncRead;

    hDevice = OpenDevice(szDeviceName);

    if (hDevice != INVALID_HANDLE_VALUE)
   {

        dwRet = DeviceIoControl( hDevice, IOCTL_ASYNC_READ,
                                 pAsyncRead, ulBufferSize,
                                 pAsyncRead, ulBufferSize,
                                 &dwBytesRet, NULL );
```

```
        if ( !dwRet )
        {
           dwRet = GetLastError();
        }
        else
        {
           dwRet = ERROR_SUCCESS;

            for ( i = 0; i < (asyncRead -> nNumberOfBytesToRead /
                      sizeof (ULONG)); i++ )
            {
                PULONG ulTemp;
                ulTemp = (PULONG) &pAsyncRead->Data[i * sizeof(ULONG)];
            }
        }
        CloseHandle(hDevice);
    }

    if (pAsyncRead)
        LocalFree(pAsyncRead);

return(dwRet);
}
```

Как видно из примера, мы вначале открываем устройство, передав в качестве аргумента функции `OpenDevice` значение, полученное из функции `GetDeviceList`. Далее вызываем процедуру ввода-вывода `DeviceIoControl`, передав ей управляющий код для драйвера и указатель на буфер, в который будут записаны полученные данные. Аналогичным образом можно передать данные на устройство (листинг 18.7).

Листинг 18.7. Запись данных на устройство IEEE 1394

```
#include <winioctl.h>
#define DIAG1394_IOCTL_INDEX  0x0800
#define IOCTL_ASYNC_WRITE CTL_CODE( FILE_DEVICE_UNKNOWN,  \
       DIAG1394_IOCTL_INDEX + 3, \ METHOD_BUFFERED, \ FILE_ANY_ACCESS)
DWIRD WriteData_1394 ( LPCWSTR szDeviceName, PASYNC_WRITE asyncWrite )
{
    HANDLE          hDevice;
    DWORD           dwRet, dwBytesRet;
    PASYNC_WRITE    pAsyncWrite;
    ULONG           ulBufferSize;
    ULONG           i;
```

```
ulBufferSize =
   sizeof(ASYNC_WRITE) + asyncWrite -> nNumberOfBytesToWrite;

   pAsyncWrite = (PASYNC_WRITE)LocalAlloc(LPTR, ulBufferSize);

   FillMemory(pAsyncWrite, ulBufferSize, 0);

   *pAsyncWrite = *asyncWrite;

   for ( i = 0; i < asyncWrite -> nNumberOfBytesToWrite
                                      / sizeof(ULONG); i++ )
   {
        CopyMemory((ULONG *) &pAsyncWrite->Data + i, (ULONG *)&i,
                                                    sizeof(ULONG));
   }

for ( i = 0; i < ( asyncWrite -> nNumberOfBytesToWrite
                                     / sizeof(ULONG)); i++ )
{
    PULONG ulTemp;
    ulTemp = (PULONG) &pAsyncWrite -> Data[i * sizeof(ULONG)];
}

hDevice = OpenDevice(szDeviceName);

if (hDevice != INVALID_HANDLE_VALUE)
{
    dwRet = DeviceIoControl( hDevice, IOCTL_ASYNC_WRITE,
                             pAsyncWrite, ulBufferSize,
                             pAsyncWrite, ulBufferSize,
                             &dwBytesRet, NULL );

    if ( !dwRet )
    {
        dwRet = GetLastError();
    }
    else
    {
        dwRet = ERROR_SUCCESS;
    }

    CloseHandle(hDevice);
}
```

```
    if (pAsyncWrite)
        LocalFree(pAsyncWrite);

    return(dwRet);
}
```

И последний пример, который мы рассмотрим, поможет получить информацию о хост-контроллере (листинг 18.8).

Листинг 18.8. Получение информации о хост-контроллере

```
DWORD GetHostInformation( LPCWSTR szDeviceName,
                          PGET_LOCAL_HOST_INFORMATION getHostInfo )
{
    PGET_LOCAL_HOST_INFORMATION pGetHostInfo;
    HANDLE hDevice;
    DWORD dwRet, dwBytesRet;
    ULONG ulBufferSize;

    GetLocalHostInfo->Status = 0;
        if (getHostInfo->nLevel == 1)
            ulBufferSize = sizeof(GET_LOCAL_HOST_INFO1);
        else if (getHostInfo->nLevel == 2)
            ulBufferSize = sizeof(GET_LOCAL_HOST_INFO2);
        else if (getHostInfo->nLevel == 3)
            ulBufferSize = sizeof(GET_LOCAL_HOST_INFO3);
        else if (getHostInfo->nLevel == 4)
            ulBufferSize = sizeof(GET_LOCAL_HOST_INFO4);
        else if (getHostInfo->nLevel == 5)
        {
            dwRet = ERROR_INVALID_FUNCTION;
            return dwRet;
        }
        else if (getHostInfo->nLevel == 6)
        {
            dwRet = ERROR_INVALID_FUNCTION;
            return dwRet;
        }

        ulBufferSize += sizeof(GET_LOCAL_HOST_INFORMATION);

        pGetHostInfo =
       (PGET_LOCAL_HOST_INFORMATION)LocalAlloc(LPTR, ulBufferSize);
```

