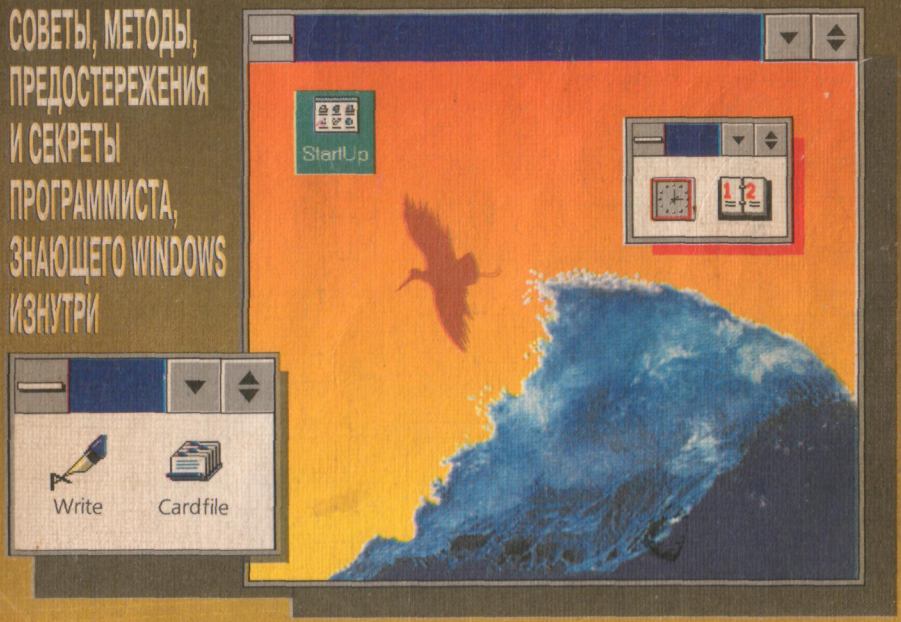
CLUB Computer

Том Сван

WINDOWS

ФОРМАТЫ ФАЙЛОВ

СОВЕТЫ, МЕТОДЫ, ПРЕДОСТЕРЕЖЕНИЯ И СЕКРЕТЫ ПРОГРАММИСТА, ЗНАЮЩЕГО WINDOWS ИЗНУТРИ



BINOM Publishers

CLUB Computer

Уважаемый читатель!

Книгой *"Форматы файлов Windows"* издательство "БИНОМ" открывает новую серию **Computer Club.**

В серии будет представлен широкий спектр книг компьютерной тематики: от книг для начинающих до профессиональных изданий в 1000 и более страниц.

В настоящее время готовятся к изданию около 20 книг серии **Computer Club,** из которых в декабре 1994-го и январе 1995-го годов планируются к выходу в свет следующие:

- **Программирование в Paradox for Windows на примерах (версии 4.5 и 5.0)**
- **Программирование игр для Windows на Borland C++**
- **Сжатие диска**
- **PC изнутри**
- **Руководство по DOS Питера Нортона (версии 6.2 и 6.22)**

Форматы файлов Windows

Tom Swan

Inside Windows File Formats

SAMS
PUBLISHING

Том Сван

Форматы файлов Windows

перевод с английского
Д. А. Зарецкого

БИНОМ
Москва 1995

УДК 681.3.06

Том Сван
Форматы файлов Windows. Пер. с англ. — М.: БИНОМ, 1994 — 288 с.: ил.
ISBN 5-7503-0014-5

Эта книга является детальным руководством по многим файлам, поставляемым вместе с Microsoft Windows. Рассматриваются такие типы файлов, как графические растровые изображения (.BMP), курсоры (.CUR), пиктограммы (.ICO), шрифты (.FNT), файлы редактора Write (.WRI), календарь (.CAL), картотеки (.CRD), группы (.GRP), информационные файлы программ (.PIF), ресурсы, EXE-файлы и др.

Вам также пригодится приводимый краткий обзор приемов обработки файлов и использования структур языка Си.

ISBN 5-7503-0014-5

Оглавление

ПРЕДИСЛОВИЕ

Эта книга была написана по одной простой причине: мне нужен был удобный справочник по структурам файлов Windows, но не пособие по программированию, а непосредственно сосредоточенный на внутренних деталях таких типов файлов, как пиктограммы, курсоры, шрифты, ресурсы, группы, EXE-, PIF- и BMP-файлы, файлы редактора Write, календаря, картотеки и др.

К сожалению, подобной книги не существовало, а вся информация по этому предмету была опубликована беспорядочно и в сильно несовместимых формах. Поэтому я решил написать свой собственный справочник в едином стиле. Это *Форматы файлов Windows* — сжатое детальное руководство по большинству файловых типов Microsoft Windows.

После кратких наставлений по структурам и по технологии файловой обработки в книге описываются форматы нескольких типов файлов Windows. В каждой главе эти форматы показаны двумя способами: структурными диаграммами, содержащими имя поля, его длину и байтовое смещение, а также как структуры языка Си.

Хотя все листинги в книге даны на языке Си, информация пригодна для использования в любых системах разработки программ под Windows, способных читать или записывать дисковые файлы. Чтобы пользоваться данной книгой, желательно немного знать язык Си, но отнюдь не быть Си-гуру. Если вы программируете на таких языках, как Visual BASIC, Turbo Pascal for Windows, то диаграммы файловых структур, приведенные здесь, окажутся особенно полезными для написания программ доступа к файлам Windows. Листинги структур совместимы со всеми компиляторами Си и Си++.

В будущих редакциях я надеюсь исследовать большее число файловых типов Windows. Сообщите мне, какой из них вы хотели бы видеть в следующей книге. Вы можете связаться со мной через издательство или через электронную почту. Я с удовольствием выслушаю вас.

Том Сван
CompuServ ID: 73627,3241

Глава 1
ВВЕДЕНИЕ

В этой главе рассматриваются требования к программному обеспечению и аппаратуре, благодаря которым вы извлечете максимальную пользу, работая с данной книгой.

Программы

- MS-DOS 3.3 или более поздней версии
- Windows 3.1 или более поздней версии
- Компилятор Си или Си++ либо иной язык программирования для Windows

Аппаратура

- 100% совместимый персональный компьютер (рекомендуется 80386 или выше).
- Графический дисплейный адаптер (рекомендуется VGA).
- Жесткий диск с не менее чем 2,5 мегабайтами свободного пространства.
- Флоппи-дисковод (рекомендуется 3,5").
- Мышь (рекомендуется).

Как работать с книгой

В этой книге рассматриваются структуры некоторых файлов в операционной системе Windows. Критерии отбора описываемых файлов таковы:

- поставка вместе с Windows,
- однозначная идентификация по расширению имени,
- наличие интереса со стороны программистов в плане обработки этих файлов собственными средствами.

Примерами файлов, удовлетворяющих данным критериям, могут служить файлы растровых изображений (.BMP), файлы календаря (.CAL), файлы картотек (.CRD), файлы групп (.GRP) и др. Полный список рассматриваемых файлов вы найдете в оглавлении книги.

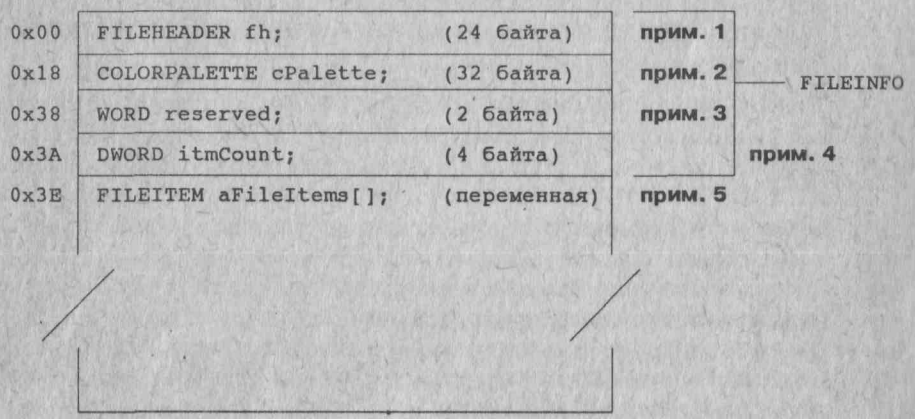
Многие из приведенных здесь структур, программ и прочей информации требуют Windows 3.1. Что-то пригодно и для ранних версий Windows (например, 2.0 или 3.0), но для того чтобы добиться наилучших результатов, вы должны пользоваться новейшими из доступных модификаций Windows.

Каждый файл документируется двумя способами: с помощью диаграммы, в которой даются размеры и относительные внутренние позиции файловых компонентов, и с помощью одной или нескольких структур языка Си, содержащих типы данных и идентификаторы, которые можно использовать в ваших программах.

Структурные диаграммы файлов

Каждый тип файла, рассматриваемый в этой книге, сопровождается одной или несколькими *структурными диаграммами*, в которых показаны имя каждой структурной единицы файла, ее байтовый размер и относительную позицию. На рис. 1.1 демонстрируется формат и особенности структурной диаграммы фиктивного файла с расширением SMP.

Рис. 1.1. Структура фиктивного SMP-файла



В большинстве случаев подобные структурные диаграммы сопровождаются списком примечаний, дающим детальное описание компонентов. Например, следующий список примечаний дает представление об описании большинства структурных диаграмм в книге. (Заодно обратите внимание, как оформляется текст примечаний.)

Примечания к рис. 1.1

1. Приведенный в качестве примера файл начинается 24-байтовым объектом fh типа FILEHEADER. Байтовое смещение (0x00), изображенное в левой колонке, говорит о том, что это первый элемент файла. Следуя практике программирования на Си, будем давать размеры объектов в десятичной форме, а смещения — в шестнадцатеричной. В некоторых случаях, когда объекты переменной длины не позволяют задавать фиксированные байтовые смещения для всех компонентов, см. текстовые примечания о том, как получить доступ к таким структурным единицам.
2. Вторым компонентом файла является 32-байтовая структура cPalette, имеющая тип COLORPALETTE и располагающаяся ровно с 0x18 (десятичное 24) байта от начала файла. Чтобы подробнее описать этот объект, примечание, подобное тому, что вы сейчас читаете, может отослать вас к другой структурной диаграмме, которая описывает внутреннее устройство поля cPalette. В данном конкретном случае нет других диаграмм, но имейте в виду, что вам может потребоваться изучить несколько структурных диаграмм, чтобы получить полную картину содержимого файла.
3. Зарезервированные объекты либо не используются и не документируются, либо предназначены для свободного употребления. Несколько зарезервированных объектов, как правило, обозначаются reserved1, reserved2 и т.д. Обычно при записи файлов зарезервированные объекты должны быть обнулены, а при чтении — пропущены.
4. Линии, объединяющие три предыдущих примечания, говорят о том, что далее имеется Си-структура FILEINFO, состоящая из полей fh, cPalette, reserved и itemCount. Использование структур, подобных FILEINFO, часто необязательно, т.к. в диаграмме уже имеется достаточно информации для индивидуального чтения/записи компонентов файла. Однако бывает удобно определить переменную типа FILEINFO для хранения данного фрагмента файла или для того, чтобы передать эту переменную как параметр функции. Заметьте также, что FILEINFO располагается вне основного прямоугольника диаграммы, показывая тем самым, что данная структура образует более высокий уровень иерархии компонентов файла.

5. Последний объект в нашем файле — массив переменной длины, содержащий один или несколько объектов типа FILEITEM. В графе, содержащей длины полей, в этом случае пишется: (переменная), что предполагает некоторые другие условия определения размера объекта. Например, переменная itemCount может содержать число элементов массива. Программа должна использовать это значение, чтобы выделить достаточное количество памяти для всего массива или применить itemCount в качестве счетчика цикла поэлементного чтения или записи массива. Массивы переменной длины могут даже вообще не иметь элементов, т.е. отсутствовать в файле. Обращайте внимание на такую возможность, обычно указываемую в примечаниях, но не столь очевидную из структурной диаграммы. (См. гл. 2 для более подробной информации об объектах переменной длины.)

Структурные описания файлов

В каждой главе вслед за структурными диаграммами файла следуют одна или несколько объявлений структур на языке Си, демонстрирующих использование содержимого файла в программах. Имеются три типа таких структур:

официальное объявление, как, например, BITMAPFILEHEADER, находящееся в windows.h или в других файлах, поставляемых вместе с вашим языком программирования;

новое объявление, которое отсутствует в windows.h, но может приводиться в той или иной форме в других книгах по программированию в среде Windows, хотя иногда с другим наименованием;

Си-подобная псевдоструктура, носящая чисто информационный характер и не использующаяся непосредственно для определения программных переменных.

Если вы программируете на Си или на Си++, то можете, разумеется, использовать любую структуру, объявленную в windows.h. Только добавьте следующую строку в текст вашей программы:

```
#include <windows.h>
```

На протяжении всей книги моей целью было единообразие описания всех структур. В некоторых случаях для этого потребовалось изменить некоторые идентификаторы, упоминаемые в других публикациях. Иногда я был вынужден вводить новые имена. Мною также были исправлены некоторые неточности в официальных описаниях структур. Во всех таких случаях различия между описанием файлов в данной книге и аналогичной информацией, опубликованной где-то еще, указываются особо.

Совет. Разумеется, все Windows-программы, написанные на Си, *должны* включать windows.h. Но вы можете написать программу, работающую только в среде DOS и также включающую этот заголовок, иными словами, вам не нужно писать Windows-программу, чтобы работать с Windows-файлами. Поскольку немалая доля процесса программирования под Windows посвящена проблемам интерфейса и не имеет отношения к обработке файлов, часто бывает проще написать для этих целей программу под DOS. (Попутно обратите внимание, как оформляется в книге тексты советов, первый из которых вы читаете в настоящий момент.)

В листинге 1.1 приводится пример структуры на языке Си для объекта типа FILEINFO, изображенного на рис. 1.1. В реальных файлах Windows будут приведены также структуры других именованных объектов, таких, как FILEHEADER и COLORPALETTE. Идущие вслед за листингами и помеченные черными квадратиками комментарии поясняют назначение каждого поля структуры.

Листинг 1.1. FILEINFO

```
typedef struct tagFILEINFO {
  FILEHEADER fh;
  COLORPALETTE cPalette;
  WORD reserved;
  DWORD itemCount;
} FILEINFO;
```

- fh — содержит общую информацию о файле.
- cPalette — определяет значения цветов.
- reserved — не используется и не документируется.
- itemCount — число элементов в массиве переменной длины, непосредственно следующего за этим полем.

Ключевое слово typedef, начинающее объявление структуры, создает новый тип данных, который упрощает работу по определению структурных переменных. Два слова — struct tagFILEINFO — официально объявляют структуру. FILEINFO — новый тип данных, создаваемый с помощью typedef, — всегда пишется прописными буквами. Переменную типа FILEINFO можно определить в программе следующим образом:

```
FILEINFO fInfo;
```

Этот оператор эквивалентен следующей длинной форме, которая может быть названа "классическим определением структуры в языке Си":

```
struct tagFILEINFO fInfo;
```

Короче говоря, FILEINFO — *псевдоним* структуры tagFILEINFO. Не имеет значения, какой способ вы применяете — оба они создают переменную fInfo. Однако большинство программистов предпочитают более краткий способ определения, считая его более ясным.

Программисты на языке Си++ могут дополнительно упростить объявление структур, убрав ключевое слово typedef вместе с прототипом имени (начинающегося на tag). Например, в Си++ предыдущая структура без учета ее элементов имеет вид:

```
struct FILEINFO {
...
};
```

Имея такое модифицированное объявление в Си++, но не в Си, вы можете определить переменную fInfo следующим образом:

```
FILEINFO fInfo;
```

Предупреждение. Объявления структур в этой книге совместимы с Си и Си++. Однако, если вы уберете ключевое слово typedef, как только что предлагалось, приведенный выше оператор будет работать только в Си++. По этой причине, видимо, будет лучше использовать длинные объявления и псевдонимы, как напечатано в книге, а не рисковать, делая ваши программы несовместимыми с Си. (Попутно обратите внимание, как оформляются в книге тексты предупреждений о возможных конфликтах.)

В тех случаях, когда в windows.h объявляется несколько псевдонимов типа для структуры, последняя также воспроизводится здесь. Например типичное объявление структуры может сопровождаться псевдонимами, которые просто определяют указатели на структурную переменную. Вместе с дополнительными псевдонимами полное объявление структуры FILEINFO (без ее элементов) может выглядеть следующим образом:

```
typedef struct tagFILEINFO {
...
} FILEINFO;
typedef FILEINFO* PFILEINFO;
typedef FILEINFO FAR* LPFILEINFO;
```

Две дополнительные строки, следующие за описанием структуры, создают два псевдонима: PFILEINFO (указатель на объект типа FILEINFO) и LPFILEINFO (длинный, или *дальний* указатель на объект типа FILEINFO). Использовать ли дальние указатели — тема для книги по программированию

на Си, а не для этой. Однако в общем случае вы можете употреблять дальний указатель одним из следующих способов.

Выберите в вашем компиляторе "большую" (large) модель памяти, в которой все указатели являются дальними по умолчанию. Тогда PFILEINFO и LPFILEINFO будут эквивалентны.

Пишите программу так, чтобы она была совместима со всеми моделями памяти. В этом случае PFILEINFO может быть предпочтительнее, т.к. позволяет компилятору выбирать ближний или дальний указатели в зависимости от используемой модели.

Адресуйте переменные, используя явно объявленные ближние или дальние указатели, и применяйте как PFILEINFO, так и LPFILEINFO.

Замечание. Листинги структур включают псевдонимы типа указатель, если только эти имена доступны в windows.h или в других заголовочных файлах. Я не придумывал новые псевдонимы для каждой структуры в книге. (Попутно обратите внимание на оформление текстов замечаний, первое из которых вы сейчас прочитали.)