```
        *pGetHostInfo = *getHostInfo;
    }

    hDevice = OpenDevice(szDeviceName);

    if (hDevice != INVALID_HANDLE_VALUE)
    {

        dwRet = DeviceIoControl( hDevice,
                                 IOCTL_GET_LOCAL_HOST_INFORMATION,
                                 pGetHostInfo, ulBufferSize,
                                 pGetHostInfo, ulBufferSize,
                                 &dwBytesRet, NULL );

        if ( ( dwRet ) && ( !pGetHostInfo->Status) )
        {
            dwRet = ERROR_SUCCESS;

            if (pGetHostInfo->nLevel == 1)
            {
                PGET_LOCAL_HOST_INFO1 LocalHostInfo1;
                LocalHostInfo1 =
                     (PGET_LOCAL_HOST_INFO1)&pGetHostInfo->Information;

             }
            else if (pGetHostInfo->nLevel == 2)
            {
                PGET_LOCAL_HOST_INFO2 LocalHostInfo2;
                LocalHostInfo2 =
                     (PGET_LOCAL_HOST_INFO2)&pGetHostInfo->Information;

            }
            else if (pGetHostInfo->nLevel == 3)
            {
                PGET_LOCAL_HOST_INFO3 LocalHostInfo3;
                LocalHostInfo3 =
                     (PGET_LOCAL_HOST_INFO3)&pGetHostInfo->Information;

            }
            else if (pGetHostInfo->nLevel == 4)
            {
                PGET_LOCAL_HOST_INFO4 LocalHostInfo4;
                LocalHostInfo4 =
                     (PGET_LOCAL_HOST_INFO4)&pGetHostInfo->Information;
            }
```

```
else if (pGetHostInfo->nLevel == 5)
{
    PGET_LOCAL_HOST_INFO5 LocalHostInfo5;
    LocalHostInfo5 =
         (PGET_LOCAL_HOST_INFO5)&pGetHostInfo->Information;


}
else if (pGetHostInfo->nLevel == 6)
{
    PGET_LOCAL_HOST_INFO6 LocalHostInfo6;
    LocalHostInfo6 =
         (PGET_LOCAL_HOST_INFO6)&pGetHostInfo->Information;


    if (LocalHostInfo6->CsrBaseAddress.Off_Low ==
                              SPEED_MAP_LOCATION)
    {
        PSPEED_MAP SpeedMap;
        ULONG NumNodes, i;


        SpeedMap =
         (PSPEED_MAP)&LocalHostInfo6->CsrDataBuffer;
    }
    else if (LocalHostInfo6->CsrBaseAddress.Off_Low ==
                              TOPOLOGY_MAP_LOCATION)
    {
        PTOPOLOGY_MAP TopologyMap;
        ULONG i;
        PSELF_ID SelfId;
        PSELF_ID_MORE SelfIdMore;
        BOOL bMore;
        TopologyMap =
         (PTOPOLOGY_MAP)&LocalHostInfo6->CsrDataBuffer;


        for (i = 0; i < TopologyMap->TOP_Self_ID_Count; i++)
        {
            SelfId = &TopologyMap->TOP_Self_ID_Array[i];


            if (SelfId->SID_More_Packets)
                bMore = TRUE;
            else
                bMore = FALSE;
```

```
                        while (bMore)
                        {
                            i++;
                            SelfIdMore =
                      (PSELF_ID_MORE)&TopologyMap->TOP_Self_ID_Array[i];

                            if (SelfIdMore->SID_More_Packets)
                                bMore = TRUE;
                            else
                                bMore = FALSE;
                        }
                    }
                }
                else
                {
                     // ошибка
                }
            }
        }
        else if (pGetLocalHostInfo->Status)
        {
            dwRet = ERROR_INSUFFICIENT_BUFFER;
        }
        else
        {
            dwRet = GetLastError();
        }
        CloseHandle(hDevice);
    }
    LocalFree(pGetLocalHostInfo);

    return(dwRet);
}
```

На этом мы завершаем программирование устройств IEEE 1394.

18.3. Интерфейс Wireless

Данный интерфейс относится к беспроводным высокоскоростным каналам для обмена данными. На его основе разработаны различные расширения: Wireless Lan, Wi-Fi, Wireless USB, WirelessMAN и др. Несомненным преимуществом этого интерфейса служит масштабируемость и скорость переда-

чи данных (например, для Wireless USB — от 53 до 480 Мбит/с). Более подробно со структурой и принципами работы беспроводных каналов вы можете познакомиться в соответствующей литературе.

Здесь же мы поговорим о том, как беспроводные устройства взаимодействуют с компьютером и как описать это на языке программирования. В первую очередь, связь устройств с системой осуществляется посредством хост-контроллера через пространство шины PCI (PCI-X). Контроллер реализует управление и обработку данных, поступающих от внешних устройств и передает ее на шину PCI, которая уже стандартным образом выделяет необходимые системные ресурсы для программного доступа к беспроводным устройствам.

Контроллер интерфейса Wireless (*WHCI* — Wireless Host Controller Interface), о котором пойдет речь, включает в себя мультиинтерфейсный контроллер (*UMC* — UMB Multi-Interface Controller) для шины Wireless и последовательный контроллер для шины USB (*Wireless USB* — Wireless Universal Serial Bus). Это позволяет подключать устройства Wireless напрямую, а также через интерфейс USB. Связь между внешними устройствами поддерживается посредством промежуточного радиоконтроллера (*URC* — UWB Radio Controller).

Контроллер WHCI включает в себя несколько наборов регистров:

- регистры конфигурации для устройств PCI;
- отображаемые в памяти аппаратные регистры возможностей UWB;
- отображаемые в памяти регистры радиоуправления UWB;
- отображаемые в памяти регистры Wireless USB хост-контроллера.

Рассмотрим перечисленные регистры по порядку.

18.3.1. Регистры конфигурации шины PCI

Данные регистры необходимы для связи контроллера Wireless и пространства шины PCI. Основная часть их доступна только для чтения и позволяет шине PCI правильно идентифицировать подключенный интерфейс и выделить для него адреса памяти и другие системные ресурсы. Список регистров конфигурации показан в табл. 18.46.

Таблица 18.46. *Список регистров конфигурации*

Адрес смещения PCI	Описание регистра
00h—08h	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI

Таблица 18.46 *(окончание)*

Адрес смещения PCI	Описание регистра
09h—0Bh	Регистр кода класса
0Ch—0Fh	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI
10h—17h	Регистр адресов
18h—33h	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI
34h	Регистр поддержки управления питанием
35h—FFh	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI

18.3.1.1. Регистр кода класса устройства

Данный регистр определяет код класса и код подкласса, как это определено в спецификации PCI. Регистр является 24-разрядным и доступен только для чтения. Значение по умолчанию равно 0D1010h. Формат регистра представлен в табл. 18.47.

Таблица 18.47. *Формат регистра кода класса*

Биты	Описание
23—16	Базовый код класса (для контроллера Wireless значение равно 0Dh)
15—8	Код подкласса (для RF-контроллера равно 10h)
7—0	Программируемый интерфейс (для UMC равно 10h)

18.3.1.2. Регистр адреса

Данный регистр содержит базовый адрес для отображения в пространстве шины PCI. Является 64-разрядным и доступен для чтения и записи. Формат регистра представлен в табл. 18.48.

Кратко опишем поля таблицы.

- Биты 63—8 определяют базовый адрес. Поле доступно для чтения и записи.
- Биты 7—3 зарезервированы и должны быть установлены в 0. Поле доступно только для чтения.

Таблица 18.48. *Формат регистра адреса*

Биты	Описание
63—8	Базовый адрес
7—3	Резерв
2—1	Тип адресации
0	Резерв

- Биты 2—1 определяют тип адресации (`00b` — 32-разрядная адресация, `01b` — 64-разрядная адресация). Поле доступно только для чтения.
- Бит 0 зарезервирован и должен быть установлен в 0. Поле доступно только для чтения.

18.3.2. Регистры аппаратных возможностей

К ним относятся два регистра: регистр информации и регистр данных. Первый содержит информацию о версии и количестве блоков данных для второго регистра. Регистр информации расположен по адресу `00h` (смещение относительно базового адреса UWB), а регистр данных может иметь несколько блоков, первый из которых адресуется значением `08h` относительно базового адреса.

18.3.2.1. Регистр информации

Данный регистр доступен только для чтения и является 64-разрядным. Формат регистра представлен в табл. 18.49.

Таблица 18.49. *Формат регистра информации*

Биты	Описание
63—48	Резерв
47—32	Номер версии
31—18	Смещение для регистра URC
17—16	Доступ к регистрам URC
15—4	Резерв
3—0	Число регистров данных

Приведем краткое описание таблицы.

- Биты 63—48 зарезервированы и должны быть установлены в 0.
- Биты 47—32 определяют текущую версию интерфейса в двоично-десятичном формате (например, `0095h`).

- Биты 31—18 определяют смещение для первого регистра URC.
- Биты 17—16 определяют доступ к регистрам URC. Доступны значения 0 (смещение для регистра равно 10h), 1 (смещение для регистра равно 18h) и 2 (смещение для регистра равно 20h) для пространства шины PCI.
- Биты 15—4 зарезервированы и должны быть установлены в 0.
- Биты 3—0 определяют количество регистров данных, доступных хосту. Максимальное значение равно 8.

18.3.2.2. Регистр данных