Несмотря на то, что все файловые структуры в этой книге описаны на Си, вы можете использовать их как справочное руководство при работе на других языках программирования. Например, программисты, пишущие на Turbo Pascal for Windows, могут использовать структурные диаграммы и соответствующие конструкции struct языка Си для построения эквивалентных им записей (данных типа record) Паскаля. Вы можете также применять другие языки программирования для программ обработки файлов Windows, а не писать их исключительно на Си. Однако, если вы вообще не знаете Си или несколько подзабыли его, можете просмотреть справочные главы по Си-структурам, указателям и функциям распределения памяти. Чем больше вы знаете о Си, тем легче поймете материал книги.

Глава 2

ПРИЕМЫ РАБОТЫ СО СТРУКТУРАМИ

В этой книге форматы файлов описаны как типы struct языка Си, и чтобы наилучшим образом понять приведенные примеры, вам необходимо отшлифовать свое искусство работы со структурами. Имеется несколько важных приемов, с некоторыми из которых даже опытные программисты не так хорошо знакомы, как следовало бы. В этой главе рассматриваются следующие вопросы.

- Как определять и использовать глобальные и локальные структуры.
- Как выделять глобальную память Windows для динамических структур.
- Как определять и использовать структуры переменной длины.
- Как определять и использовать большие буфера памяти, которые нужны для обработки структур некоторых видов, например массивов пикселов в графических растровых файлах (bitmap).

Определение глобальных и автоматических структур

Структура — один из самых полезных инструментов языка Си. Глобальные структуры определяются вне функций. *Автоматические структуры* определяются внутри функций. Вы *объявляете* структуру, вставляя ее имя и идентификаторы полей в текст программы. Вы *определяете* структуру, когда выделяете для нее память. Структура есть объект, занимающий память. Слово *объект* здесь не имеет отношения к понятию *объекта класса*. (Эта книга не требует понимания объектно-ориентированного программирования.)

Автоматические структуры локальны для свой определяющей функции, т.е. только определяющая функция может непосредственно использовать эту структуру и последняя автоматически удаляется, когда функция завершается. Простой оператор, резервирующий память для объекта aSrtuct типа ANYSTRUCT, имеет вид:

```
ANYSTRUCT aStruct;
```

Если этот оператор встречается вне функции, он создает глобальный объект в сегменте данных программы. Если этот оператор встречается внутри функции, то возникает автоматический объект, который автоматически создается при вызове данной функции и удаляется по ее окончании. Расположение сегмента данных и стека определяется самой системой Windows.

Замечание. В 386 расширенном режиме Windows процессор адресует память не напрямую, а через *селекторы*, которые отображаются на физические адреса. Windows может также преобразовывать часть пространства жесткого диска в *виртуальную память*. Этот факт означает, что объекты могут вытесняться на диск или перемещаться по памяти, освобождая при этом место для других данных, а затем автоматически возвращаться в память по старым или новым адресам. К счастью, все сказанное происходит без участия прикладной программы, которой в большинстве случаев даже не нужно знать об этих закулисных манипуляциях с памятью.

Для обращения к элементам структуры используется символ точки. Например, если ANYSTRUCT имеет элемент count целого типа, следующий оператор присваивает его значение целой переменной x:

```
x=aStruct.count;
```

Чтобы указать адрес объекта, используйте aStruct с символом амперсанда (&) в начале имени. Например, для указания адреса элемента count структуры aStruct применяйте обозначение &aStruct.count. Вы можете передать aStruct *по адресу* в функцию, параметр которой объявлен как ANYSTRUCT * (указатель на объект типа ANYSTRUCT):

```
f(&aStruct);
```

Чтобы передать aStruct по значению в функцию с параметром типа ANYSTRUCT (а не указатель на объект данного типа), просто укажите имя структуры:

```
f(aStruct);
```

Эти приемы работы со структурами должны быть освещены в любом хорошем руководстве по Си или в курсе по программированию. Если вы не владеете ими, изучите основы программирования на Си, прежде чем продолжить чтение книги.

Предупреждение. Объявление больших структур как глобальных может быстро исчерпать 64 Кбайтовый предел сегмента данных программы. По этой причине самые большие структуры должны быть размещены в динамической памяти Windows, как будет показано далее.

Размещение динамических структур

Динамическая память (memory heap) — это область ОЗУ, существующая специально для *динамических объектов*. Часть этой памяти может быть виртуализирована на диске. Программы получают блоки динамической памяти с помощью специальных функций. После размещения в динамической памяти объект используется точно так же, как и глобальный или автоматический объекты, за исключением того, что динамические объекты адресуются указателями.

Когда работа с динамическим объектом закончена, программа вызывает функцию, которая освобождает выделенную память для распределения ее в будущем другим объектам. В этом случае динамическая память рассматривается как большой буферный пул, в котором перемещаются объекты различных типов. Поскольку динамическая память — *разделяемый ресурс*, для его правильного использования требуется некоторая осторожность в программировании. *Отдавайте себе отчет в том, что ваша программа и все прочие приложения разделяют одну и ту же динамическую память.* Это означает, что ваши объекты хранятся вперемешку с объектами таких приложений, как Excel или Word for Windows — концепция, неожиданная для многих программистов в среде DOS, привыкших рассматривать всю имеющуюся память как доступную для своих программ. (Windows поддерживает неразделяемую локальную динамическую память, но этот ресурс ограничен по размеру и подходит только для небольших объектов.)

Для Windows-программ доступны два вида динамической памяти: *локальная* и *глобальная*. Не путайте их с локальными и глобальными объектами. Вопреки схожести названий эти понятия различны, как огонь и вода. *Глобальный объект* — структура или иной элемент, определенный вне какой-либо функции. *Локальный объект* — один из тех, что может быть использован только определившей его функцией. (Например, автоматическая переменная, определенная внутри функции, локальна по отношению к ней.)

Локальная динамическая память организуется внутри 64 Кбайтового сегмента данных программы, не располагающего достаточным пространством для сотен огромных объектов, типичных для программ Windows. *Глобальная динамическая память* состоит из памяти, которая не используется программами или самой системой Windows. Так как глобальная динамическая память способна хранить объекты размером до 16 Мбайтов в 386 расширенном режиме (до одного Мбайта в стандартном режиме), она должна быть предпочтительнее для размещения больших структур данных, связанных со многими файлами Windows.

Предупреждение. Объект, хранящийся в динамической памяти, является *постоянным*, т.е. продолжает занимать выделенное пространство, даже если породившая его функция завершилась. Как общее правило, важно, чтобы функция удаляла созданные ею динамические объекты либо некоторым способом сохраняла дескриптор или указатель объекта для последующего освобождения выделенной области. В противном случае возникает так называемая *утечка памяти*. Разумеется память не "вытекает" из компьютера, но эффект такой, как будто она начинает просачиваться через дверь, в роли которой выступает ваш жесткий диск. Большинство программистов видели, как Windows замедляется, диск часто дергается и пиктограммы теряют свой цвет. По мере все большего вытекания памяти мышь и клавиатура перестают реагировать, и вам ничего не остается, как перезагрузить компьютер. Утечка памяти — серьезная ошибка, которая должна предотвращаться всеми средствами.

Распределение памяти

Для размещения в глобальной динамической памяти следует вызвать функцию GlobalAlloc(). Интерфейс прикладного программирования (API) Windows объявляет эту функцию следующим образом:

```
HGLOBAL GlobalAlloc(UINT fuAlloc, DWORD cbAlloc);
```

- *fuAlloc — беззнаковое целое, задающее различные опции размещения.*
- cbAlloc — число байтов, запрашиваемое в глобальной динамической памяти. В стандартном режиме на процессоре 80286 предел этого значения ограничен 1 Мбайтом минус 80 (или точно 1.048.496). В 386 расширенном режиме на процессоре 80386 или совместимом с ним предельное значение составляет 16 Мбайт минус 64 Кбайт (точно 16.711.680).

GlobalAlloc() возвращает дескриптор (handle) типа HGLOBAL — целое значение, идентифицирующее выделенную память. Если это значение NULL, функция не смогла выделить область требуемого размера. В против-

ном случае можно считать запрос удовлетворенным и блок памяти готовым для использования.

Имеется несколько опций для конфигурации распределенной памяти, многие из которых имели большее значение в ранних версиях Windows, когда не было поддержки виртуальной памяти. Большинство книг по программированию под Windows и систем оперативной подсказки перечисляют все доступные опции, передаваемые как параметры fuAlloc. Однако наилучшим способом размещения динамической структуры является макрос GPTR, представляющий собой комбинацию опций GMEM_FIXED и GMEM_ZEROINT. Эти две опции гарантируют, что блок будет помечен как "фиксированный" (т.е. неперемещаемый) и сразу после распределения обнулен.

Замечание. Поскольку в 386 расширенном режиме Windows использует виртуальную память, "фиксированные" блоки фактически не стоят на одном месте. Windows может перемещать их по памяти или выгружать на диск для удовлетворения последующих запросов на размещение. Слово "фиксированный" означает, что используемый в программе адрес блока не изменится. Несмотря на перемещение блоков по памяти и даже вытеснение их на диск, вы всегда сможете адресовать их.

Важно ясно представлять, что HGLOBAL, возвращаемый GlobalAlloc(), не является адресом памяти, а лишь идентификатором, который может быть использован другими функциями для ссылок на выделенный блок. Чтобы получить адрес последнего, нужно вызвать функцию GlobalLock(). Вот пример того, как Windows-программа может распределить и адресовать 2048-байтовый блок:

```
HGLOBAL hMem;
void FAR* lpBuffer;
hMem = GlobalAlloc(GPTR, 2048);
lpBuffer = GlobalLock(hMem);
if (!lpBuffer) Error();
```

Первая строка листинга определяет дескриптор hMem типа HGLOBAL для ссылок на выделяемый блок. Вторая строка определяет дальний указатель типа void, названный lpBuffer, который будет адресовать тот же самый блок. В третьей строке вызывается GlobalAlloc() — запрос на 2048 байтов из динамической памяти, используя упоминаемый выше макрос GPTR. В четвертой строке вызывается функция GlobalLock() с hMem в качестве аргумента, а результат функции присваивается указателю lpBuffer. Наконец, если lpBuffer имеет значение, отличное от NULL, то он адресует выделенный блок. (Не нужно проверять, возвратила ли GlobalAlloc() значение NULL, т.к.

если это произошло, GlobalAlloc() завершится аварийно и проблема будет решена, прежде чем что-то сломается.)

Освобождение памяти

С этого момента lpBuffer адресует 2048-байтовый буфер. После того как программа закончила работу с буфером, она должна разблокировать и освободить память, чтобы та могла быть использована для будущих объектами. Это всегда делается с помощью следующей пары функций:

```
GlobalUnlock(hMem);
GlobalFree(hMem);
```

Каждой функции передается дескриптор, возвращаемый GlobalAlloc(). С каждым вызовом GlobalLock() *счетчик блокирования* области памяти увеличивается на единицу. С каждым вызовом GlobalUnlock() счетчик блокирования области уменьшается на единицу. GlobalFree() будет в состоянии освободить блок только в том случае, когда счетчик блокировок равен нулю. Таким образом, прежде чем освобождать память, для каждого дескриптора следует вызывать GlobalUnlock() столько раз, сколько вызывалась GlobalLock().

Использование указателей

Некоторые программисты не любят определять для каждого блока памяти как указатель, так и дескриптор. С помощью маленького трюка для адресации выделяемых блоков глобальной динамической памяти Windows можно обойтись только указателем. Вот типичный способ распределения size байтов и адресации их указателем p:

```
p = GlobalLock(GlobalAlloc(GPTR, size));
```

Если указатель p не NULL, он готов для работы. Так как этот прием исключает использование дескриптора, возвращаемого GlobalAlloc(), другим функциям нужно как-то найти его значение, чтобы иметь возможность освободить занимаемую память. Для этого вызывайте GlobalHandle(), как показано в следующем предложении языка Borland C++:

```
HANDLE hMem = LOWORD(GlobalHandle(FP_SEG(p)));
if (hMem)
  if (GlobalUnlock(hMem))
    GlobalFree(hMem);
```

Функции GlobalHandle() требуется селектор, получаемый в результате выполнения макроса FP_SEG (сегмент дальнего указателя) для указателя p. Другие компиляторы Си и языки программирования, вероятно, предложат

похожие средства для получения адреса сегмента дальнего указателя, который фактически является селектором, а не физическим адресом. Младшее слово значения, возвращаемого GlobalHandle(), есть дескриптор блока памяти, а старшее слово содержит селектор. Если значение дескриптора не NULL, можно передавать его функциям GlobalUnlock() и GlobalFree() для разблокирования и освобождения выделенной памяти.

Замечание. Windows позволяет использовать максимум 8000 селекторов во всех приложениях. Это значит, что не более 8000 объектов могут быть размещены в глобальной динамической памяти одновременно. Для хранения большего числа объектов требуется разместить буфера большого размера, способные содержать несколько объектов. Например, если каждый буфер может содержать четыре объекта, то максимальное число объектов возрастет до 32000 (8000 буферов по четыре объекта в каждом).

Размещение и удаление структур

Для размещения структур с помощью GlobalAlloc() вызовите функцию sizeof(), чтобы получить размер в байтах, и используйте выражение приведения типов для присвоения значения, возвращаемого GlobalLock(), указателю на структуру. Приведем типичную последовательность размещения объекта типа ANYSTRUCT:

```
HGLOBAL hMem;
LPANYSTRUCT lpStruct;
hMem = GlobalAlloc(GPTR, sizeof(ANYSTRUCT));
lpStruct = (LPANYSTRUCT)GlobalLock(hMem);
if (!lpBuffer) Error();
```

Если тип LPANYSTRUCT не определен с помощью typedef, можно также использовать ANYSTRUCT FAR *

```
HGLOBAL hMem;
ANYSTRUCT FAR * lpStruct;
hMem = GlobalAlloc(GPTR, sizeof(ANYSTRUCT));
lpStruct = (ANYSTRUCT FAR *)GlobalLock(hMem);
if (!lpBuffer) Error();
```

Удалите размещенную структуру, используя ее дескриптор, как было показано ранее. Работайте со структурой, как с любым адресуемым указателем объектом. Например, если структура, адресуемая указателем lpStruct, имеет элемент count, его значение можно присвоить переменной x, используя оператор ->

```
x = lpStruct -> count;
```

Использование malloc() и free()

В большинстве компиляторов Си и Си++ имеются стандартные функции malloc() и free() для выделения и освобождения блоков глобальной динамической памяти. Эти функции вызывают GlobalAlloc() и GlobalFree(), так что конечный результат подобен описанному в предыдущих разделах.

Однако в некоторых компиляторах (например, Borland C++) функция malloc() модифицирована с целью резервирования памяти кусками, которые разделены на еще меньшие порции. Если прикладная программа запрашивает память с помощью malloc(), она может получить лишь часть ранее выделенного куска. И только когда нет доступных кусков, malloc() затребует дополнительное пространство у Windows. Таким образом, потенциальное количество объектов увеличивается. Например, если каждый кусок может содержать четыре объекта, то программа в состоянии создать 32000 динамических структур — по четыре объекта для каждого из 8000 селекторов. Тем не менее на практике соревнование за глобальную динамическую память препятствует монополизации единственной программой всех селекторов и точный максимум, видимо, меньше, чем теоретический предел.

Применяйте malloc() и free() под Windows так же, как и под DOS. Во-первых, включите соответствующий заголовочный файл alloc.h. Затем используйте следующие операторы для определения указателя структуры и выделения памяти под структуру:

```
LPANYSTRUCT pStruct;
pStruct = (LPANYSTRUCT)malloc(sizeof(ANYSTRUCT));
if (!pStruct) Error();
```

Для удаления объекта передайте его указатель функции free():

```
free(pStruct);
```

Совет. Может оказаться полезным выделять память кусками не только с помощью malloc(), но и используя GlobalAlloc() непосредственно. Например, программа может зарезервировать *массив* объектов, вызвав GlobalAlloc() однажды, а не всякий раз при размещении каждого объекта. Этот прием позволяет вам точно знать, сколько памяти приходится на один кусок.

Использование new и delete в Си++

Вы можете использовать операторы Си++ new и delete для размещения и удаления объектов в памяти Windows. Обычно эти операторы реализуются через вызовы malloc() и free() и, следовательно, будут работать, как описано в предыдущем разделе. В общем случае оператор Си++

```
pStruct = new ANYSTRUCT;
```

эквивалентен оператору Си

```
pStruct = (LPANYSTRUCT)malloc(sizeof(ANYSTRUCT));
```

а следующая строка в Си++

```
delete pStruct;
```

эквивалентна оператору Си

```
free(pStruct);
```

Замечание. Если вы не планируете писать программы на Си++ с использованием объектно-ориентированной технологии, не будет большого выигрыша от использования new и delete вместо malloc() и free(). Тем не менее многие находят, что new и delete понятнее и, следовательно, удобнее.

Другие функции и макросы

Дополнительно к функциям и макросам, рассмотренных до сих пор в этой главе, для написания программ обработки файлов могут оказаться полезными следующие функции и макросы.

Функция GetFreeSpace()

Функция GetFreeSpace() определяет максимальное число байтов, доступное в динамической памяти Windows. Объявляется следующим образом:

```
DWORD GetFreeSpace(UINT fuFlags);
```

- fuFlags специфицирует опции флажков. Windows 3.1 игнорирует это значение, которое должно быть нулевым.

В стандартном режиме GetFreeSpace() возвращает точное число доступных байтов. В 386 расширенном режиме функция возвращает приближенную оценку доступного пространства.