Используется для загрузки информации об интерфейсе (его возможностях) для предоставления программному обработчику. Может быть доступно более одного регистра. Адрес первого следует сразу за регистром информации. Регистр данных является 64-разрядным и доступен только для чтения. Формат регистра представлен в табл. 18.50.

Таблица 18.50. *Формат регистра данных*

Биты	Описание
63—48	Резерв
47—32	Идентификатор версии
31—18	Смещение для адреса первого регистра
17—16	Используемая область
15—8	Размер
7—0	Идентификатор возможностей

Приведем краткое описание таблицы.

- Биты 63—48 зарезервированы и должны быть равны 0.
- Биты 47—32 определяют идентификатор текущей версии.
- Биты 31—18 определяют смещение для адреса первого регистра возможностей.
- Биты 17—16 определяют доступ к регистрам возможностей. Доступны значения 0 (смещение для регистра равно 10h), 1 (смещение для регистра равно 18h) и 2 (смещение для регистра равно 20h) для пространства шины PCI.
- Биты 15—8 задают размер регистра возможностей.
- Биты 7—0 содержат идентификатор (код) поддерживаемых возможностей. Доступные значения кодов представлены в табл. 18.51.

Таблица 18.51. *Значения кодов возможностей*

Значение	Описание
00h	Значение зарезервировано и не используется
01h	Хост-контроллер поддерживает регистры возможностей
02h—7Fh	Значения зарезервированы для будущих версий
80h—FFh	Определяются производителями оборудования

18.3.3. Регистры радиоуправления

Данный набор регистров является операционным, т. е. эти ригистры позволяют выполнять различные операции. Расположены они начиная со смещения, содержащегося в регистре информации (биты 31—18). К ним относятся:

- *регистр команд* (смещение 00h), имеющий размер 4 байта;
- *регистр состояния* (смещение 04h), имеющий размер 4 байта;
- *регистр прерываний* (смещение 08h), имеющий размер 4 байта;
- *регистр адреса команд* (смещение 10h), имеющий размер 8 байт;
- *регистр адреса событий* (смещение 18h), имеющий размер 8 байт.

Рассмотрим назначение и формат каждого регистра подробнее.

18.3.3.1. Регистр команд

Данный регистр предназначен для общего сброса всех регистров, а также для передачи управляющих команд (например, Scan, Start и др.). Как уже говорилось, размер регистра равен 4 байтам. Он доступен для чтения и записи, а значение по умолчанию для него равно 00000000h. Формат регистра представлен в табл. 18.52.

Таблица 18.52. *Формат регистра команд*

Биты	Описание
31	Аппаратный сброс
30	Выполнение команды
29	Адрес регистра событий
28—16	Резерв
15	Активность
14	Доступность прерываний
13	Резерв
12—0	Размер команды

Приведем краткое описание таблицы.

- Бит 31 отвечает за аппаратный сброс контроллера. При установке значения 1 происходит сброс логики, таймеров, счетчиков и т. д. Любые выполняющиеся в этот момент транзакции сразу завершаются. Следует заметить, что сброс не затрагивает данные в регистрах PCI. После выполнения сброса значение бита равно 0.
- Бит 30 отвечает за выполнение команды. Значение 1 запускает выполнение, а 0 — прекращает работу команды. В момент остановки отменяются все необработанные команды, ожидающие своей очереди.
- Бит 29 определяет произошедшие изменения в регистре событий.
- Биты 28—16 зарезервированы и всегда равны 0.
- Бит 15 определяет выполнение команды. Значение 1 говорит о том, что выполняется команда, а 0 информирует о завершении выполнения команды и готовности принять следующую.
- Бит 14 определяет использование прерывания после выполнения команды. Если значение равно 1, то после выполнения команды будет вызвано прерывание.
- Бит 13 зарезервирован и всегда равен 0.
- Биты 12—0 определяют размер команды в байтах, находящейся в буфере регистра.

18.3.3.2. Регистр состояния

Регистр позволяет получить информацию о состоянии контроллера и отложенных прерываниях. Он является 32-разрядным и доступен только для чтения (определенные биты доступны и для записи). Значение по умолчанию для него равно `00010000h`. Формат регистра представлен в табл. 18.53.

Таблица 18.53. *Формат регистра состояния*

Биты	Описание
31—18	Зарезервированы и всегда равны 0
17	Статус события
16	Статус выполнения команды
15—11	Зарезервированы и всегда равны 0
10	Системная ошибка хоста
9	Статус отложенного события
8	Статус прерывания
7—0	Источник прерывания

Приведем краткое описание таблицы.

- Биты 31—18 зарезервированы и всегда равны 0.
- Бит 17 определяет текущее состояние обработки события. Установка бита в 1 разрешает обработку события.
- Бит 16 определяет состояние выполнения текущей команды и доступен только для чтения. Значение 1 говорит о том, что выполнение команды остановлено. Бит доступен только для чтения
- Биты 15—11 зарезервированы и всегда равны 0.
- Бит 10 определяет системную ошибку хоста. Значение 1 информирует о серьезной ошибке. Ошибка может быть связана с проблемами обработки шины PCI. После того как произошла ошибка, бит сбрасывается в 0 для дальнейшего выполнения команд.
- Бит 9 определяет наличие событий в буфере регистра. Значение по умолчанию равно 0.
- Бит 8 информирует о наличии прерывания. Значение по умолчанию равно 0.
- Биты 7—0 доступны только для чтения и определяют источник прерывания.

18.3.3.3. Регистр прерываний

Данный 32-разрядный регистр включает или выключает обработку прерываний регистром команд. Он доступен для чтения и записи, а значение по умолчанию равно `00000000h`. Формат регистра представлен в табл. 18.54.

Таблица 18.54. *Формат регистра прерываний*

Биты	Описание
31—11	Резерв
10	Обработка прерываний для системных ошибок хоста
9	Обработка прерываний для событий
8	Доступность прерываний в регистре команд
7—0	Определение источника прерываний

Приведем краткое описание таблицы.

- Биты 31—11 зарезервированы и должны быть установлены в 0.
- Бит 10 определяет обработку прерываний для системных ошибок хоста (1 — обработка включена, 0 — выключена).

- Бит 9 разрешает или запрещает обработку прерываний для событий (1 — включена, 0 — выключена).
- Бит 8 разрешает или запрещает обработку прерываний для регистра команд (1 — обработка включена, 0 — выключена).
- Биты 7—0 разрешают или запрещают идентификацию источника прерываний для регистра команд (1 — идентификация включена, 0 — выключена).

18.3.3.4. Регистр адреса команд

Данный регистр является 64-разрядным и определяет адрес для буфера команд. Он доступен для чтения и записи. Формат регистра представлен в табл. 18.55.

Таблица 18.55. *Формат регистра адреса*

Биты	Описание
63—2	Адрес для буфера команд
2—0	Зарезервированы и должны быть установлены в 0

18.3.3.5. Регистр адреса событий

Данный регистр является 64-разрядным и определяет адрес для буфера событий. Он доступен для чтения и записи. Формат регистра представлен в табл. 18.56.

Таблица 18.56. *Формат регистра событий*

Биты	Описание
63—12	Адрес для буфера событий
11—0	Текущее смещение для указания адреса следующего события

18.3.4. Регистры хост-контроллера

Представляют собой большую группу регистров, обрабатывающих возможности и различные операции хост-контроллера. К ним относятся:

- *регистр номера версии интерфейса* (смещение `00h`), имеющий размер 2 байта;
- *регистр параметров* (смещение `04h`), имеющий размер 4 байта;

- *регистр команд* (смещение 08h), имеющий размер 4 байта;
- *регистр состояния* (смещение 0Ch), имеющий размер 4 байта;
- *регистр состояния команды* (смещение 14h), имеющий размер 4 байта;
- *регистр адреса* (смещение 20h), имеющий размер 8 байт;
- *регистр информации* (смещение 38h), имеющий размер 8 байт;
- *регистр ключа безопасности* (смещение 48h), имеющий размер 16 байт;
- *регистр канала времени* (смещение 68h), имеющий размер 4 байта;

Рассмотрим назначение и формат каждого регистра подробнее.

18.3.4.1. Регистр номера версии интерфейса

Регистр является 16-разрядным и предоставляет информацию о текущей версии интерфейса. Значение по умолчанию для версии 0.95 равно 0095h. Данные регистра хранятся в двоично-десятичном коде.

18.3.4.2. Регистр параметров

Регистр доступен только для чтения и предоставляет информацию о различных параметрах хоста. Формат регистра представлен в табл. 18.57.

Таблица 18.57. *Формат регистра параметров*

Биты	Описание
31—24	Биты зарезервированы и должны быть установлены в 0
23—16	Определяют число одновременно поддерживаемых хостом блоков, которые он может обработать в одно время (правильные значения лежат в диапазоне от 03h до 10h)
15—8	Биты определяют число таблиц для ключей безопасности, которые хост может одновременно сохранять (правильные значения лежат в диапазоне от 03h до 80h)
7	Зарезервирован и должен быть установлен в 0
6—0	Биты определяют количество устройств Wireless USB, поддерживаемых хостом (правильные значения лежат в диапазоне от 02h до 7Fh)

18.3.4.3. Регистр команд

Данный регистр предназначен для определения выполняемой хостом команды и является 32-разрядным. Значение по умолчанию равно 00000000h. Формат регистра представлен в табл. 18.58.

Таблица 18.58. *Формат регистра команд*

Биты	Описание
31—24	Резерв
23—16	Идентификатор
15—13	Резерв
12	Асинхронный список (удаление)
11	Периодический список (удаление)
10—8	Индекс потока
7	Прерывание для асинхронного списка
6	Прерывание для периодического списка
5	Асинхронный список (обновление)
4	Периодический список (обновление)
3	Асинхронный список (включение)
2	Периодический список (включение)
1	Сброс хост-контроллера
0	Выполнение команд

Приведем краткое описание таблицы.

- Биты 31—24 зарезервированы и должны быть установлены в 0.
- Биты 23—16 служат для хранения идентификатора устройства, используемого хостом.
- Биты 15—13 зарезервированы и должны быть установлены в 0.
- Бит 12 используется хостом для удаления запроса из очереди при асинхронном обмене данными. Значение по умолчанию равно 0. Установка значения в `1Bh` удаляет запрос из очереди.
- Бит 11 используется хостом для удаления запроса из очереди при периодическом обмене данными. Значение по умолчанию равно 0. Установка значения в `1Bh` удаляет запрос из очереди.
- Биты 10—8 служат для установки хостом индекса потока данных.
- Бит 7 служит для передачи запроса хосту на прерывание при асинхронном обмене данными. Значение по умолчанию равно 0.
- Бит 6 служит для передачи запроса хосту на прерывание при периодическом обмене данными. Значение по умолчанию равно 0.
- Бит 5 используется хостом для обновления запроса из очереди при асинхронном обмене данными. Значение по умолчанию равно 0. Установка значения в `0Bh` обновляет запрос в очереди.

- Бит 4 используется хостом для обновления запроса из очереди при периодическом обмене данными. Значение по умолчанию равно 0. Установка значения в 0Bh обновляет запрос в очереди.
- Бит 3 определяет использование асинхронного списка. Значение по умолчанию равно 0 и указывает, что асинхронный список не используется. Установка значения 1 разрешает использование.
- Бит 2 определяет использование периодического списка. Значение по умолчанию равно 0 и указывает, что периодический список не используется. Установка значения 1 разрешает использование.
- Бит 1 управляет сбросом хост-контроллера. При установке бита в 1 выполняется аппаратный сброс всей логики, таймеров и счетчиков хоста. При этом обмен данными немедленно завершается.