Предупреждение. Даже если GetFreeSpace() показывает достаточное для структуры количество памяти, было бы неверно полагать, что последующие вызовы GlobalAlloc() пройдут безошибочно. Не забывайте, что другие программы, работающие в фоновом режиме, могут запрашивать области глобальной динамической памяти. Кроме того, в 386 расширенном режиме GlobalAlloc() возвращает приближенное, а не точное значение свободного пространства. Всегда проверяйте результат, возвращаемый GlobalAlloc(), для гарантии того, что память действительно была распределена.

Функция GlobalCompact()

Функция GlobalCompact() перетасовывает области глобальной динамической памяти с целью выделения запрошенного пространства. Объявляется следующим образом:

```
DWORD GlobalCompact(DWORD dwMinFree);
```

- dwMinFree — число запрашиваемых байтов. Обычно это значение передается затем функции GlobalAlloc(). Установите dwMinFree равным нулю, чтобы получить наибольшее свободное пространство, которое будет доступно после уплотнения динамической памяти.

Совет. Чтобы освободить максимально возможное пространство, передайте -1 функции GlobalCompact(). Поскольку при этом из динамической памяти удаляются все выгружаемые блоки, программа может получать таким способом огромные области, если, разумеется, на компьютере установлено большое количество микросхем памяти. Однако повторная загрузка вытесненных блоков требует времени, поэтому имейте в виду, что этот трюк может негативно влиять на производительность вашей и других программ.

Классическим способом распределения памяти в Windows является вызов GlobalCompact() непосредственно перед GlobalAlloc(). Передавайте одинаковое число запрашиваемых байтов обеим функциям.

```
HGLOBAL hMem;
LPANYSTRUCT lpStruct;
GlobalCompact(sizeof(ANYSTRUCT));
hMem = GlobalAlloc(GPTR, sizeof(ANYSTRUCT));
```

Вызов GlobalCompact() не является обязательным, однако делает более вероятным успешное завершение GlobalAlloc().

Замечание. В 386 расширенном режиме GlobalCompact() не имеет такого значения, как в стандартном режиме, т.к. Windows может автоматически выгружать на диск блоки памяти, чтобы освободить место для GlobalAlloc(). Это исключает необходимость уплотнения динамической памяти посредством GlobalCompact(), если GlobalFree() сообщает о наличии достаточного свободного пространства.

Функция GlobalReAlloc()

Функция GlobalReAlloc() изменяет размер или опции ранее размещенных объектов. Объявляется следующим образом:

```
HGLOBAL GlobalReAlloc(HGLOBAL hglbl, DWORD cbNewSize, UINT fuAlloc);
```

- hglbl — дескриптор объекта, возвращаемый предыдущим вызовом GlobalAlloc() или GlobalReAlloc().
- cbNewSize — новый запрашиваемый размер блока памяти.
- fuAlloc — опции, идентичные параметру fuAlloc функции GlobalAlloc().

Эта функция возвращает новый дескриптор повторно размещенного объекта требуемого размера. Если объект не может быть переразмещен, функция возвращает NULL.

Предупреждение. Никогда не присваивайте результат функции GlobalReAlloc() переменной hglbl. Если функция возвратит NULL, исходный дескриптор будет потерян. Всегда присваивайте результат функции временной переменной. Если ее значение не NULL, вы можете потом скопировать его в исходный дескриптор.

Макрос FIELDOFFSET

Используйте макрос FIELDOFFSET для определения адресного смещения элемента структуры, другими словами, относительную байтовую позицию элемента внутри структуры. Макрос определен в windows.h следующим образом:

```
int FIELDOFFSET(structname, member);
```

- structname — имя некоторой структуры.
- member — элемент, объявленный в структуре.

Макрос возвращает значение целого типа, равное байтовому смещению элемента внутри структуры. Используйте этот макрос для вычисления отно-

сительной позиции элемента, а не полагайте, что он всегда находится на фиксированном расстоянии от начала структуры. Этот прием особенно полезен для тех структур, которые могут измениться в более поздних версиях программы.

Замечание. Имейте в виду, что макрос FIELDOFSET() и следующие макросы в этом разделе не являются функциями, хотя для простоты и преподносятся в "функциональной" форме.

Макросы HIBYTE и LOBYTE

Используйте эти макросы для выделения старшего и младшего байтов в 16-разрядных беззнаковых целых числах. Макросы определяются в windows.h следующим образом:

```
BYTE HIBYTE (UINT uVal);
BYTE LOBYTE (UINT uVal);
```

- uVal — беззнаковое 16-разрядное значение, из которого выделяются старший и младший байты.

Совет. Макросы HIBYTE и LOBYTE чрезвычайно полезны при обработке файлов. Программа может читать данные из файла 16-разрядными словами, а затем с помощью макросов работать с отдельными байтами. Это сокращает количество операций ввода-вывода наполовину, и, следовательно, убыстряет программы по сравнению с теми, где используется побайтовый ввод-вывод.

Макросы HIWORD и LOWORD

Макросы HIWORD и LOWORD выделяют старшее и младшее 16-разрядные слова из 32-разрядного беззнакового целого, называемого также "двойным словом", или DWORD. Макросы определены в windows.h следующим образом:

```
WORD HIWORD(DWORD dwVal);
WORD LOWORD(DWORD dwVal);
```

- dwVal — беззнаковое 32-разрядное значение, из которого выделяются старшее и младшее 16-разрядные слова.

Совет. Для доступа к отдельным байтам 32-разрядного двойного слова передавайте результат HIWORD или LOWORD макросам HIBYTE или LOBYTE.

Макрос MAKELONG

Часто бывает необходимо запаковать два 16-разрядных целых в одно двойное слово, перед тем как записать его в файл (или присвоить его структуре, которая будет записана на диск). С этой целью используется макрос MAKELONG, определяемый в windows.h следующим образом:

```
DWORD MAKELONG(UINT uLow, UINT uHigh);
```

- uLow — беззнаковое 16-разрядное число, подлежащее вставке в младшее слово.
- uHigh — беззнаковое 16-разрядное число, подлежащее вставке в старшее слово.

Макрос MAKELP

Используйте макрос MAKELP, чтобы создать новый указатель на объект (например, внутри буфера ввода-вывода). Макрос определен в windows.h следующим образом:

```
void FAR* MAKELP(WORD wSel, WORD wOff);
```

- wSel — селектор сегмента в таком же формате, как и возвращаемый функцией GlobalLock().
- wOff — Байтовое смещение от начала сегмента, адресуемого wSel.

Предупреждение. Адресация объектов в буферах, размер которых превышает 64K, требует специальных приемов. См. "Адресация больших структур" для информации по этому хитрому вопросу.

Использование структур переменной длины

Структуры переменной длины, или *структуры с открытым концом*, — самая обычная вещь в обработке файлов Windows. Часто файл содержит несколько фиксированных элементов среди различного рода таблиц, включающих в себя от нуля до сотен объектов. Таблица цветов в файлах точечной графики — превосходный тому пример. В зависимости от типа BMP-файла

его таблица цветов может содержать от двух до 256 элементов или же вовсе быть пустой.

Структуры переменной длины требуют от программиста осторожности. Поскольку, например, размерности массивов в Си объявляются в тексте программы, а не во время выполнения, объекты переменной длины обычно определяются как динамические структуры, размещаемые с помощью GlobalAlloc(). Как правило, не представляется возможным определить структуру переменной длины как глобальную или автоматическую переменную. В любом случае такие структуры должны размещаться в динамической памяти.

Рассмотрим объявление структуры для воображаемого файла, который содержит переменное число структур, объявленных следующим образом:

```
typedef struct tagITEM {
  int x;
  int y;
} ITEM;
```

Структура файла показана с помощью другой структуры с двумя элементами: count — счетчик количества структур ITEM в файле и items — массив, содержащий множество этих структур:

```
typedef struct tagANYSTRUCT {
  int count;
  ITEM items [1];
} ANYSTRUCT;
```

Присмотритесь к ANYSTRUCT внимательнее. Далее вы встретите не одну подобную структуру. Первый элемент структуры, int count, есть элемент фиксированной длины, в нашем случае представляющий число структур ITEM в файле. Большинство файлов имеют такие счетчики. Второй элемент — массив переменной длины структур ITEM. Ввиду отсутствия в языках Си и Си++ такого типа данных, как "массив переменной длины", items объявляется, как содержащий *один-единственный* объект типа ITEM. Это намек на массив переменной длины. Бессмысленно иметь массив из одного элемента, и такое объявление — не более чем формальность. Фактически массив будет содержать ноль, один или более объектов типа ITEM, осложняя таким образом работу по определению правильной длины объекта типа ANYSTRUCT.

Воспользуемся либеральностью языка Си по отношению к ошибкам типа нарушения диапазона индексов массива. Иными словами, даже если массив объявлен как ITEM items [1], Си не будет считать ошибочным выражение items[5]. Вопрос: как определить структуру с массивом переменной длины? Обычно решение простое. Сначала прочитайте из файла поле, содержащее число структур в массиве. Затем используйте это значение для выделения пространства динамической структуре соответствующего размера. Напри-

мер, предположим, что n представляет число структур типа ITEM в файле. Следующий оператор выделяет память подходящего размера для структуры типа ANYSTRUCT:

```
pStruct = (ANYSTRUCT *)malloc(
  sizeof(ANYSTRUCT) + ((n -1) * sizeof(ITEM)));
```

Я использовал стандартную функцию Си malloc(), но та же формула справедлива и для функции GlobalAlloc() в Windows, и для new и delete в Си++. Указатель pStruct определяется как имеющий тип ANYSTRUCT * (он может быть также ANYSTRUCT FAR *). Инструкция приведения типов в круглых скобках, предшествующая malloc(), необходима в Си++, который не допускает присвоения нетипизированных указателей, возвращаемых malloc(), типизированным указателям, подобным pStruct. Оператор передает в malloc() число требуемых байтов. Это значение равно размеру ANYSTRUCT плюс размер содержащегося в этой структуре массива переменной длины. Так как объявление ANYSTRUCT уже содержит одну структуру типа ITEM в массиве items, то окончательный размер структуры будет равен размеру ANYSTRUCT плюс *уменьшенное на единицу* число ITEM'ов раз размеров ITEM.

После распределения памяти и присвоения ее адреса указателю pStruct программа может использовать структуру и содержащийся в ней массив ITEM'ов. Можно загрузить данные с диска в структуру с помощью следующего кода:

```
pStruct->count = n; // присвоить счетчику значение n
for (int i = 0; i < pStruct->count; ++i)
  ReadFromFile(pStruct->items[i]);
```

Подразумевается, что в программе имеется функция ReadFromFile(), которая читает одну структуру типа ITEM с диска. Для дальнейшей обработки массива переменной длины items программа может использовать следующий цикл:

```
int k = pStruct->count;
while (- -k >= 0)
  DoSomething(pStruct->items[k]);
```

Несмотря на то, что массив items структуры ANYSTRUCT определен как имеющий только один элемент типа ITEM, программа может иметь доступ к более чем одному объявленному элементу, распределив достаточное количество памяти для хранения элементов. Однако жизненно важно ограничивать значение индекса корректным диапазоном. В нашем случае индексы массива items могут меняться от 0 до pStruct->count-1. Выход из этого диапазона является серьезной ошибкой, которая может привести к наруше-

нию общей защиты, остановке вашей программы и, возможно, к неустойчивой работе Windows или DOS. Неосторожное использование структур переменной длины — нешуточное дело!

Предупреждение. Никогда не полагайте, что элементы переменной длины всегда присутствуют в файле. В некоторых файлах Windows размер массива переменной длины нулевой. В этом случае этот массив не существует. *Такие массивы не должны размещаться в памяти и не должны использоваться.*

Адресация больших структур

Большие структуры порождают другие общие проблемы обработки файлов в Windows. Как правило, компиляторы Си и Си++ разработаны так, чтобы генерировать код для адресации объектов в сегментированной памяти, принятой в архитектуре процессора i86. Здесь адрес представляет собой композицию сегмента и смещения, колеблющуюся от минимум 16 байтов до 64K (точно 65536). Если структура попадает в эти пределы, программа обращается к ней, используя массивы, указатели и другие общие приемы Си и Си++. Структуры, занимающие более одного 64K-сегмента, требуют специальных методов обработки.

Замечание. Более новые версии Windows, Windows NT и 32-разрядная версия Windows, которые разрабатывались в то время, когда я писал эту книгу, обещают упростить работу с большими структурами. В идеале должна иметься возможность рассматривать динамическую память как единый гигантский ресурс, ограниченный лишь количеством установленной на компьютере памяти. В Windows 3.1 и более ранних версиях, даже несмотря на наличие многих мегабайтов ОЗУ, программы вынуждены работать с большими структурами в сегментах или еще меньших областях.

Рассмотрим программу, выделяющую некоторую область в глобальной динамической памяти с помощью функции GlobalAlloc():

```
HGLOBAL hBuffer;
long size = 100000L;
hBuffer = GlobalAlloc(GPTR, size);
if (!hBuffer) Error();
```

В этом примере программа присваивает значение 100000 переменной size, представляющей размер большой структуры, хранящейся в файле. (Это значение, вероятно, хранилось где-нибудь в самом файле.) GlobalAlloc()

вызывается для размещения структуры данного размера, и результирующий дескриптор присваивается переменной hBuffer типа HGLOBAL. В случае любой ошибки программа вызывает функцию Error(), здесь не показанную.

Если не было ошибки, hBuffer указывает на 100000-байтовую область, выделенную в динамической памяти Windows. Следующая задача — прочитать данное множество байтов из файла. К сожалению, этого нельзя сделать с помощью простого оператора. Вы должны читать файл кусками в выделенную область, используя псевдоуказатель, определенный как длинное целое. Листинг 2.1 демонстрирует приемы, которые работают в Windows 3.0 и 3.1.

Листинг 2.1. Демонстрация многосегментной структуры

```
long toAddr;
long start = 0L;
long count = size;
long lp = (long)GlobalLock(hBuffer);
if (lp) {
 while (count > 0){
   toAddr = MAKELONG(LOWORD(start), HIWORD(lp) +
     (HIWORD(start) * FP_OFF(__ahIncr)));
   if (count > 0x4000L)
     count = 0x4000L;
   _lread(hFile, (LPSTR)toAddr, (WORD)count);
   start += count;
   count = size - start;
 }
 GlobalUnlock(hBuffer);
}
```

Необходимы следующие переменные:

- toAddr — адрес, по которому данные будут запоминаться в памяти;
- start — новый начальный адрес в большом буфере;
- count — общее число байтов, которое нужно прочитать с диска;
- lp — указатель (селектор и смещение) на большой буфер.

Чтобы прочитать файл в большой буфер, идентифицируемый дескриптором hBuffer, фрагмент кода из листинга 2.1 вызывает функцию GlobalLock() и передает ей hBuffer как аргумент. Результатом этой функции является приведенное к типу long значение, которое присваивается lp.

Далее, цикл while повторяется до тех пор, пока count больше нуля (т.е. пока все байты не будут введены с диска). Для формирования следующего адреса, по которому должны загружаться данные, программа выполняет следующий оператор:

```
toAddr = MAKELONG(LOWORD(start), HIWORD(lp) +
 (HIWORD(start) * FP_OFF(__ahIncr)));
```

Этот оператор использует четыре макроса, чтобы создать соответствующий 32-разрядный адрес точки буфера, определяемой переменной start. Это значение адресного смещения модифицируется "магической" функцией Windows __ahIncr(), которая объявляется в программе следующим образом:

```
// Требуется для арифметики сегментов
extern "C" {
void FAR PASCAL __ahIncr();
}
```

Не ищите эту функцию в windows.h или руководствах по Windows — она не документирована. Функция __ahIncr() возвращает значение, показывающее, на сколько должен быть увеличен сегмент, определяемый селектором, для формирования следующего адреса в области памяти, превышающей 64K.

После формирования указателя в переменной toAddr программа помещает в count размер куска файла, подлежащего загрузке на данном проходе цикла while. В нашем примере размер куска ограничен 0x4000L (или 16384) байтами. Приемлем любой другой размер, меньший одного сегмента.

Теперь программа может вызвать файловую функцию _lread() для загрузки count байтов с диска по адресу toAddr. В листинге 2.1 имеется в виду, что файл hFile открыт. Можно также заменить этот оператор вызовом функции записи на диск или на любой другой код, позволяющий обрабатывать большие буфера по кускам. (См. информацию об этих и других функциях в следующей главе.)

Наконец, переменная start увеличивается на count байтов, прочитанных с диска, после чего count уменьшается до числа оставшихся байтов. Далее, когда цикл while завершается, буфер разблокируется с помощью вызова GlobalUnlock(), соответствующего ранее вызванной GlobalLock(). После этого файл может быть закрыт, и большая структура будет готова для работы.

Глава 3
ПРИЕМЫ РАБОТЫ С ФАЙЛАМИ

Windows предлагает полный комплект функций обработки файлов, которые можно использовать для чтения и записи файлов любого вида из рассматриваемых в этой книге. В данной главе приводится обзор этих функций и приемов программирования для обработки файловых структур всех типов.

Основы обработки файлов

Даже если вы набили руку на программах обработки файлов, прочитайте эту главу о встроенных в ядро Windows файловых функциях. Несмотря на то, что программы под Windows могут вызывать DOS-подпрограммы для работы с файлами, лучше всего, вероятно, использовать набор файловых функций Windows. Это помогает гарантировать совместимость с будущими версиями Windows, а также исключает необходимость связывать программы со стандартными функциями из библиотеки компилятора Си, дублирующими те же возможности, что уже имеются в самой Windows. Можно со всей определенностью сказать, что нет причин не использовать функции Windows.

Открытие файлов

Как и в большинстве операционных систем, Windows-программы должны открыть файл, перед тем как его использовать. Операция открытия файла подготавливает внутренние буфера и совершает некоторые прочие действия, к которым программа не имеет отношения. Более важно то, что открытие файла создает дескриптор, который может быть передан другой функции, читающей или записывающей данные. Дескрипторы файлов Windows имеют тип HFILE.

Чтобы открыть файл, используйте функцию _lopen(), объявленную в windows.h следующим образом:

```
HFILE _lopen(LPCSTR lpszFilename, int fnOpenMode);
```

- lpszFilename — указатель на строку с нулем на конце, содержащую маршрут файла;
- fnOpenMode — один из флажков табл. 3.1, произвольно скомбинированный с одним из флажков табл. 3.2. Эти флажки выбирают различные опции открытия, поясняемые таблицами. Флажки из табл. 3.1 определяют, собирается ли программа читать и/или записывать данные в файл. Флажки из табл. 3.2 выбирают режимы разделения файла, которые влияют на работу других программ с этим файлом.

Замечание. Имя _lopen(), а также имена других файловых функций Windows с предшествующим одиночным символом подчеркивания, по-видимому, свидетельствуют об их "системном" характере. Несмотря на особый стиль наименования, в использовании они ничем не отличаются от других функций Windows.

Табл. 3.1.

Стандартные опции _lopen()

Имя	Значение	Назначение
READ	0x0000	Открыть файл для чтения
WRITE	0x0001	Открыть файл для записи
READ_WRITE	0x0002	Открыть файл для чтения и записи

Табл. 3.2.

Опции _lopen() для разделения файла

Имя	Значение	Назначение
OF_SHARE_COMPAT	0x0000	Разрешает одновременное открытие уже открытого файла неограниченное число раз любым числом других задач. Если другая программа уже открыла данный файл в другом режиме, отличном от OF_SHARE_COMPAT, _lopen() возвращает HFILE_ERROR. (Поскольку значение этой опции нулевое, она является режимом по умолчанию)

Табл. 3.2. (продолжение)

Имя	Значение	Назначение
OF_SHARE_EXCLUSIVE	0x0010	Открывает файл только для одной программы. Если файл уже открыт этой или иной программой, _lopen() возвращает HFILE_ERROR.
OF_SHARE_DENY_WRITE	0x0020	Открывает файл и отвергает все другие программы, имеющие право записи в него. Если другая программа уже открыла этот файл в режиме записи, _lopen() возвращает HFILE_ERROR.
OF_SHARE_DENY_READ	0x0030	Открывает файл и отвергает все другие программы, имеющие право его чтения. Если другая программа уже открыла этот файл в режиме чтения, _lopen() возвращает HFILE_ERROR.
OF_SHARE_DENY_NONE	0x0040	Открывает файл, не отвергая все другие программы, имеющие право доступа к нему. Если другая программа уже открыла этот файл в режиме OF_SHARE_COMPAT, _lopen() возвращает HFILE_ERROR.

Предупреждение. Опции разделения файла в табл. 3.2 требуют резидентной утилиты MS-DOS SHARE.EXE.

В случае успеха _lopen() возвращает дескриптор типа HFILE. Сохраните его для последующего чтения/записи данных и закрытия файла. В случае любой ошибки _lopen() возвращает HFILE_ERROR. Эта ошибка говорит о том, что либо специфицированный файл не существует, либо имеют место конфликты разделения, описанные в табл. 3.2.

Эта функция проста в употреблении. Например, для открытия большинства файлов достаточно следующих операторов:

```
HFILE hp
hf = _lopen("ANYFILE.EXT", READ);
if (hf == HFILE_ERROR) Error();
```

Если ошибки не было, hf содержит правильный дескриптор открытого файла. Чтобы открыть файл для исключительного использования какой-нибудь программой, применяйте логическую ИЛИ-комбинацию флажков из таблиц 3.1 и 3.2.

```
HFILE hp
hf = _lopen("ANYFILE.EXT", READ_WRITE | OF_SHARE_EXCLUSIVE);
if (hf == HFILE_ERROR) Error();
```

Закрытие файла

Всегда закрывайте все открытые файлы после завершения работы с ними. Закрытие файла освобождает внутренние буфера и другие переменные для использования файлами, которые будут открыты позднее. Кроме того, при этом из памяти выгружаются данные, еще не записанные на диск. Чтобы закрыть файл, вызовите функцию _lclose(), объявляемую следующим образом:

```
HFILE _lclose(HFILE hf);
```

- hf — дескриптор файла, ранее возвращенный функциями _lopen(), _lcreat() или OpenFile().

Для _lclose() крайне не характерны ошибочные ситуации, но если что-то будет не так, функция возвратит HFILE_ERROR. Поскольку закрытие файла может быть неудачным из-за невозможности выгрузки буферов на диск, имеет смысл закрывать файлы с помощью следующего цикла:

```
while (_lclose(hf) == HFILE_ERROR)
Error();
```

Этот цикл неоднократно вызывает функцию Error() до тех пор, пока файл не закроется должным образом. Неприятность состоит в том, что цикл может заклинить по необъяснимой причине, когда файл вообще не может закрыться ни при каких обстоятельствах. Это серьезная проблема, но не такая, чтобы привести к зависанию программы. В показанном здесь цикле вы можете отследить число попыток закрытия файла. После десяти раз программа может отказаться от дальнейших попыток и выбраться из ситуации иным способом.

Чтение файла

После открытия файла для чтения (или для чтения/записи) вызовите _lread(), чтобы прочитать один или более байтов. Эта функция объявляется следующим образом:

```
UINT _lread(HFILE hf, void _huge* hpvBuffer, long cbBuffer);
```

- hf — дескриптор открытого файла.
- hpvBuffer — адрес памяти, по которому функция должна передать байты с диска.
- cbBuffer — число байтов, подлежащих считыванию. Это значение должно быть меньше или равно 0xFFFE (десятичное 65534) — на два байта меньше полного сегмента размером 64K.

Замечание. В некоторых справочниках неправильно указывается, что cbBuffer должен быть равным размеру буфера, адресуемого hpvBuffer. На самом деле адресуемый буфер может быть больше, чем число байтов, заданных в cbBuffer. Разумеется, он никогда не должен быть меньше этого значения.

По мере того как программа читает байты из файла, Windows продвигает внутренний указатель, отмечающий место, с которого будет продолжено чтение при последующих вызовах _lread(). Непосредственно после открытия файл позиционируется на первый байт. (См. функцию _llseek() в этой главе, где объясняется, как передвигать текущую позицию файла и управлять таким образом порядком чтения.)

Обычно программа читает структуру из файла с помощью нескольких простых операторов. Сначала откройте файл, вызвав функцию _lopen():

```
HFILE hf;
hf = _lopen("ANYFILE.EXT", READ);
if (hf == HFILE_ERROR) Error();
```

Если не было ошибки, создайте динамическую структуру с помощью рассматриваемых в предыдущей главе методов. Эта структура будет хранить прочитанные из файла данные.

```
LPANYSTRUCT pStruct;
pStruct = (LPANYSTRUCT)malloc(sizeof(ANYSTRUCT));
if (!pStruct) Error();
```

Удостоверьтесь, что malloc() предшествует использованию указателя pStruct. Чтобы получить этот указатель, можно вызвать GlobalAlloc(), GlobalLock() и GlobalUnlock() (см. гл. 2). Далее, прочитайте структуру из файла с помощью _lread():

```
UINT result;
result = _lread(hf, (LPSTR)pStruct, sizeof(ANYSTRUCT));
if (result != sizeof(ANYSTRUCT)) Error();
```

Функция _lread() возвращает число байтов, фактически прочитанных из файла. Если это значение не равно требуемому, возможно, произойдет ошибка. (При серьезных ошибках типа сбоя диска _lread() возвращает HFILE_ERROR.)

Часто необходимо употреблять выражения приведения типов, например (LPSTR), показанное здесь, хотя в данном случае pStruct уже является совместимым по типу указателем и приведение типов необязательно. Несмотря на то, что второй параметр _lread() объявлен с модификатором _huge*, допускается передача в функцию любого дальнего (32-разрядного) указателя.

Замечание. Модификатор _huge* создает *большой указатель*, который эквивалентен дальнему указателю, но может увеличиваться и использоваться в арифметических выражениях. Дальний указатель не может употребляться таким способом, потому что его сегментное значение не нормализовано, т.е. не находится в пределах 0x0000 — 0x000F. Два или более указателей могут ссылаться на один и тот же адрес, и поэтому ненормализованные указатели нельзя безопасно сравнивать. Поскольку нормализованные указатели уникально адресуют память, не существует двух различных больших указателей, ссылающихся на один и тот же адрес. К сожалению, нормализация, пусть даже и автоматическая, требует времени, поэтому большие указатели не следует использовать без особой нужды.

Совет. См. функцию Windows 3.1 _hread() в этой главе для информации по чтению структур, превышающих 64K.

Создание нового файла

Наряду с открытием уже существующих файлов, вам, вероятно, потребуется время от времени создавать новые. Это делается с помощью функции _lcreat(), объявляемой следующим образом:

```
HFILE _lcreat(LPCSTR, lpszFilename, int fnAttribute);
```

- lpszFilename — указатель на строку с нулем на конце, содержащую маршрут создаваемого файла.
- fnAttribute — атрибут файла: 0 — нормальный (нет ограничений по чтению или записи; 1 — только читаемый (нельзя записывать); 2 — скрытый; 3 — системный.

Замечание. Параметр fnAttribute описывает атрибуты файла MS-DOS, а не режимы разделения, перечисленные для _lopen(). Например, при задании атрибута "только читаемый" (1) файл помечается в каталоге как только читаемый. Это не запрещает программе, создавшей такой файл, записывать в него, что было бы, разумеется, нелепо.

Функция _lcreat() попытается создать специфицированный файл, а в случае успеха откроет его и возвратит дескриптор. Иначе функция возвращает HFILE_ERROR. Обычно программа создает файл следующим образом:

```
HFILE hf;
hf = _lcreat("NEWFILE.EXT", 0);
if (hf == HFILE_ERROR) Error();
```

Вместо литеральной строки можно передать буфер типа char, содержащий имя файла:

```
char fname[144] = "NEWFILE.EXT";
hf = _lcreat(fname, 0);
```

Совет. В Windows максимальное число символов, которое может содержаться в маршруте, включая букву дисковода и завершающий символьную строку нулевой байт, равно 144. Всегда задавайте в Windows значение 144 для длины переменной, содержащей маршрут.

После успешного создания нового файла последний открыт для записи, как если бы вы вызвали _lopen() в режиме записи. Закрывайте файл обычным образом с помощью _lclose().

Совет. Программа не может выбрать режим разделения файла при его создании. Для задания режима разделения из табл. 3.2 создайте файл, закройте его и затем сразу же откройте с помощью _lopen().

Запись в файл

Как легко предположить, функция _lread() имеет близнеца — функцию _lwrite(), записывающую данные в файл. Эта функция объявляется следующим образом:

```
UINT _lwrite(HFILE hf, const void _huge* hpvBuffer, UINT cbBuffer);
```

Параметры идентичны параметрам функции _lread(), но указатель hpvBuffer объявлен константой. Это говорит о том, что функция не изменяет область, адресуемую данным указателем. Как и в функции _lread(), параметр cbBuffer должен быть меньше или равен 0xFFFE (десятичное 65534).

Передайте функции _lwrite() дескриптор файла, открытого для записи (или для чтения и записи). Например, чтобы записать структуру в файл, идентифицированный дескриптором hf, можно использовать следующие операторы:

```
UINT result;
result = _lwrite(hf, (const LPSTR)pStruct, sizeof(ANYSTRUCT));
if (result != sizeof(ANYSTRUCT)) Error();
```

Эта последовательность весьма схожа с используемой при чтении файла, за исключением того, что указатель структуры объявлен константой.

Совет. См. функцию Windows 3.1 _hwrite() в этой главе для информации по записи структур, превышающих 64K.

Открытие файла с помощью OpenFile()

Чтобы открыть файл, можно также использовать другую функцию Windows — OpenFile(). Эта функция возвращает дескриптор, как и _lopen(), но имеет много других опций, которые иногда могут быть полезны. (Если эти возможности вам не нужны, применяйте более простую функцию _lopen). Функции OpenFile() передается связанная с ней структура OFSTRUCT для определения различной информации о файле. В листинге 3.1 показана эта структуры и три ее указателя.

Листинг 3.1. OFSTRUCT (windows.h)

```
typedef struct tagOFSTRUCT {
  BYTE cBytes;
  BYTE fFixedDisk;
  UINT nErrCode;
  BYTE reserved[4];
  BYTE szPathName[128];
} OFSTRUCT;
typedef OFSTRUCT* POFSTRUCT;
typedef OFSTRUCT NEAR* NPOFSTRUCT;
typedef OFSTRUCT FAR* LPOFSTRUCT;
```

- cBytes — число байтов в структуре OFSTRUCT.
- fFixedDisk — не равен нулю, если файл на жестком диске; равен нулю в противном случае.
- nErrCode — код ошибки MS-DOS (допустим, если только OpenFile() возвращает HFILE_ERROR).
- reserved[4] — не документировано и не используется.
- szPathName[128] — имя файлового маршрута в OEM (читай MS-DOS) символьном коде.

Передайте объект типа OFSTRUCT функции OpenFile() по адресу, чтобы открыть, создать, удалить или повторно открыть файлы. Можно также использовать эту функцию для проверки некоторой информации о файле. Например, находится ли файл на жестком или на гибком диске. OpenFile() объявляется следующим образом:

```
HFILE OpenFile(LPCSTR lpszFileName,
  OFSTRUCT FAR* lpOpenBuf, UNIT fuMode);
```

- lpszFileName — указатель строки с именем файла. Маски типа "*.*" недопустимы.
- lpOpenBuf — дальний указатель объекта типа OFSTRUCT. В зависимости от выбранных опций, элементы структуры могут требовать или не требовать инициализации.
- fuMode — Логическая комбинация опций из табл. 3.3 с помощью связки "ИЛИ", некоторые из которых дублируют флажки из табл. 3.2.

Табл. 3.3.

Опции функции OpenFile()

Имя	Значение	Назначение
OF_SHARE_COMPAT	0x0000	См. табл. 3.2
OF_READ	0x0000	Открывает файл для чтения
OF_WRITE	0x0001	Открывает файл для записи
OF_READWRITE	0x0002	Открывает файл для чтения/записи
OF_SHARE_EXCLUSIVE	0x0010	См. табл. 3.2
OF_SHARE_DENY_WRITE	0x0020	См. табл. 3.2
OF_SHARE_DENY_READ	0x0030	См. табл. 3.2
OF_SHARE_DENY_NONE	0x0040	См. табл. 3.2
OF_PARSE	0x0100	Присваивает значения элементам объекта типа OFSTRUCT, но не выполняет никаких файловых операций.
OF_DELETE	0x0200	Удаляет файл, если тот будет найден.
OF_SEARCH	0x0400	Ищет указанный файл в каталогах, перечисленных в команде PATH. Осуществляет поиск, даже если имя файла содержит маршрут и идентификатор дисковода.
OF_VERIFY	0x0400	Согласно официальной документации эта опция предписывает функции OpenFile() возвращать HFILE_ERROR, если дата и время указанного файла не соответствуют дате и времени в структуре типа OFSTRUCT. Однако т.к. последняя не содержит этой информации, OF_VERIFY, видимо, не имеет практического значения.

Табл. 3.3. (продолжение)

Имя	Значение	Назначение
OF_CANCEL	0x0800	Употребляется вместе с OF_PROMPT для добавления кнопки Cancel в диалоговую панель. Если пользователь "нажимает" Cancel, OpenFile() возвращает HFILE_ERROR, эмулируя ошибку "файл не найден".
OF_CREATE	0x1000	Создает новый файл. Игнорирует все флажки разделения. Для задания режима разделения создайте и закройте файл, а затем сразу же откройте его с нужными опциями разделения.
OF_PROMPT	0x2000	Если файл с указанным именем не найден, данная опция предписывает функции OpenFile() отобразить на экране диалоговое окно, предлагающее пользователю вставить содержащий файл диск в дисковод A:. Используйте эту опцию вместе с OF_CANCEL для добавления в упомянутое диалоговое окно кнопки Cancel.
OF_EXIST	0x4000	Проверяет существование указанного файла. Если тот не существует, OpenFile() возвращает HFILE_ERROR. В противном случае функция заполняет поля структуры OFSTRUCT.
OF_REOPEN	0x8000	Не слишком хорошо документирована. Предположительно повторно открывает файл, используя, очевидно, данные из структуры OFSTRUCT. Используйте данную опцию с осторожностью.

Замечание. Табл. 3.3 содержит три синонима для идентификаторов из табл. 3.1. Это пары READ и OF_READ, WRITE и OF_WRITE, READ_WRITE и OF_READWRITE. Для функции _lopen() подходят и те, и другие.

Функция OpenFile() фактически является коллекцией подпрограмм, выбираемых в зависимости от опций из табл. 3.3. Далее показано, как использовать данную функцию для открытия файла.

```
HFILE hf;
OFSTRUCT ofs;
hf = OpenFile ("ANYFILE.EXT", (OFSTRUCT FAR*)&ofs, OF_READ);
if (hf == HFILE_ERROR) Error();
```

Эти строки похожи на те, которые использовались при открытии файла с помощью _lopen(). Отличие в том, что здесь присутствует объект ofs типа

OFSTRUCT, передаваемый по адресу в функцию, которая заполняет соответствующие элементы структуры. Приведем более сложный пример:

```
HFILE hf;
OFSTRUCT ofs;
hf = OpenFile ("ANYFILE.EXT", (OFSTRUCT FAR*)&ofs,
  OF_READWRITE |
  OF_SHARE_EXCLUSIVE |
  OF_SEARCH |
  OF_PROMPT |
  OF_CANCEL);
if (hf == HFILE_ERROR) Error();
```

Эти операторы открывают файл в режиме чтения/записи для исключительного использования одной программой. Различные флажки из табл. 3.3 комбинируются с помощью логической связки ИЛИ (оператор |). Если указанный файл не найден, Windows осуществляет его поиск во всех каталогах, перечисленных в команде PATH. Если и после этого файл не найден, Windows отображает на экране диалоговую панель с кнопкой Cancel, предлагая пользователю вставить диск с файлом в дисковод A:.