- Бит 0 разрешает или запрещает обработку хостом списка запросов в очереди. Значение 1 запрещает, а 0 разрешает обработку.

18.3.4.4. Регистр состояния

Данный регистр предназначен для определения различных состояний хоста и отложенных прерываний. Он является 32-разрядным. Значение по умолчанию равно 00001000h. Формат регистра представлен в табл. 18.59.

Таблица 18.59. *Формат регистра состояния*

Биты	Описание
31—16	Резерв
15	Статус асинхронного списка
14	Статус периодического списка
13	Статус списка DNTS
12	Статус выполнения
11—10	Резерв
9	Подтверждение выполнения команды
8	Использование канала времени
7	Переполнение буфера
6	Изменение периода
5	Системная ошибка хоста
4	Статус прерывания асинхронного списка
3	Статус прерывания периодического списка

Таблица 18.59 *(окончание)*

Биты	Описание
2	Период прерывания
1	Ошибка прерывания
0	Подтверждение транзакции

Приведем краткое описание таблицы.

- Биты 31—16 зарезервированы и должны быть установлены в 0.
- Бит 15 определяет состояние асинхронного списка. Значение 1 допускает выдачу информации о состоянии. Значение 0 отключает такую возможность.
- Бит 14 определяет состояние периодического списка. Значение 1 допускает выдачу информации о состоянии. Значение 0 отключает такую возможность.
- Бит 13 определяет состояние списка DNTS. Значение 1 допускает выдачу информации о состоянии. Значение 0 отключает такую возможность.
- Бит 12 определяет состояние выполнения команды. Значение равно 0, когда бит 0 в регистре команд установлен в 1.
- Бит 11—10 зарезервированы и должны быть установлены в 0.
- Бит 9 определяет состояние выполнения команды. Значение бита равно 1, когда выполнение команды завершено.
- Бит 8 устанавливается в 1 после обнуления регистра канала времени.
- Бит 7 определяет переполнение буфера на устройстве.
- Бит 6 устанавливается в 1, когда было изменено начальное время периода.
- Бит 5 определяет системную ошибку хоста (значение равно 1).
- Бит 4 определяет состояние прерывания для асинхронного списка.
- Бит 3 определяет состояние прерывания для периодического списка.
- Бит 2 определяет состояние прерывания для списка DNTS.
- Бит 1 определяет ошибочное прерывание, когда значение равно 1.
- Бит 0 определяет ошибочное прерывание при обработке дескриптора.

18.3.4.5. Регистр состояния команды

Данный регистр предназначен для определения состояния выполнения команды. Он является 32-разрядным и доступен для чтения и записи. Значение по умолчанию равно `00000000h`. Формат регистра представлен в табл. 18.60.

Таблица 18.60. *Формат регистра состояния команды*

Биты	Описание
31—24	Состояние
23	Состояние прерывания
22—8	Резерв
7—0	Тип команды

Приведем краткое описание таблицы.

- Биты 31—24 определяют текущее состояние выполнения команды. При этом используются только биты 7—0. Если бит 7 установлен в 1, то это значит команда выполняется, иначе (значение 0) выполнение команды завершено. Биты 6—1 зарезервированы и должны быть установлены в 0. Бит 0 установлен в 1, если в процессе выполнения команды произошла ошибка.
- Бит 23 определяет значение бита 9 в регистре статуса после выполнения команды. Если бит установлен в 1, то значит бит 9 также будет равен 1. Кроме того, хостом будет сгенерировано прерывание.
- Биты 22—8 зарезервированы и должны быть установлены в 0.
- Биты 7—0 определяют тип команды, которая будет выполнена. Доступны следующие значения:
  - 0 — резерв;
  - 1 — команда `Add MMC IE`;
  - 2 — команда `Remove MMC IE`;
  - 3 — команда `Set WUSB MAS`;
  - 4 — команда `Channel Stop`;
  - 5 — команда `Remote Wake Poll Enable`;
  - 6 — 255 зарезервированы.

18.3.4.6. Регистр адреса

Данный регистр предназначен для получения и установки адреса, по которому хост будет обращаться к данным. Он является 64-разрядным и доступен для чтения и записи. Формат регистра представлен в табл. 18.61.

Таблица 18.61. *Формат регистра адреса*

Биты	Описание
63—0	Указатель на адрес в памяти

18.3.4.7. Регистр информации

Данный 64-разрядный регистр позволяет получить адрес в памяти, по которому расположены данные, описывающие текущее устройство. Регистр доступен только для чтения. Формат регистра представлен в табл. 18.62.

Таблица 18.62. *Формат регистра информации*

Биты	Описание
63—6	Указатель на буфер, в котором хранится информация об устройстве
5—0	Биты зарезервированы и должны быть равны 0

18.3.4.8. Регистр ключа безопасности

Данный 32-разрядный регистр предназначен для установки или удаления ключа безопасности. Значение по умолчанию равно 0. Формат регистра представлен в табл. 18.63.

Таблица 18.63. *Формат регистра ключа безопасности*

Биты	Описание
31	Установка ключа
30	Удаление ключа
29—9	Резерв
8	Группа ключа
7—0	Индекс ключа в таблице

Приведем краткое описание таблицы.

- Бит 31, установленный в 1, позволяет установить ключ шифрования.
- Бит 30, установленный в 1, позволяет удалить ключ шифрования.
- Биты 29—9 зарезервированы и должны быть установлены в 0.
- Бит 8 определяет использование текущего ключа как нового в группе.
- Биты 7—0 определяют индекс ключа в таблице.

Данный регистр связан с регистром списка ключей, который здесь не рассматривается.

18.3.4.9. Регистр канала времени

Данный регистр доступен только для чтения и возвращает информацию о текущем времени, заданном для канала. Значение по умолчанию равно `00000000h`. Формат регистра представлен в табл. 18.64.

Таблица 18.64. *Формат регистра канала времени*

Биты	Описание
31—24	Зарезервированы и должны быть установлены в 0
23—0	Значение времени

На этом мы завершим описание регистров. Более подробную информацию можно получить в спецификации стандарта.