Когда указана опция OF_SEARCH, функция OpenFile() ищет файл во многих местах, помимо заданных в PATH. Порядок просмотра каталогов следующий:

1. Текущий рабочий каталог, не обязательно являющийся тем же самым, откуда была запущена программа.

2. Исходный (home) каталог Windows, содержащий WIN.COM. Этот каталог возвращается функцией GetWindowsDirectory().

3. Системный каталог Windows. Если C:\WINDOWS есть исходный каталог, то C:\WINDOWS\SYSTEM — системный каталог. Последний также возвращается функцией GetSystemDirectory().

4. Исходный каталог программы, т.е. каталог, содержащий EXE-файл. Этот каталог возвращается функцией GetModuleFileName().

5. Все каталоги, перечисленные в команде PATH, выдаваемой из командной строки MS-DOS (или задаваемой в командном (batch) файле) до начала работы с Windows. После старта Windows невозможно изменить список маршрутов из окна DOS.

6. Сетевые маршруты.

Совет. Функция OpenFile() широко применяется такими файловыми утилитами, как менеджер файлов Windows. Чтобы запретить использование OpenFile(), добавьте в программу оператор #define NOOPENFILE перед включением заголовка windows.h или определите этот символ в опциях компилятора. Тогда заголовочный файл пропустит прототип OpenFile() и объявление OFSTRUCT. Этот трюк может быть полезен для предотвращения использования OpenFile() бригадой программистов, что гарантирует лучшую совместимость операторов файловой обработки.

Другие полезные файловые функции

Рассмотрим еще несколько функций, которые могут быть полезны при написании программ файловой обработки в Windows. Первая из этих функций, называемая SetHandleCount(), объявляется следующим образом:

```
UINT SetHandleCount(UINT cHandles);
```

- cHandles — новое максимальное число дескрипторов, которые программа собирается использовать; должно быть меньше или равным 255.

Обычно Windows-программы получают максимум по 20 дескрипторов каждая до общего максимума обозначенного в команде FILES=n файла CONFIG.SYS, где n изменяется примерно от 40 до 255. Многие эксперты рекомендуют устанавливать n равным 60, как минимум. Если вам нужно открыть более 20 файлов одновременно, вызовите SetHandleCount() для увеличения количества доступных программе файлов. В случае успеха функция возвращает положительное значение, равное cHandles, или меньшее значение, показывающее число доступных программе дескрипторов. В противном случае SetHandleCount() возвращает число дескрипторов, доступных программе прежде.

Предупреждение. Нет никакой гарантии, что вы получите требуемое число дескрипторов. Самое лучшее — никогда не писать под Windows программ, запрашивающих более чем 20 дескрипторов.

Другая полезная функция возвращает идентификатор дисковода, который подходит для хранения временных файлов. Эта функция, GetTempDrive(), объявляется следующим образом:

```
BYTE GetTempDrive(char chDriveLetter);
```

- chDriveLetter — обеспечивает совместимость с ранними версиями Windows. Этот параметр игнорируется и должен быть равным нулю.

GetTempDrive() возвращает значение типа BYTE, содержащее ASCII-идентификатор рекомендуемого для временных файлов дисковода. Вызывайте данную функцию следующим образом:

```
BYTE drive;
drive = GetTempDrive(0);
```

В дальнейшем можно использовать переменную drive как букву, идентифицирующую дисковод для временных файлов. Как правило, GetTempDrive() возвращает идентификатор первого дисковода (обычно C). Если доступного дисковода нет (маловероятно, т.к. Windows не устанавливается на флоппи-диски), GetTempDrive() возвращает идентификатор текущего дисковода.

Более полезной функцией для создания временных файлов является GetTempFileName(), объявляемая следующим образом:

```
int GetTempFileName(
BYTE bDriveLetter,
LPCSTR lpszPrefixString,
UINT uUnique,
LPSTR lpszTempFileName);
```

- bDriveLetter — идентификатор дисковода, который вы хотели бы использовать для хранения временных файлов.
- lpszPrefixString — указатель на строку с нулем на конце, содержащую префикс, который используется при создании временных файлов и маршрутов.
- uUnique — уникальное целое число, используемое при создании имени временного файла. Устанавливайте его отличным от нуля для использования явного значения. Задавайте его равным нулю, чтобы GetTempFileName() создала уникальное имя временного файла, используя значение текущего времени. Как правило, нуль является наилучшим выбором, если у вас нет веских причин задать *особенное* имя для временного файла.
- lpszTempFileName — указатель на 144-байтовый или более длинный символьный буфер, который функция заполняет именем будущего временного файла в OEM-кодировке, т.е. совместимой с текстовым режимом MS-DOS.

Несмотря на несколько опубликованных отчетов о противоположном, GetTempFileName() создает просто имя файла, а не собственно файл. Для создания самого файла вызовите _lcreat() после GetTempFileName(). В большинстве случаев временный файл создается следующим образом:

```
HFILE hf;
char szFileName[144];
GetTempFileName(0, "APP", 0, szFileName);
hf = _lcreat(szFileName, 0);
if (hf == HFILE_ERROR) Error();
```

Переменная hf является дескриптором нового файла. Символьный буфер szFileName содержит имя временного файла. Это имя начинается префиксом APP или любым другим трехбуквенным префиксом. Два нулевых параметра предписывают функции сформировать полное имя временного файла с использованием текущего дисковода и системного времени. Заданные параметры гарантируют уникальность создаваемого имени в пределах выбранного каталога.

После того как имя временного файла получено, вызовите _lcreat() для создания этого файла. Наверное, будет излишне указывать опции разделения файла, которые потребовали бы закрытия и затем повторного открытия файла с помощью _lopen(). Вы можете это сделать, если захотите, но GetTempFileName() гарантирует уникальность имени файла, так что нет нужды опасаться других программ, способных запортить данный файл.

Предупреждение. Всякие файлы, временные или постоянные, не удаляются автоматически после завершения программы. Всегда удаляйте все временные файлы с помощью функции OpenFile().

Имя файла создается функцией GetTempFileName() в следующей форме:

```
drive:\path\prefixxxxx.TMP
```

Идентификатор диска drive и маршрут path берутся из переменной окружения TEMP, если та была задана до старта Windows. Для этого добавьте следующую строку в ваш AUTOEXEC.BAT *в точности, как показано, без лишних пробелов вокруг знака равенства:*

```
set TEMP=C:\TEMP
```

Это подразумевает наличие предварительно созданного пустого подкаталога TEMP в корневом каталоге диска C:.

Одной из наиболее важных функций, необходимой почти всем программам обработки файлов, является SetErrorMode(), которая не часто рассматривается в руководствах по программированию для Windows. Эта функция объявляется следующим образом:

```
UINT SetErrorMode(UINT fuErrorMode);
```

- fuErrorMode — флажок или комбинация флажков, перечисленных в табл. 3.4.

Табл. 3.4.

Флаговые константы функции SetErrorMode()

Константа	Значение	Назначение
SEM_FAILCRITICALERRORS	0x0001	Если происходит нарушение разделения и этот флажок установлен, Windows не отображает диалоговую панель с сообщением об ошибке. Вместо этого программе предлагается самой обработать критическую ошибку. Например, анализируя возвращенное значение _lopen(). Большинство приложений должны использовать этот флажок.
SEM_NOGPFAULTERRORBOX	0x0002	Если возникает общее нарушение защиты и этот флажок установлен, Windows не отображает соответствующую диалоговую панель, как это обычно происходит в таких случаях. Пользовательские приложения никогда не должны использовать этот флажок.
SEM_NOOPENFILEERRORBOX	0x8000	Если файл не найден и этот флажок установлен, Windows не отображает соответствующую диалоговую панель. Почти все приложения должны использовать этот флажок.

Предупреждение. Приложения никогда не должны передавать константу SEM_NOGPFAULTERRORBOX в функцию SetErrorMode(). Это могут делать только отладчики и подобные им программы, которые ловят ошибки типа общего нарушения защиты.

Как правило, программа должна вызывать SetErrorMode() с первой и последней опциями из табл. 3.4. Это предотвращает появление сообщений об ошибках, которые могут смутить пользователей вашей программы. Вызывайте эту функцию так:

```
SetErrorMode(SEM_FAILCRITICALERRORS | SEM_NOOPENFILEERRORBOX);
```

Функция возвращает текущую установку режима обработки ошибок. Вы можете сохранить это значение в переменной типа UINT, однако вряд ли вам это понадобится.

Поиск в файле

Редкие программы открывают файл и просто читают его сверху донизу. Как правило, необходимо прыгать по файлу, читая и записывая поля в разных местах. Например, вам может потребоваться особый байт для проверки природы файла или нужно будет предварительно прочитать счетчик для создания массива переменной длины.

Операции подобного рода возможны с помощью функции _llseek(), которая позиционирует внутренний указатель файла на любой байт. После вызова _llseek() можно использовать _lread() или_lwrite() для чтения или записи данных с новой позиции. Функция _llseek() объявляется следующим образом:

```
LONG _llseek(HFILE hf, LONG lOffset, int nOrigin);
```

- hf — дескриптор открытого файла.
- lOffset — число байтов, на которое должен быть продвинут внутренний указатель. Точный смысл этого значения зависит от идущего следом параметра.
- nOrigin — указывает направление продвижения внутреннего указателя на lOffset байтов. Устанавливайте nOrigin равным одному из перечисленных в табл. 3.5 значений.

Табл. 3.5.

Значения nOrigin функции _llseek()

Имя	Значение	Назначение
SEEK_SET	0	Продвинуть внутренний указатель вперед на lOffset байтов относительно начала файла.
SEEK_CUR	1	Продвинуть внутренний указатель вперед на lOffset байтов относительно текущей позиции.
SEEK_END	2	Продвинуть внутренний указатель назад на lOffset байтов относительно конца файла.

В случае успеха _llseek() возвращает новую текущую позицию, выраженную числом байтов от начала файла. В противном случае _llseek() возвращает HFILE_ERROR. Несколько простых операторов демонстрируют применение этой функции. Обычная цель — передвинуть внутренний указатель к определенному байту (имеется в виду, что hf является дескриптором уже открытого файла).

```
if (_llseek(hf, 0x1A, SEEK_SET) == HFILE_ERROR)
Error();
```

С помощью этого оператора вы подготавливаете чтение одного или более байтов с позиции 0x1A, быть может, используя одну из структурных диаграмм файлов в нашей книге для определения расположения в файле нужного объекта. Функция _llseek() почти всегда вызывается в операторе if, который проверяет, не вернула ли функция ошибочное значение. С другой стороны, можно сохранить возвращаемое значение в переменной:

```
LONG fpos;
fpos = _llseek(hf, 0x1A, SEEK_SET);
if (fpos == HFILE_ERROR) Error();
```

В дальнейшем вы должны будете использовать fpos для возврата в ту же самую позицию, быть может, для записи модифицированного объекта обратно на диск.

Применяйте _llseek() для подготовки к дописыванию новой информации в конец файла. Для этого позиционируйте файл на конец, установив длинный литеральный нуль (0L) для lOffset и SEEK_END для nOrigin:

```
fpos = _llseek(hf, 0L, SEEK_END);
```

Если fpos не равно HFILE_ERROR, можно вызывать _lwrite() для добавления новой информации в конец файла.

Поиск от текущей позиции часто полезен при пропуске записей. Предположим, что вы работаете с файлом, который содержит дополнительную структуру ANYSTRUCT, начиная с байта номер 0x0020. Для позиционирования внутреннего указателя файла на первый байт структуры используйте следующий код:

```
if (_llseek(hf,0x0020, SEEK_SET) == HFILE_ERROR)
  Error();
if (_llseek(hf,sizeof(ANYSTRUCT), SEEK_CUR) == HFILE_ERROR)
  Error();
```

Затем с помощью _lread() или _lwrite() можно прочитать или записать данные, хранящиеся в файле сразу после ANYSTRUCT.

Другое распространенное применение для _llseek() — определение того, является ли данный файл файлом некоторого определенного вида, посредством чтения с известной позиции *байта сигнатуры* или иного числа. Листинг 3.2 демонстрирует типичный случай, когда функция IsBitmapFile() возвращает истинное значение, если идентифицируемый переменной lpszFileName файл содержит сигнатуру растрового изображения — число 0x28L в позиции 14. В случае успеха функция возвращает дескриптор открытого BMP-файла, иначе будет возвращен HFILE_ERROR.

Листинг 3.2. Функция IsBitmapFile()

```
HFILE IsBitmapFile(LPSTR lpszFileName)
{
  long fileID; /* 0x28 == файл растровой графики */
  HFILE hFile; /* дескриптор файла */
  hFile = _lopen(lpszFileName, OF_READ);
  if (hFile != HFILE_ERROR) {
    _llseek(hFile, 14, 0);
    _lread(hFile, (LPSTR)&fileID, sizeof(fileID));
    if (fileID != 0x28) {
      _lclose(hFile);
      return HFILE_ERROR;
    }
  }
  return hFile;
}
```

Эта функция открывает файл для чтения с помощью _lopen(). Если открытие прошло успешно, внутренний указатель файла позиционируется на 14-й байт, хранящий уникальную сигнатуру. (В нашем случае сигнатура — размер объекта типа struct в BMP-файле. См. гл. 4 для дополнительных методов распознавания файлов растровой графики.)

Я не побеспокоился о проверке результата _llseek() потому, что в случае ошибки последующий вызов _lread() также закончится неудачей и, следовательно, все ошибки будут схвачены. Заметьте, как в _lread() передается адрес длинной целой переменной fileID, определенной в функции. В выражении:

```
(LPSTR)&fileID
```

используется оператор & для формирования адреса fileID. Такое значение адреса обусловлено применением выражения приведения типа (LPSTR). Это гарантирует, что компилятор создаст дальний (32-разрядный) адрес. Будет работать и любой другой идентификатор дальнего указателя, однако LPSTR удобен и часто применяется для этой цели.

Заметьте также, что выражение sizeof(fileID) используется как число читаемых из файла байтов. Это намного предпочтительнее, чем указывать литеральные значения, например 4, даже если *известно*, что читаемое данное имеет длину именно четыре байта. Чтение данных в слишком маленькую для их хранения переменную — серьезная ошибка, которая может привести к хаосу, краху и даже к общему нарушению защиты. Старайтесь всегда использовать sizeof(X) для указания количества байтов, подлежащих чтению в переменную X.

Замечание. Посмотрите, как функция из листинга 3.2 передает дескриптор файла в качестве результата. Вызывающая программа может сохранить его и, в конечном итоге, использовать для закрытия файла. Это удовлетворяет общему правилу, следуя которому, программа обязана либо закрыть все открытые или вновь созданные ею файлы, либо передать их дескрипторы для более позднего закрытия файлов другими операторами, прежде чем программа завершится.

Большие файлы

Windows 3.1 вводит две новые функции обработки файлов для чтения/записи структур, превышающих 64K. Имеется также третья функция для копирования больших структур в памяти.

Предупреждение. Функции, описанные в этом разделе, требуют Windows 3.1 или более позднюю версию. Все программы, использующие хотя бы одну из этих функций, не должны запускаться в Windows 3.0 или более ранних версиях. Приемы работы с большими структурами, которые приемлемы для всех версий Windows, начиная с 2.0, описаны в разделе "Адресация больших структур" в гл. 2.

Для чтения структур, превышающих 64K, вызывайте _hread(), объявляемую следующим образом:

```
long _hread(HFILE hf, void _huge* hpvBuffer, long cbBuffer);
```

- hf — дескриптор открытого файла.
- hpvBuffer — адрес, по которому данные передаются из файла в память.
- cbBuffer — число байтов, подлежащий считыванию из файла. Его минимальное значение равно 1. Максимальное значение теоретически равняется наибольшему положительному числу, которое может быть выражено 4-байтовым длинным целым (long), 0x7FFFFFFF (десятичное 2147483647). Так как Windows не может использовать столько памяти, фактический максимум гораздо меньше. В любом случае число байтов практически не ограничено.

Используйте _hread() точно так же, как и _lread(). Функция возвращает фактически прочитанное число байтов. Если последнее не соответствует запрошенному числу байтов, то либо произошла ошибка, либо конец файла встретился раньше, чем все данные были прочитаны. Значение результата равное -1L указывает на ошибку диска. Как правило, _hread() вызывается после GlobalAlloc(), размещающей в динамической памяти большую структуру.

Для размещения и чтения большой структуры используйте следующие далее операторы. Сначала распределите память для структуры или буфера (подразумевается, что переменная size, задающая размер структуры, имеет тип long):

```
HGLOBAL hMem = GlobalAlloc(GPTR, size);
if (!hMem) Error ();
```

Теперь hMem адресует size-байтовый блок памяти. Далее, откройте файл и присвойте его дескриптор переменной hf типа HFILE. Затем прочитайте size байтов из файла в буфер с помощью следующих операторов:

```
void _huge * toAddr == GlobalLock(hMem);
_hread(hf, (LPSTR)toAddr, size);
GlobalUnlock(hMem);
```

Как обычно, вызывается GlobalLock(), чтобы получить указатель области памяти, идентифицируемой дескриптором hMem. В нашем случае данный указатель имеет тип void _huge *. Функция _hread() загружает size байтов из файла по этому адресу, который, как правило, приведен к типу (LPSTR), хотя и не обязательно. Затем вызывается функция GlobalUnlock(), соответствующая ранее вызванной функции GlobalLock().

Замечание. Вообще говоря, указатель toAddr, передаваемый функции _hread() в данном примере, может быть типизирован, т.е. объявлен как ANYSTRUCT _huge *. Но поскольку большинство компиляторов Си под Windows не позволяют программам использовать структуры, превышающие 64K, "типизированный указатель на большую структуру" является терминологическим противоречием. Таким образом, лучше всего объявлять указатель, передаваемый в _hread(), как показано здесь.

Чтобы записать большую структуру в файл, вызовите функцию _hwrite(), объявляемую следующим образом:

```
long _hwrite(HFILE hf, const void _huge* hpvBuffer, long cbBuffer);
```

Параметры те же, что и у функции _hread(), за исключением объявления hpvBuffer как const. Это свидетельствует о том, что _hwrite() не модифицирует область, адресуемую этим указателем.

Используйте _hwrite() так же, как и _lwrite(). Функция возвращает число фактически записанных на диск байтов. Если оно не равно заданному в cbBuffer, возможно, произошла ошибка записи на диск. Результат равный -1L также указывает на данную ошибку. Это говорит о том, что объект не может быть записан на диск, вероятно, из-за переполнения последнего.

Третья функция, hmemcpy(), которая может копировать одну большую структуру в другую, полезна для перемещения по памяти больших объектов. В отличие от большинства других функций в этой главе, hmemcpy() не имеет символа подчеркивания в начале своего имени. Функция объявляется следующим образом:

```
void hmemcpy(void _huge* hpvDest,
const void _huge* hpvSource, long cbCopy);
```

- hpvDest — адрес буфера назначения — место, *куда* будут копироваться байты.
- hpvSource — адрес источника — место *откуда* будут копироваться байты. Обратите внимание, что этот параметр объявлен константой, указывающей на то, что функция не модифицирует исходный буфер.
- cbCopy — число байтов, подлежащих копированию из исходного буфера в буфер назначения. Его значение должно быть положительным. Будьте осторожны. Никогда не задавайте cbCopy большим буфера назначения.

Функция hmemcpy() не возвращает никакого значения. Она всегда завершается успешно, за исключением случаев, когда исходный и принимающий буфера пересекаются. При этом возможна потеря информации.

Глава 4

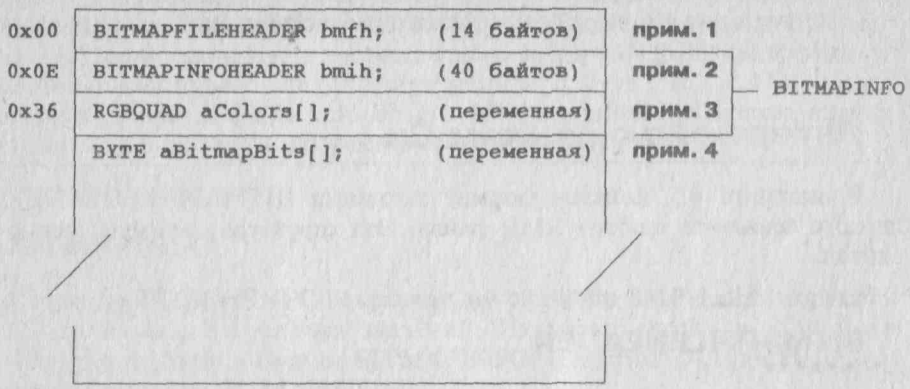
ФАЙЛЫ РАСТРОВОЙ ГРАФИКИ (.BMP)

Растровый графический файл — не просто множество точек, как следует из его названия. Это файл с весьма сложной структурой, содержащий информацию о типе, размере и цвете, а также элементы изображения — *пикселы*. BMP-файлы могут хранить не только полноцветные изображения фотографического качества, но и относительно простые картинки пиктограмм, кнопок, курсоров и других фигур.

Формат файла

На рис. 4.1 показана структура независимого от устройства графического растрового файла (DIB-формат). Все BMP-файлы должны храниться в DIB-формате, чтобы их можно было показывать на компьютерах с различной графической аппаратурой.

Рис. 4.1. DIB-формат растрового графического файла



Примечания к рис. 4.1

1. Все BMP-файлы начинаются структурой BITMAPFILEHEADER, которая идентифицирует его как растровый графический и хранит другую информацию о содержании файла.
2. Вторая структура, BITMAPINFOHEADER, содержит такие характеристики растрового изображения, как ширина и высота. Заголовок bmfh и массив aColors[] могут быть доступны как индивидуально, так и в составе структуры BIMAPINFO.
3. Массив aColors содержит структуры типа RGBQUAD интенсивностей красного, зеленого и синего цветов. Размер массива зависит от числа используемых цветов. Например, восьмицветное растровое изображение может иметь восемь RGBQUAD-элементов в массиве aColors. В растровых изображениях с 24-разрядным представлением цвета массива aColors[] нет. Поскольку этот массив переменной длины, число его элементов влияет на расположение массива aBitmapBits, описанного в следующем пункте.
4. Массив aBitmapBits содержит пикселы, которые различаются по формату в зависимости от типа растрового изображения. Кроме того, байты могут быть сжаты. Байты хранятся строками слева направо, а каждая строка представляет собой *линию развертки* (scan line) изображения, которые могут быть дополнены до 32-разрядной границы. Линии развертки упорядочены снизу вверх, т.е. *первый* элемент массива содержит пикселы *последней* строки изображения. В растровых изображениях с 24-разрядным представлением цвета, где массив aColors

отсутствует, байты в aBitmapBits непосредственно представляют 24-разрядные триадные цвета пикселов. В других растровых изображениях пикселы из aBitmapBits представляют собой индексы массива aColors.

Интерфейс с языком Си

В листинге 4.1 показан формат заголовка BITMAPFILEHEADER — первого элемента любого BMP-файла. Эта структура занимает ровно 14 байтов.

BITMAPFILEHEADER

Листинг 4.1. BITMAPFILEHEADER

```
typedef struct tagBITMAPFILEHEADER {
  UINT bfType;
  DWORD  bfSize;
  UINT bfReserved1;
  UINT bfReserved2;
  DWORD  bfOffBits;
} BITMAPFILEHEADER;
typedef BITMAPFILEHEADER* PBITMAPFILEHEADER;
typedef BITMAPFILEHEADER FAR* LBITMAPFILEHEADER;
```

- bfType — должен содержать два ASCII-символа: "B" и "M", разумеется, означающие *bitmap*. Соответствующие шестнадцатеричные значения равны 0x42 и 0x4D. При любых других значениях этого поля файл не является растровым изображением, принятым в Windows.
- bfSize — равен размеру файла в байтах, как указано в соответствующем элементе оглавления дискового каталога. Используйте bfSize для проверки целостности файла и для распределения памяти под весь файл. Значение bfSize не является размером растрового изображения.
- bfReserved1 — не документировано и не используется. Должно быть равно нулю.
- bfReserved2 — не документировано и не используется. Должно быть равно нулю.
- bfOffBits — специфицирует байтовое смещение до начала растрового изображения. Используйте это значение для определения местонахождения массива aBitmapBits в файле.

Совет. В целях безопасности для идентификации BMP-файла используйте bfType и bfSize вместе. Если bfType содержит символы "B" и "M", а bfSize не равен длине файла, то это не BMP-файл. Этот прием предотвращает ошибки, возникающие при попытке отобразить другой файл как картинку, например текстовый файл, начинающийся с букв "B" и "M". Если расширение имени файла есть BMP, а bfType и bfSize проверены, значит это, по всей видимости, корректное растровое изображение.

BITMAPINFO

Структура BITMAPINFO покрывает два элемента BMP-файла: структуру заголовка и массив цветовых значений. Их можно читать и записывать индивидуально, а не в составе BITMAPINFO. В листинге 4.2 показано, как выглядит структура BITMAPINFO.

Предупреждение. BITMAPINFO — структура переменной длины. Размер массива bmiColor зависит от типа растрового изображения. Поэтому никогда не определяйте в программах переменные типа BITMAPINFO непосредственно. При использовании этой структуры разместите bmiHeader фиксированной длины и выделите достаточное количество памяти для массива bmiColors переменной длины.

Листинг 4.2. Структура BITMAPINFO

```
typedef struct tagBITMAPINFO {
  BITMAPINFOHEADER bmiHeader;
  RGBQUAD bmiColors[1];
} BITMAPINFO;
tepedef BITMAPINFO* PBITMAPINFO;
typedef BITMAPINFO FAR* LPBITMAPINFO;
```

- bmiHeader — содержит размерность растрового изображения, формат и другую информацию. В следующем разделе эта структура рассматривается подробно.
- bmiColors[1] — содержит информацию о цвете в виде массива структур типа RGBQUAD. Формат и число элементов этого массива зависят от типа растрового изображения и других факторов, определяемых полями bmiHeader. Для растровых изображений с 24-разрядным представлением цвета bmiColors отсутствует. См. следующие два раздела для информации по использованию значений из этого массива.

BITMAPINFOHEADER

Предыдущая структура BITMAPINFO начиналась с элемента типа BITMAPINFOHEADER, детализированного в листинге 4.3. Эта структура полностью описывает растровое изображение, хранимое в BMP-файле. После того как программа проверит, действительно ли данный файл является BMP-файлом, можно использовать поля этой структуры для подготовки растрового изображения к показу на экране, распаковке и к прочим манипуляциям с ним. Структура занимает ровно 40 байтов.

Листинг 4.3. Структура BITMAPINFOHEADER

```
typedef struct tagBITMAPINFOHEADER {
  DWORD  biSize;
  LONG biWidth;
  LONG biHeight;
  WORD biPlanes;
  WORD biBitCount;
  DWORD  biCompression;
  DWORD  biSizeImage;
  LONG biXPelsPerMeter;
  LONG biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;
}BITMAPINFOHEADER;
typedef BITMAPINFOHEADER* PBITMAPINFOHEADER;
typedef BITMAPINFOHEADER FAR* LPBITMAPINFOHEADER;
```

- biSize — специфицирует собственный размер структуры в байтах. Это значение должно использоваться для определения первого байта массива bmiColors в структуре BITMAPINFO. Подразумевая, что указатель pbmi адресует структуру BITMAPINFO, описанную в предыдущем разделе, приведем оператор, который устанавливает указатель pc на массив цветов:

```
pc = ((LPSTR(pbmi) + (WORD)(pbmi->bmiHeader.biSize));
```

> **Совет.** Некоторые программисты используют biSize для проверки того, является ли данный файл BMP-файлом или нет. Если biSize равно 0x28 (десятичное 40), то это, по всей вероятности, BMP-файл. Данное поле находится на расстоянии 14 байтов от начала файла. Для проверки файла позиционируйте его внутренний указатель на четырнадцатый байт, прочитайте с этого места двойное слово и сравните его значение с 0x28.

- biWidth — содержит ширину изображения в пикселах.
- biHeight — содержит высоту изображения в пикселах.

- biPlanes — должен быть равен единице, т.к. BMP-файлы, какого бы типа они ни были, хранятся в независимом от устройства формате с одной цветовой плоскостью (color-plane).
- biBitCount — содержит число битов на пиксел. Кроме того, отличает монохромное изображение от цветного. Должно быть равно 1, 4, 8 или 24. Обычно используется в конъюнкции с biClrUsed и biClrImportant. См. табл. 4.1, описывающую значения biBitCount и то, как они влияют на содержимое массива bmiColors в структуре BITMAPINFO.

Табл. 4.1.

Значения biBitCount

biBitCount	Действие
1	Помечает изображение как монохромное. Показывает, что массив bmiColors в структуре BITMAPINFO содержит два элемента типа RGBQUAD. Каждый бит изображения, хранимый в массиве aBitmapBits, служит индексом массива bmiColors. Бит, равный 0, окрашен в соответствии с содержимым bmiColors[0], единичный бит — в соответствии с содержимым bmiColors[1]. Программа не должна полагать, что монохромные пикселы непосредственно представляют белый и черный цвета, хотя часто так оно и есть.
4	Показывает, что массив bmiColors содержит до 16 цветовых значений типа RGBQUAD. В этом самом распространенном формате каждый байт содержит два 4-битовых пиксела, а каждый пиксел есть индекс массива bmiColors, определяющего цвет. Например, байт изображения, равный 0x23, содержит два пиксела, один из которых равен 0x02, а второй — 0x03. Цвет первого пиксела равен bmiColors[0x02], а цвет другого — bmiColors[0x03].
8	Показывает, что массив bmiColors содержит до 256 цветовых значений типа RGBQUAD. Каждый байт в массиве aBitmapBits представляет собой отдельный пиксел, являющийся индексом массива. Цвет любого пиксела q равен bmiColors[q]. Этот формат наиболее удобен для обработки и быстрого отображения на экране монитора, но занимает примерно в два раза больше места на диске, чем 16-цветный BMP-файл.
24	Этот необычный формат может описывать изображение с более чем 16-ю миллионами цветовых оттенков. Иногда называемые полноцветными (true color), эти изображения обычно используются для представления фотографий. В данном формате массив bmiColors отсутствует. Вместо этого 24-разрядные значения массива aBitmapBits представляют каждый пиксел как красно-зелено-синюю триаду (тип RGBTRIPLE). BMP-файл этого типа может занимать огромное пространство на диске и в памяти.

- biCompression — показывает, хранится ли данное растровое изображение в сжатом виде, а также метод его упаковки. Равен BI_RGB (изображение не сжато), BI_RLE8 (8-разрядное групповое кодирование) или BI_RLE4 (4-разрядное групповое кодирование). См. "Сжатие растровых изображений" далее в этой главе.
- biSizeImage — содержит размер растрового изображения в байтах. Может быть нулевым, если biCompression = BI_RGB (изображение не сжато).
- biXPelsPerMeter — Указывает предпочтительное разрешение по горизонтали в пикселах на метр. Используйте это значение, чтобы выбрать подходящее изображение среди многих его вариантов с разными разрешениями. Теоретически чем ближе разрешающая способность устройства отображения соответствует этому и следующему параметрам, тем лучше будет выглядеть изображение на экране. На практике это значение используется редко.
- biYPelsPerMeter — указывает предпочтительное разрешение по вертикали в пикселах на метр.
- biClrUsed — обычно содержит число цветов, используемое в растровом изображении и определяемое массивом bmiColors типа RGBQUAD. Если biClrUsed равен нулю, как это обычно и бывает, в изображении используется максимальное количество цветов, возможное для изображения данного типа (см. biBitCount). Нулевое значение biClrUsed не указывает на то, что массив ColorArray пуст.
- biClrImportant — содержит число важных цветов изображения. Например, если это значение равно 3, первые три значения цвета в массиве bmiColors должны отображаться на экране с как можно более точным соответствием. Другие пикселы могут отображаться с измененным цветом или безболезненно пропускаться. Если biClrImportant равен нулю, все цвета считаются важными. Этот параметр не имеет смысла для растровых изображений с 24-разрядным представлением цвета.

Совет. В менеджере программ системы OS/2 BMP-файлы используют заголовок типа BITMAPCOREHEADER — сокращенная структура с первыми пятью элементами полного заголовка BITMAPINFOHEADER. Если biSize равен 0x0C (десятичное 12), изображение, по всей вероятности, принадлежит этой операционной системе, где используется похожая модель, но массив цветов образован не 8-байтовыми значениями типа RGBQUAD, а 6-байтовыми структурами типа RGBTRIPLE. Для преобразования BMP-файла из OS/2 в Windows загрузите его в графический редактор Windows Paintbrush и примените команду Save as для сохранения изображения в файле под другим именем.

RGBQUAD

Пикселы растрового изображения в общем случае являются индексами массива цветовых значений типа RGBQUAD. Цветовое значение представляет собой трехбайтовую композицию интенсивностей красного, зеленого и синего цветов. Четвертый байт дополняет структуру таким образом, что каждый элемент в RGBQUDR-массиве начинается с четного адреса. Хотя это несколько расточительно, данная схема здорово повышает производительность, т.к. процессор i86 фирмы Intel может выбирать выровненные на четную границу данные быстрее.

Замечание. Лишний четвертый байт объясняет "quad" (четверка) в названии типа структуры (RGBQUAD). Похожая структура, RGBTRIPLE, не показанная здесь, идентична RGBQUAD, но не имеет элемента rgbReserved.

Структура типа RGBQUAD показана в листинге 4.4. В BMP-файле элемент bmiColors (если таковой вообще присутствует) структуры BITMAPINFO содержит массив структур типа RGBQUAD.

```
typedef struct tagRGBQUAD {
  BYTE rgbBlue;
  BYTE rgbGreen;
  BYTE rgbRed;
  BYTE rgbReserved;
} RGBQUAD;
typedef RGBQUAD FAR* LPRGBQUAD;
```

- rgbBlue — содержит относительную интенсивность синего цвета от 0 до 255.
- rgbGreen — содержит относительную интенсивность зеленого цвета от 0 до 255.
- rgbRed — содержит относительную интенсивность красного цвета от 0 до 255.
- rgbReserved — Не используется. Предполагается равным нулю, хотя, вероятно, его фактическое значение не важно.

Замечание. Структура типа RGBQUAD может представлять 16777216 уникальных значений цвета, включая черный (все элементы нулевые) и белый (все элементы равны 255). Поскольку немногие из дисплеев персональных компьютеров могут отобразить так много цветов одновременно, комбинация значений из RGBQUAD будет воспроизводить соответствующий эквивалентный оттенок.