18.3.5. Команды и события

Для обработки команды требуется передать данные контроллеру в определенном формате. Мы рассмотрим основные структуры данных для работы с беспроводными устройствами.

Структура команды состоит из непрерывного набора значений. Первое определяет саму команду, а остальные описывают дополнительные параметры команды (табл. 18.65).

Таблица 18.65. *Структура команды*

Смещение, байт	Описание
0 (7—0)	Тип команды (00h — основная, 01h — EFh зарезервированы и F0h — FFh определяются производителем)
1 (23—8)	Код команды
3 (31—24)	Идентификатор команды (допустимо одно значение FEh)
4 (32—64)	Описание первого параметра
N	Описание параметра N

Как видно из таблицы, вначале передается команда, а за ней следуют 1 или большее число параметров. Список стандартных команд представлен в табл. 18.66.

Таблица 18.66. *Стандартные команды*

Значение кода команды	Описание
16	CHANNEL_CHANGE
17	DEV_ADDR
18	GET_IE
19	RESET

Таблица 18.66 *(окончание)*

Значение кода команды	Описание
20	SCAN
21	SET_BEACON_FILTER
22	SET_DRP_IE
23	SET_IE
24	SET_NOTIFICATION_FILTER
25	SET_TX_POWER
26	SLEEP
27	START_BEACONING
28	STOP_BEACONING
29	BP_MERGE
30	SEND_COMMAND_FRAME
31	SET_ASIE_NOTIFICATION

В результате выполнения команды будет возвращен (записан в регистр ошибок) код подтверждения. Возможные значения кодов показаны в табл. 18.67.

Таблица 18.67. *Стандартные коды подтверждения*

Значение кода	Описание
0	SUCCESS
1	FAILURE
2	FAILURE_HARDWARE
3	FAILURE_NO_SLOTS
4	FAILURE_BEACON_TOO_LARGE
5	FAILURE_INVALID_PARAMETER
6	FAILURE_UNSUPPORTED_PWR_LEVEL
7	FAILURE_INVALID_IE_DATA
8	FAILURE_BEACON_SIZE_EXCEEDED
9	FAILURE_CANCELLED
10	FAILURE_INVALID_STATE
11	FAILURE_INVALID_SIZE
12	FAILURE_ACK_NOT_RECEIVED

Таблица 18.67 *(окончание)*

Значение кода	Описание
13	FAILURE_NO_MORE_ASIE_NOTIFICATION
255	FAILURE_TIME_OUT

Структура события так же, как и структура команды, состоит из блока непрерывных значений. Первые 4 байта описывают само событие, а остальные определяют дополнительные параметры. Формат структуры события показан в табл. 18.68.

Таблица 18.68. *Структура события*

Смещение, байт	Описание
0 (7—0)	Тип события (00h — основной, 01h — EFh зарезервированы и F0h — FFh определяются производителем)
1 (23—8)	Код события
3 (31—24)	Если событие инициализировано командой, значение должно совпадать с идентификатором команды. Значение 00h указывает, что событие порождено не командой
4 (32—64)	Описание первого параметра
N	Описание параметра N

Список стандартных кодов событий представлен в табл. 18.69.

Таблица 18.69. *Стандартные события*

Значение кода события	Описание
0	IE_RECEIVED
1	BEACON_RECEIVED
2	BEACON_SIZE_CHANGE
3	BPOIE_CHANGE
4	BP_SLOT_CHANGE
5	BP_SWITCH_IE_RECEIVED
6	DEV_ADDR_CONFLICT
7	DRP_AVAILABILITY_CHANGE
8	DRP

Таблица 18.69 *(окончание)*

Значение кода события	Описание
9	BP_SWITCH_STATUS
10	COMMAND_FRAME_RECEIVED
11	CHANNEL_CHANGE_IE_RECEIVED

Каждая команда и событие имеют свой формат данных. Однако здесь все это рассматривать не будем. Читатели смогут самостоятельно познакомиться с этой информацией, которая подробно представлена в спецификации контроллера Wireless.

Для программирования контроллера беспроводных устройств также можно воспользоваться функциями менеджера устройств и универсальной функцией ввода-вывода `DeviceIoControl`. Принцип работы с ними мало чем отличается от программирования устройств USB или IEEE 1394. Аналогично можно работать и с другим беспроводным интерфейсом — Bluetooth.

ПРИЛОЖЕНИЕ 1

Глоссарий

- **AGP** (Accelerated Graphics Port) — специально выделенный высокоскоростной порт для наилучшей передачи графических трехмерных данных от видеоадаптера в системную память.
- **ANSI** (American National Standards Institute) — негосударственная американская организация, определяющая различные стандарты для необязательного применения.
- **API** (Application Programming Interface) — прикладной интерфейс программирования, предоставляющий низкоуровневые средства доступа к ресурсам базовой системы (например, операционной).
- **ASCII** (American Standard Code for Information Interchange) — стандартный американский код для обмена информацией, выполняющий преобразование символов в цифровую форму.
- **ATA** (AT Attachment) — интегрированная шина обмена данными между дисковой подсистемой и процессором, используемая наряду с IDE.
- **ATAPI** (AT Attachment Packet Interface) — набор программных и аппаратных ресурсов, описывающих интерфейс (на базе шины ATA) между процессором и устройствами чтения (записи) компакт-дисков.
- **BIOS** (Basic Input/Output System) — базовая система ввода-вывода, осуществляющая низкоуровневый доступ к аппаратным ресурсам системы.
- **Bluetooth** — интерфейс, осуществляющий обмен данными посредством беспроводных сетей.
- **CDFS** (Compact Disc File System) — файловая система, разработанная для представления и доступа к данным на компакт-диске.
- **CD-ROM** (Compact Disc Read-only Memory) — оптическое постоянное устройство хранения данных, доступных только для чтения.

- **COM** (Component Object Model) — объектная модель для независимого многоуровневого представления различных системных и определяемых пользователем компонентов.
- **DDC** (Display Data Channel) — канал обмена данными между дисплеем и видеоадаптером.
- **DDK** (Device Driver Kit) — набор программных компонентов для разработки драйверов. Для разных версий Windows имеются собственные пакеты разработчика драйверов.