Сжатие растровых изображений

В Windows 3.0 и более поздних версиях BMP-файлы могут хранить сжатое изображение, используя простую, но эффективную технику *группового кодирования*, где группы из нескольких одноцветных пикселов записаны в виде числа n и цветового значения v. Таким образом, для распаковки сжатого изображения дисплейный драйвер Windows отображает n пикселов v-цвета, а затем переходит к другой кодовой группе. Группы до 255 пикселов требуют только два байта, поэтому чем длиннее группа, тем эффективнее сжатие. Так как дисплейный драйвер Windows отвечает за распаковку сжатого растрового изображения, нет необходимости писать специальные программы для этой цели. Однако имейте в виду, что не во всех этих драйверах алгоритмы распаковки реализованы корректно.

Элемент biCompressed структуры BITMAPINFOHEADER (листинг 4.3) содержит тип сжатия: BI_RGB для несжатых изображений, BI_RLE4 для 4-разрядного группового кодирования 16-цветных изображений или BI_RLE8 для 8-разрядного группового кодирования 256-цветных изображений. Монохромные и полноцветные (true color) изображения не могут быть сжаты. Следующие разделы поясняют, как работает каждый формат сжатия.

Замечание. Не часто упоминается тот факт, что *только* полноцветные изображения технически несжимаемы, т.к. его пикселы непосредственно представляют значения цвета. В некотором смысле даже BI_RGB и монохромные изображения уже сжаты, потому что их пикселы представляют собой 4- или 8-разрядные индексы в массиве bmiColors, содержащем 32-разрядные RGBQUAD-структуры.

Формат сжатия BI_RLE8

В данном формате пикселы хранятся в виде многобайтовых блоков, как минимум, по два байта каждый. Если первый байт блока равен нулю, он представляет собой код выхода, за которым сразу же следуют командные байты (см. табл. 4.2). Например, последовательность 00 01 отмечает конец данных сжатого растрового изображения.

Табл. 4.2.

Командные последовательности, используемые в формате BI_RLE8

Код выхода	Байты	Команда
00	00	Закончить текущую строку развертки и начать новую.
00	01	Конец данных. Закончить распаковку изображения.
00	02 xx yy	Дельта (или вектор). Отобразить следующий пиксел с xx-горизонтальным и yy-вертикальным смещением от текущей позиции. Цвета пропущенных пикселов не определяются.
00	nn k1, k2,...kn	Абсолютный режим. Для 3 <= nn <=255 отобразить nn несжатых пикселов с цветами, определяемыми индексами k1—kn. Поскольку все команды должны иметь четное число байтов, блок этого типа заканчивается дополнительным нулевым байтом, если nn нечетное.

Если первый байт блока не равен нулю, он представляет собой беззнаковое целое q = 1, 2, ... 255. Вслед за q идет 8-разрядный индекс цвета i. Распаковывая групповой код этого типа, Windows отображает q пикселов RGBQUAD-цвета, взятого из bmiColors[i].

Остановимся на теории сжатия с использованием группового кодирования. Несколько примеров поясняют, как работает этот метод. Рассмотрим следующий блок:

```
07 1E
```

Так как первый байт этого блока отличен от нуля, то он представляет собой число отображаемых на экране пикселов. Второй байт — индекс цвета. При распаковке этого блока прикладная программа или дисплейный драйвер отобразит семь пикселов с цветом bmiColors[0x1E] слева направо, начиная с текущей позиции линии развертки.

Ниже приводятся еще несколько примеров группового кодирования:

```
0A 01
15 24
0F 6F
```

В первой строке закодировано 10 (шестнадцатеричное 0A) пикселов цвета 0x01, во второй строке — 21 (шестнадцатеричное 0x15) пиксел цвета 0x24, в третьей — 15 (шестнадцатеричное 0xF) пикселов цвета 0x6F. Имейте в виду,

что 0x01, 0x24 и 0x0F являются индексами массива bmiColors, а не фактическими значениями красно-зелено-синей триады.

Блок с кодом выхода, т.е. в котором первый байт всегда равен нулю, требует специальной обработки. Второй байт этого блока является командой или иным значением от 0 до 255. Одной из простейших команд является команда конца текущей линии развертки:

```
00 00
```

По прочтении двойного нуля программа должна отобразить следующий пиксел с новой строки развертки. Второй байт, равный 0x01, отмечает конец данных в сжатом растровом изображении:

```
00 01
```

В этом случае программа должна прекратить распаковку изображения. Если растровое изображение распаковывалось в буфер памяти, программа должна интерпретировать этот блок как команду отображения на экране распакованного изображения.

Если второй байт блока с кодом выхода равен 0x02, это означает начало *дельта-команды*, определяющей горизонтальное и вертикальное смещения позиции следующего пиксела. Например, показанные далее четыре байта дельта-команды, определяют позицию следующего пиксела на семь точек правее и на четыре точки ниже текущей позиции:

```
00 02 07 04
```

Замечание. Дельта-команды полезны для наложения сжатых изображений переднего плана на некоторый фиксированный фон. Это один из важных приемов мультипликации. Значения цветов пропущенных пикселов не определяются. При использовании дельта-команды программа обычно сначала отображает фон, а затем изображение переднего плана, которое было упаковано с помощью дельта-команд, определяющих контур изображения. Однако дельта-команды не всегда полезны для сжатия большинства BMP-файлов.

Команда четвертого и последнего типа из блока с кодом выхода хранит группу несжатых пикселов в *абсолютном режиме*, полезного для малого числа смежных разноцветных пикселов, которые после сжатия почти не экономят пространство. В команде абсолютного режима первый байт равен нулю, а второй — некоторому значению n = 3, 4, ... 255. За этим байтом следуют ровно n индексов несжатых пикселов. При нечетном n группу дополняет лишний нулевой байт. Таким образом, следующий байт начинается с четного адреса. Приведем пример команды абсолютного режима:

```
00 04 03 0A 01 02
```

Как обычно, первый байт — нулевой код выхода. Второй байт, больший или равный трем, показывает, что данная кодовая группа не упакована (абсолютный режим), а также количество следующих далее пикселов (в данном случае четыре). Последующие четыре байта хранят неупакованные индексы пикселов. Для декодирования этого блока программа отображает на экране четыре пиксела с индексами цветов 0x03, 0x0A, 0x01 и 0x02 соответственно. Как и в сжатой кодовой группе, эти значения являются индексами массива bmiColors, а не фактическими цветами.

Если число пикселов в абсолютной последовательности нечетно, игнорируйте оконечный нулевой байт, как показано в следующих трех примерах:

```
00 03 04 02 0E 00
00 07 08 09 01 04 09 02 00
00 05 0B 01 04 03 08 00
```

Первая строка представляет три неупакованных пиксела со значениями индексов цветов, равными 0x04, 0x02 и 0x0E соответственно. Игнорируйте конечный нуль, дополняющий эту последовательность до четного числа (здесь до 6) байтов. Во второй строке записано семь абсолютных пикселов. В третьей строке представлено пять абсолютных пикселов. Каждая последовательность заканчивается дополнительным нулем, который должен быть игнорирован.

Тем не менее возможны случаи, когда последний байт, а также некоторые байты внутри абсолютной последовательности равны нулю и одновременно являются индексами цвета пикселов. Например, если n четно, все последующие байты являются значимыми.

```
00 04 05 00 03 00
```

Данная абсолютная последовательность имеет четыре пиксела с индексами цвета 0x05, 0x00, 0x03 и 0x00. В этом случае последний нуль не игнорируется.

Формат сжатия BI_RLE4

В сжатых 16-цветных изображениях используется схема группового кодирования, похожая на ту, которая применяется для 256-цветных изображений, но информация упаковывается потетрадно. Растровое изображение сжато в этом формате, если поле biCompression в структуре BITMAPINFOHEADER (листинг 4.3) равно BI_RLE4.

В данном методе сжатия одноцветные последовательности хранятся в виде двухбайтовых блоков. Первый ненулевой байт представляет количество пикселов q. В старшей тетраде второго байта хранится индекс цвета i1, а в

младшей — индекс цвета i2. Для распаковки кодовой группы этого типа программа должна отобразить q пикселов таким образом, чтобы первый пиксел имел цвет bmiColors[i1], второй — bmiColors[i2], третий — bmiColors[i1], четвертый — bmiColors[i2] и т.д., чередуя эти два индекса, пока не будут отображены все q пикселов.

Рассмотрим несколько примеров, поясняющих работу этого метода сжатия.

```
07 4A
```

Первый байт не равен нулю и, следовательно, представляет собой количество пикселов q. Второй байт хранит два цветовых индекса: 0x04 и 0x0A. При декодировании данной последовательности программа отображает семь пикселов со следующими индексами цвета:

```
4 A 4 A 4 A 4
```

Помните, что эти числа являются индексами, а не цветами. Соответствующие им фактические цвета находятся в bmiColors.

Метод сжатия BI_RLE4 может упаковывать последовательности с чередующейся парой цветов и, следовательно, теоретически он более эффективен, чем сжатие по методу BI_RLE8, где упаковываются только одноцветные группы. Однако регулярные последовательности с чередованием цвета весьма редки, и к тому же их распознавание затруднительно для программы сжатия. Более обычные, одноцветные группы упаковываются, как показано в следующем примере для восьми пикселов с цветом bmiColors[0x03]:

```
08 03
```

Первый байт, равный нулю, представляет собой код выхода (см. табл. 4.2). Конец текущей линии развертки отмечается двумя нулевыми байтами. Последовательность 00 01 завершает растровое изображение. Дельта-команда начинается последовательностью 00 02. Те же самые команды используются в формате сжатия BI_RLE8.

Однако абсолютный режим в формате BI_RLE4 другой. В этом режиме первый байт блока равен нулю. Второй байт содержит некоторое число из диапазона 3—255, представляющее собой количество следующих далее тетрад. Общее число байтов в абсолютной группе дополняется нулями до границы 16-разрядного слова. Ниже приводится пример абсолютной 16-цветной последовательности:

```
00 07 A7 1E 64 90
```

Первый байт равен нулю и, следовательно, представляет собой код выхода. Байт номер два больше или равен 3, т.е. идентифицирует абсолютную группу. В данном примере имеется семь несжатых пикселов. Каждый после-

дующий байт состоит из пары тетрад, содержащих индексы цвета в шестнадцатеричном коде. Первый пиксел имеет цвет bmiColors[0x0A], второй — bmiColors[0x07], третий — bmiColors[0x01], четвертый — bmiColors[0x0E], пятый — bmiColors[0x06], шестой — bmiColors[0x04] и седьмой — bmiColors[0x09]. Последняя нулевая тетрада в данном случае является дополнением и не значима. Однако если бы счетчик пикселов был равен 8, последняя тетрада представляла бы собой цвет bmiColors[0x00].

Возможны случаи, когда абсолютная последовательность заканчивается тремя нулевыми тетрадами. Например, группа может быть закодирована следующим образом:

```
00 05 12 34 50 00
```

Пять пикселов в данной абсолютной последовательности имеют индексы цвета 0x01, 0x02, 0x03, 0x04 и 0x05. Три конечных нулевых тетрады — только дополнение, которое должно быть игнорировано.

Совет. Независимо от метода сжатия упакованные блоки данных должны иметь четное число байтов. Преимущество этого условия в том, что сжатый BMP-файл можно читать 16-разрядными словами, декодируя их побайтно или потетрадно согласно ранее описанным командам. При этом также автоматически отбрасываются все байты или тетрады, дополняющие кодовую группу.

Глава 5

ФАЙЛЫ ПИКТОГРАММ (.ICO)

Разумеется, вам известно, что пиктограммы — маленькие растровые изображения, представляющие, как правило, файлы, программы, логотипы и другую легко символизируемую информацию. Пиктограммы используются в Windows почти повсюду. В менеджере программ они применяются для идентификации элементов групп, минимизированные (свернутые) программы показывают себя в виде пиктограмм на рабочем столе Windows, некоторые программы используют их для изображения кнопок, программы с многодокументным интерфейсом (MDI) отображают в виде пиктограмм минимизированные дочерние окна и т.д.

Обычно одна или более пиктограмм запоминаются в ICO-файле либо для длительного хранения, либо как часть множества различных ресурсов, объединяемых в EXE-файле. Эта глава поясняет, как пиктограммы хранятся в ICO-файлах.

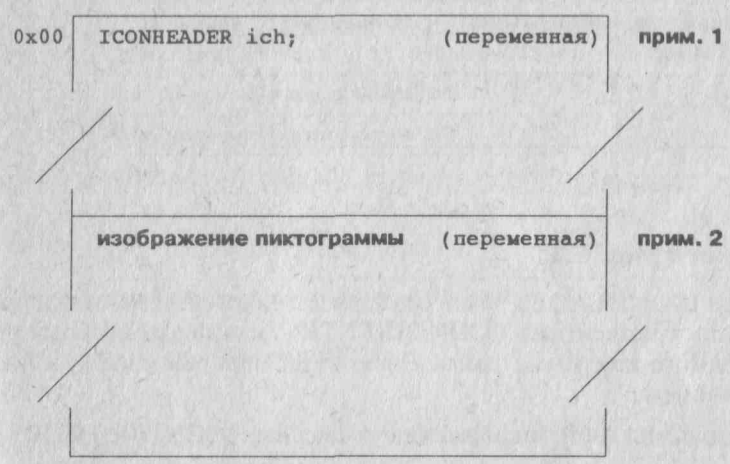
Формат файла

В отличие от некоторых прочих файлов, пиктограммы не содержат уникальных сигнатур, так что если имя файл имеет расширение ICO, то обычно нет другой альтернативы, кроме как предположить, что это корректная пиктограмма. (Я предлагаю несколько способов идентификации файлов пиктограмм, однако официально санкционированных приемов для этого нет.) Один ICO-файл может содержать одну или несколько пиктограмм.

Предупреждение. Не все файлы, имена которых имеют расширение ICO, содержат пиктограммы. Microsoft Word, например, включает файл WORD.ICO, несовместимый по формату с файлами пиктограмм в Windows.

Общий формат ICO-файла показан на рис. 5.1. Сначала идет заголовок ICONHEADER, показывающий количество и тип пиктограмм, хранящихся в файле. Вслед за заголовком идет собственно изображение, хранимое в независимом от устройства формате (DIB).

Рис. 5.1. Формат файла пиктограмм



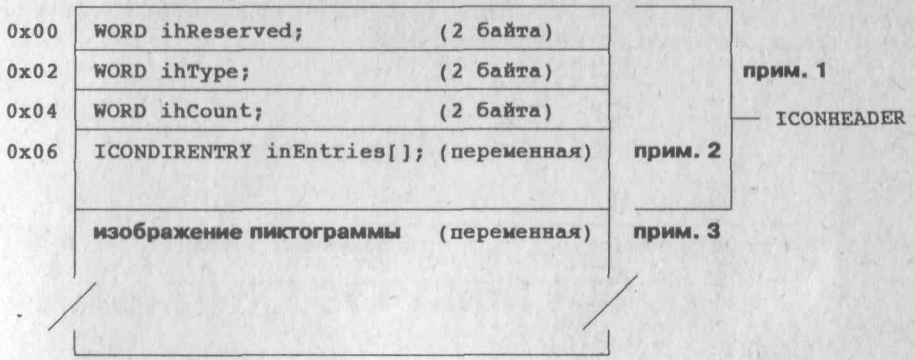
Примечания к рис. 5.1

1. Windows не объявляет структуру ICONHEADER, иногда также называемую ICONDIR. См. рис. 5.2 для дополнительной информации об этом заголовке.

2. Изображения пиктограмм хранятся в многосекционном DIB-формате, похожем на описанный в предыдущей главе, но более ограниченном. (Например, растровое изображение пиктограмм не может быть сжато.)

Заголовок ICONHEADER (рис. 5.1) содержит несколько полей, образующих каталог хранимых в файле пиктограмм. Программа может прочитать этот каталог, чтобы определить местоположение некоторой пиктограммы в файле.

Рис. 5.2. Структура ICONHEADER



Примечания к рис. 5.2.

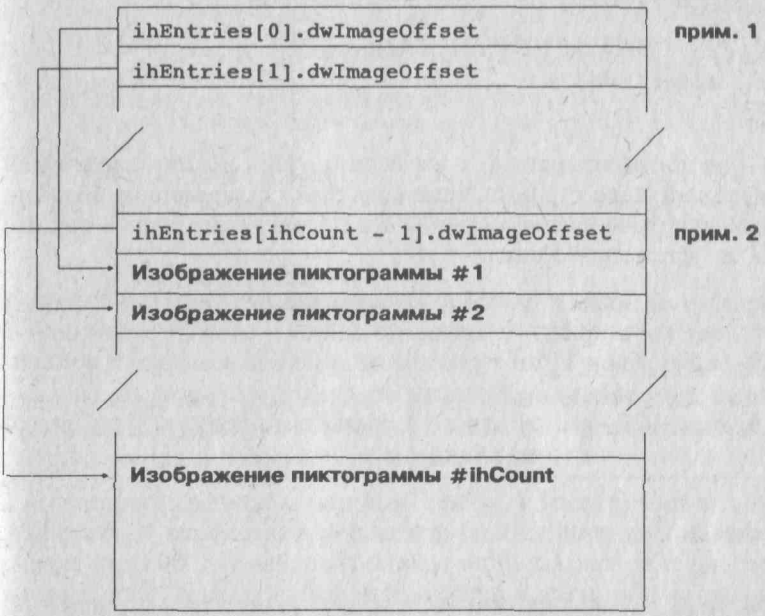
1. Структура ICONHEADER заканчивается массивом переменной длины, содержащим записи типа ICONDIRENTRY, каждая из которых описывает одну пиктограмму файла. Элемент ihCount равен числу записей в этом массиве.
2. Для дополнительной информации о массиве ICONDIRENTRY см. рис. 5.3.
3. Это то же DIB-изображение, что и на рис. 5.1.

На рис. 5.3 показано, как программа может использовать информацию из рис. 5.1 и 5.2 для определения местоположения изображения в ICO-файле. Каждая запись типа ICONDIRENTRY в структуре ICONHEADER содержит элемент dwImageOffset, равный числу байтов от начала файла до начала изображения пиктограммы. Это значение может быть использовано при создании указателя на загруженный в память ICO-файл или для поиска произвольной пиктограммы внутри файла.

Примечания к рис. 5.3

1. Хотя по изображенным стрелкам можно предположить, что dwImageOffset является указателем, это не так. Данное значение представляет собой число байтов от начала файла до изображения пиктограммы.
2. Каждое изображение пиктограммы в конце ICO-файла хранится в многосекционном DIB-формате, описанном в следующем разделе.

Рис. 5.3. Определение местоположения изображения пиктограммы в ICO-файле с помощью dwImageOffset



Интерфейс с языком Си

Поскольку Windows не определяет типы данных для файла пиктограмм, следующие далее структуры не содержатся в windows.h и весьма информативны. Вы можете использовать их как руководство по чтению/записи различных полей из файлов пиктограмм.

ICONHEADER

Файл пиктограмм начинается структурой ICONHEADER, объявленной в листинге 5.1. Размерность массива ihEntries зависит от числа пиктограмм в файле, и поэтому нельзя непосредственно определить переменную типа ICONHEADER. Самым лучшим будет сначала прочитать первые три слова заголовка, а затем выделить память для каталога ihEntries. Кроме того, можно использовать приемы обработки структур переменной длины, рассмотренные в гл. 2, для размещения динамического объекта типа ICONHEADER.

Листинг 5.1. Заголовок ICONHEADER

```
typedef struct tagICONHEADER {
  WORD ihReserved;
  WORD ihType;
  WORD ihCount;
ICONDIRENTRY  ihEntries[1];
} ICONHEADER;
```

- ihReserved — не документировано и не используется. Должно равняться нулю. По меньшей мере странно, начинать файл с зарезервированного слова, но, как уже было сказано, оно всегда должно быть нулем, так что вряд ли используется как-то еще.
- ihType — должно равняться единице. Поскольку процессор i86 фирмы Intel меняет местами старший и младший байты в слове, третий байт в каждом ICO-файле равен 1. Этот факт можно использовать для приблизительной проверки файла на корректность хранимой в нем пиктограммы. В правильном ICO-файле первые четыре байта должны быть равны 00 00 01 00 и следовать именно в таком порядке.
- ihCount — число пиктограмм в файле. Принципиально возможно, хотя и крайне необычно, хранение в ICO-файле большого числа пиктограмм. Если данное значение чрезвычайно велико (например, 10000), то можно с полным правом усомниться в целостности файла, а если ihCount равно нулю, то это определенно неправильный файл.
- ihEntries — массив переменной длины, содержащий структуры ICONDIRENTRY, каждая из которых описывает различные признаки пиктограммы. Данный массив содержит ihCount элементов, пронумерованных от 0 до ihCount-1.

Совет. Чтобы загрузить массив ihEntries в память, прочитайте слово ihCount и присвойте переменной n значение (ihCount * sizeof(ICONDIRENTRY)). Создайте n-байтовый буфер и прочитайте туда массив ihEntries, начиная с седьмого байта файла. Еще один совет: байтовое смещение от начала файла до начала первой пиктограммы равно n+6.

ICONDIRENTRY

Каждая структура ihEntries из каталога пиктограмм описывает характеристики одной пиктограммы. В листинге 5.2 показан формат структуры ICONDIRENTRY, которая занимает ровно 16 байтов.

Замечание. В Windows не объявлен официальный тип данных ICONDIRENTRY. В некоторых документах эта структура называется IconDirectoryEntry.

Листинг 5.2. ICONDIRENTRY

```
typedef struct tagICONDIRENTRY {
    BYTE   bWidth;
    BYTE   bHeight;
    BYTE   bColorCount;
    BYTE   bReserved;
    WORD   wReserved1;  /* wPlanes; */
    WORD   wReserved2;  /* wBitCount; */
    DWORD  dwBytesInRes;
    DWORD  dwImageOffset;
} ICONDIRENTRY;
```

Предупреждение. В некоторых документах поля wReserved1 и wReserved2 неправильно именуются wPlanes и wBitCount соответственно. Эти поля не используются и должны быть равны нулю.

- bWidth — ширина изображения пиктограммы в пикселах. Возможные значения: 16, 32 и 64.
- bHeight — высота изображения пиктограммы в пикселах. Возможные значения: 16, 32 и 64. Часто выбирается равным bWidth.
- bColorCount — число цветов, присутствующих в изображении пиктограммы. Возможные значения: 2 (монохромное изображение), 8 или 16. Иногда Windows использует эти значения для выбора пиктограммы, подходящей для конкретного режима дисплея. Например, для отображения пиктограммы минимизированной программы на монохромном дисплее Windows будет выбирать 2-цветную пиктограмму, если, конечно, таковая имеется.
- bReserved — не документировано и не используется. Должно быть равным нулю.
- wReserved1 — не документировано и не используется. Должно быть равным нулю. Иногда неправильно помечается wPlanes.
- wReserved2 — не документировано и не используется. Должно быть равным нулю. Иногда неправильно помечается wBitCount.
- dwBytesInRes — размер в байтах ресурса пиктограммы, который состоит из всех пиктограмм файла минус ICONHEADER (См. следующее замечание).

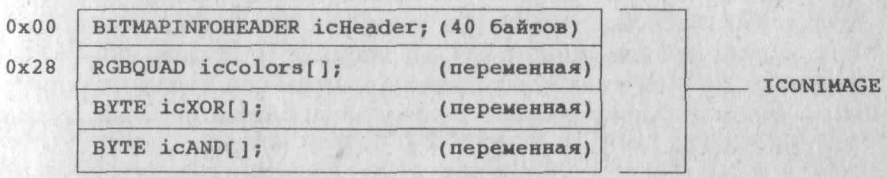
- dwImageOffset — байтовое смещение до начала изображения пиктограммы, начиная от начала файла.

Замечание. В типичном ICO-файле длиной 766 байтов, содержащем одну пиктограмму, поле dwBytesInRes равно 744 (шестнадцатеричное 0x02E8), т.е. длине файла минус 22 байта структуры ICONHEADER, включая массив переменной длины ihEntries записей типа ICONDIRENTRY, каждая из которых занимает ровно 16 байтов.

ICONIMAGE

Пиктограммы фактически являются многосекционными независимыми от устройства растровыми изображениями, состоящими из четырех секций, показанных на рис. 5.4. В Windows не объявлена структура, соответствующая изображенной здесь, потому что размерности массивов icColors, icXOR и icAND изменяются в зависимости от типа пиктограмм в файле. Только 40-байтовый заголовок типа BITMAPINFOHEADER, начинающий каждое изображение пиктограммы, имеет постоянную длину.

Рис. 5.4. Изображение пиктограммы



В листинге 5.3 показан формат изображения пиктограммы в виде псевдо-структуры языка Си. В действительности такой структуры в Си быть не может, т.к. размерности массивов icColors, icXOR и icAND переменны.

Предупреждение. Структура ICONIMAGE в листинге 5.3 является только иллюстрацией и не может быть использована для определения переменных в программе.

Листинг 5.3. ICONIMAGE

```
typedef struct tagICONIMAGE {
  BITMAPINFOHEADER icHeader;
  RGBQUAD          icColors[];
  BYTE             icXOR[];
  BYTE             icAND[];
} ICONIMAGE;
```

- icHeader — каждая пиктограмма начинается 40-байтовым заголовком, формат которого описан в гл. 4. В этой структуре для пиктограмм используются только поля biSize, biWidth, biHeight, biPlanes, biBitCount и biSizeImage. Остальные поля должны содержать нули.
- icColors — массив переменной длины, содержащий структуры RGBQUAD, которые задают значения цвета, используемые маской icXOR, рассматриваемой ниже. Эти значения определяют цвета переднего плана в изображении пиктограммы. Размерность этого массива зависит от типа растрового изображения пиктограммы, подобно тому, как обычное растровое изображение определяет размер своей таблицы цветов (см. табл. 4.1.).
- icXOR — байтовый массив переменной длины, представляющий собой исключающую ИЛИ-маску, а проще говоря, изображение переднего плана пиктограммы. Битовый формат в данном массиве зависит от типа пиктограммы так же, как и значения пикселов в обычном растровом изображении зависят от типа этого изображения. Монохромные пиктограммы хранятся в формате один бит на пиксел, а 8- и 16-цветные пиктограммы — одна тетрада на пиксел.
- icAND — байтовый массив переменной длины, всегда имеющий формат монохромного (один бит на пиксел) изображения независимо от типа пиктограммы и числа используемых цветов. Кратко можно было бы сказать, что icAND-маска "пробивает дыру" в том месте, куда затем накладывается icXOR-маска. См. раздел "Как Windows отображает пиктограммы" далее в этой главе.

Замечание. Заголовок BITMAPINFOHEADER, начинающий изображение пиктограммы, содержит два поля: biWidth и biHight, конфликтующие, как может показаться, с полями bWidth и bHight структуры ICONDIRENTRY. В элементе каталога пиктограмм bWidth и bHight представляют собой ширину и высоту экранного изображения пиктограммы. В заголовке пиктограммы значения biWidth и biHight представляют собой размеры пиктограммы в DIB-формате. Так как каждая пиктограмма имеет два таких растровых изображения, значение biHight из заголовка обычно вдвое больше значения bHight из структуры ICONDIRENTRY. (biWidth и bWidth всегда равны.) Данное очевидное различие может предназначаться для того, чтобы позволить программе обращаться с icXOR- и icAND-масками пиктограммы, как с одним DIB-изображением, хотя это редко делается на практике, а потому во многих ICO-файлах некорректно установлены одинаковые значения для bHight и biHight. Не слишком волнуйтесь по этому поводу — это, по-видимому, не оказывает практического влияния на использование ICO-файлов. Однако при создании своих собственных пиктограмм, наверное, будет лучше установить значение biHight в два раза большим, чем значение bHight. Хороший редактор пиктограмм делает это автоматически.

В листингах 5.4 (ICONIMAGE2), 5.5 (ICONIMAGE8) и 5.6 (ICONIMAGE16) показаны более распространенные на практике версии структуры ICONIMAGE для 2-, 8- и 16-цветных изображений пиктограмм. ICONIMAGE8 и ICONIMAGE16 идентичны, потому что 8-цветные пиктограммы имеют 16-элементный массив цветов.

Листинг 5.4. ICONIMAGE2

```
/* bColorCount == 2 */
typedef struct tagICONIMAGE2 {
    BITMAPINFOHEADER  icHeader;
    RGBQUAD           icColors[2];
    BYTE              icXOR[(bWidth * bHeight) / 8];
    BYTE              icAND[(bWidth * bHeight) / 8];
} ICONIMAGE2;
```

Листинг 5.4. ICONIMAGE8

```
/* bColorCount == 8 */
typedef struct tagICONIMAGE8 {
    BITMAPINFOHEADER  icHeader;
    RGBQUAD           icColors[16];
    BYTE              icXOR[(bWidth * bHeight) / 2];
    BYTE              icAND[(bWidth * bHeight) / 8];
} ICONIMAGE8;
```

Листинг 5.4. ICONIMAGE16

```
/* bColorCount == 16 */
typedef struct tagICONIMAGE16 {
    BITMAPINFOHEADER  icHeader;
    RGBQUAD           icColors[16];
    BYTE              icXOR[(bWidth * bHeight) / 2];
    BYTE              icAND[(bWidth * bHeight) / 8];
} ICONIMAGE16;
```

Совет. Несмотря на принципиальную возможность создания пиктограмм в разных пропорциях, квадратные изображения лучше выглядят на большинстве дисплеев. Например, теоретически можно было бы нарисовать пиктограмму размером 16x64 пиксела, однако на практике из множества такого рода экзотических размеров фактически применяется лишь один — 32x16, предназначенный специально для режима CGA 320x200 точек. Наиболее приемлемыми являются размеры 16x16, 32x32 и 64x64, из которых 32x32 — самый распространенный.

Как Windows отображает пиктограммы

Хорошо разработанная пиктограмма кажется прозрачной, как накладываемая иллюстрация в энциклопедиях, нарисованная на прозрачной пластиковой подложке. Изображение строится по мере того, как вы переворачиваете страницы, сквозь прозрачные области которых виден фон и части других рисунков.

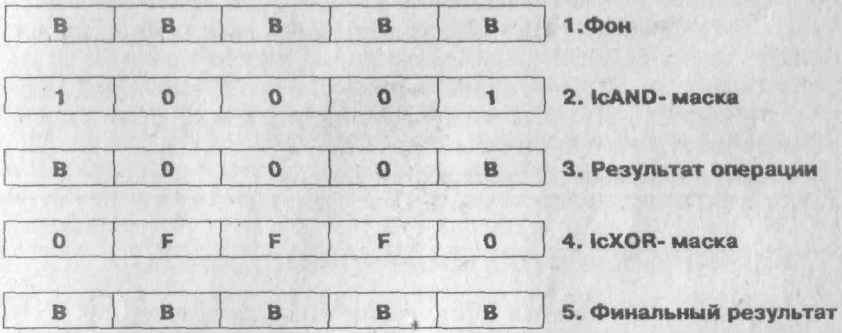
При отображении пиктограмм на рабочем столе Windows или внутри графического окна подобный трюк осуществляется в два приема. Кажется, что изображение пиктограммы наплывает на пикселы фоновой картинки. Для достижения этого эффекта Windows сначала "пробивает дыру" в фоновом изображении, комбинируя монохромную icAND-маску пиктограммы с пикселами фона. Это действие осуществляет предварительную подготовку фона для последующего отображения переднего плана пиктограммы с помощью цветной или монохромной icXOR-маски .

Замечание. Как и любое растровое изображение, icXOR-маска пиктограммы состоит из линий развертки (горизонтальных строк пикселов), хранимых в обратном порядке. Самая нижняя строка развертки изображения хранится в файле первой.

На рис. 5.5 показано, как Windows отображает пиктограмму. Верхняя строка (1) прямоугольников с буквой B представляет собой пикселы фона.

Следующая строка (2) является icAND-маской пиктограммы с 0 (черный) там, где должно появиться изображение переднего плана, и с 1 там, где фон должен просвечивать. Промежуточный результат конъюнкции первой и второй строк показан в следующей (3) строке. Нулевое значение в этой строке представляет собой "дыру" в фоновой картинке, в которую помещается icXOR-маска пиктограммы (4) посредством побитовой операции исключающего ИЛИ. В последней строке (5) показан конечный результат — комбинация пикселов фона (B) и пикселов изображения переднего плана пиктограммы.

Рис. 5.5. Техника отображения пиктограмм



Чтобы понять отношения между пиктограммными масками, рассмотрим типичную для Windows пиктограмму. На рис 5.6 показана icXOR-маска, представляющая изображение переднего плана пиктограммы. Представленная на рис. 5.7 icAND-маска является негативом этого изображения. Комбинация icAND-маски с фоновыми пикселами на экране пробивает дыру (*рис. 5.8*), в которую помещается окончательное изображение. В результате (рис. 5.9) получается пиктограмма, прекрасно вписывающаяся в любую фоновую картинку.

Рис. 5.6. icXOR-маска типичной пиктограммы

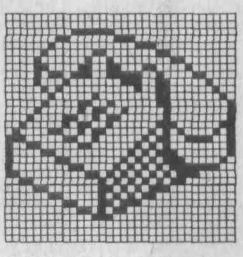


Рис. 5.7. icAND-маска типичной пиктограммы

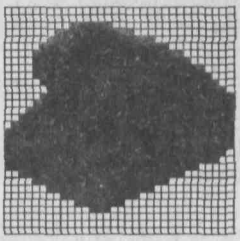


Рис. 5.8. Пробивание дыры в фоновом изображении

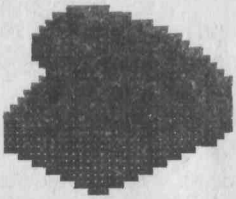
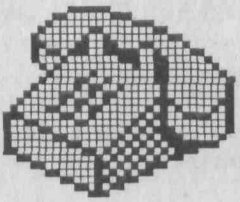


Рис. 5.9. Итоговое отображение пиктограммы



Совет. Любой хороший редактор ресурсов должен уметь создавать icAND- и icXOR-маски автоматически. Все, что вам нужно сделать — это нарисовать изображение переднего плана пиктограммы. Однако, как правило, вам еще потребуется "закрасить" прозрачные области пиктограммы (сквозь которые просвечивает фон) любым прозрачным цветом, имеющемся в вашем редакторе. К сожалению, немногие редакторы пиктограмм в Windows позволяют работать с каждой маской отдельно. (Кстати, у некоторых редакторов ресурсов на компьютерах Macintosh такая возможность имеется.)

Глава 6
ФАЙЛЫ КУРСОРОВ (.CUR)

Строго говоря, *курсор* представляет собой текущую позицию ввода или вывода на символьном дисплее. *Графический указатель* таких устройств, как мышь, отмечает произвольное место на экране. Однако в Windows различия между курсорами и указателями стирается, и оба они рассматриваются как одно и то же. (С технической точки зрения единственным эквивалентом классического курсора в Windows является мигающий вертикальный столбик в текстовом редакторе.)

Каким бы способом ни вызывались графические курсоры, они всегда находятся в файлах с расширением .CUR в виде маленьких монохромных пиктограмм, большинство из которых являются легко узнаваемыми всеми пользователями Windows стрелками. Хотя курсоры и пиктограммы похожи, форматы их файлов не идентичны и имеют несколько важных различий, которые нужно знать программистам.