- **DIB** (Device-independent Bitmap) — формат графического представления растровых изображений в виде, не зависимом от устройства отображения. Другими словами, любая графика в этом формате будет одинаково выглядеть на различных системах.
- **DLL** (Dynamic Link Library) — двоичный тип файла, динамически загружаемый по мере необходимости в область памяти вызывающей программы. Он используется в первую очередь для разделения ресурсов и программных интерфейсов, тем самым уменьшая размер исполняемого файла.
- **DMA** (Direct Memory Access) — прямой доступ к памяти, позволяющий устройству передавать или получать данные без участия процессора, что значительно экономит время и системные ресурсы.
- **DPMS** (Display Power Management Signaling) — стандарт управления питанием дисплея.
- **DSP** (Digital Signal Processor) — цифровой сигнальный процессор, предназначенный для цифровой обработки при высокоскоростной передаче данных. Наиболее часто применяется в устройствах связи и звуковых платах.
- **ECP** (Extended Capabilities Port) — асинхронный 8-разрядный канал для параллельной передачи данных от компьютера на устройство и обратно.
- **FAT** (File Allocation Table) — таблица размещения файловых объектов на носителе, позволяющая упорядочить хранение и доступ к данным.
- **FDC** (Floppy disk Drive Controller) — контроллер накопителя на гибких дисках.
- **GUID** (Globally Unique Identifier) — уникальный глобальный идентификатор, представляющий собой 16-байтовое значение, гарантирующее корректный доступ к указанному объекту в операционной системе.
- **HCD** (Host Controller Driver) — устройство ведущего контроллера. В контексте шины USB оно предназначено для задающего устройства управления чипсетом периферийного оборудования.
- **HCI** (Host Controller Interface) — интерфейс ведущего контроллера, используемый на шине USB.

- **HDC** (Hard Disk I/O Controller) — контроллер ввода-вывода жесткого диска.
- **HID** (Human Interface Device) — интерфейс устройства, используемого на шине USB. I2C (Inter-Integrated Circuit Bus) — последовательная шина передачи данных по двум проводам (синхронизация и данные).
- **IEEE 1394** — цифровой интерфейс, использующий последовательную шину.
- **IrDA** (Infra red Data Assotiation) — интерфейс связи, основанный на открытом оптическом канале в инфракрасном диапазоне.
- **IDE** (Integrated Device Electronics) — дисковое устройство хранения данных с интегрированным контроллером.
- **MIDI** (Musical Instrument Digital Interface) — цифровой интерфейс описания музыкальных инструментов, позволяющий с помощью компьютера управлять различными музыкальными устройствами.
- **NMI** (Nonmaskable Interrupt) — немаскируемое прерывание. Данный тип прерывания является приоритетным и не может быть аннулирован другим запросом на обслуживание.
- **OLE** (Object Linking and Embedding) — технология, позволяющая встраивать внешние (принадлежащие другим приложениям) объекты в собственную программу.
- **PCI** (Peripheral Component Interconnect) — межкомпонентная 32- или 64-разрядная шина, объединяющая различные устройства.
- **PIC** (Programmable Interrupt Controller) — программируемый контроллер прерываний.
- **POST** (Power-On Self-Test) — самотестирование компьютерной системы после включения питания, инициируемое BIOS.
- **SMART** (Self-monitoring, analysis, and reporting technology) — технология, представляющая собой систему автоматического самотестирования, сбора и анализа данных, используемых производителями устройств хранения данных для диагностики и выявления на ранней стадии потенциальных проблем. Позволяет не только определить ошибки в работе оборудования, но и предотвратить потерю данных.
- **SVGA** (Super VGA) — стандарт дисплея с высоким разрешением (не менее 1024×768) и качеством изображения.
- **USB** (Universal Serial Bus) — универсальная последовательная шина, поддерживающая двунаправленный обмен данными и динамическое подключение (при включенном питании компьютера) устройств.

- **VGA** (Video Graphics Array) — стандарт дисплея, поддерживающего разрешение цветного экрана не менее 640×480 пикселов.
- **Win32 API** — прикладной 32-разрядный интерфейс программирования для всех операционных систем Windows.
- **Wireless LAN** — высокоскоростной беспроводный интерфейс обмена данными.

ПРИЛОЖЕНИЕ 2

Описание компакт-диска

Прилагаемый к книге компакт-диск содержит исходные коды всех примеров и системные драйверы для работы с аппаратными портами ввода-вывода. Примеры расположены в папке **Examples**, а драйверы — в **Drivers**. Нумерация примеров совпадает с соответствующими номерами листингов. Выбор драйвера зависит от установленной операционной системы: для Windows 9X/ME предназначен драйвер Io32port.vxd, а для Windows NT/2000/XP/SR3/Vista — IOtrserv.sys. Чтобы использовать драйвер, достаточно скопировать его в папку с проектом, снять атрибут "только для чтения" и добавить код класса CIO32 или CIO32NT, описанный в *главе 1*.

Дополнительно в папке **Documents** собраны различные данные, которые могут понадобиться при самостоятельном программировании аппаратных ресурсов Windows.

Просмотр диска осуществляется только в ручном режиме:

1. Вставьте диск в устройство чтения.
2. Откройте папку **Мой компьютер** (My Computer), расположенную на рабочем столе (Desktop).
3. Выберите из списка устройство чтения компакт-дисков и откройте его.

Внимание!

Предлагаемые драйверы предназначены только для тестирования примеров из данной книги и не должны быть использованы как-либо иначе.

Предметный указатель

А

В

Д

З

И

К

М

Н

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш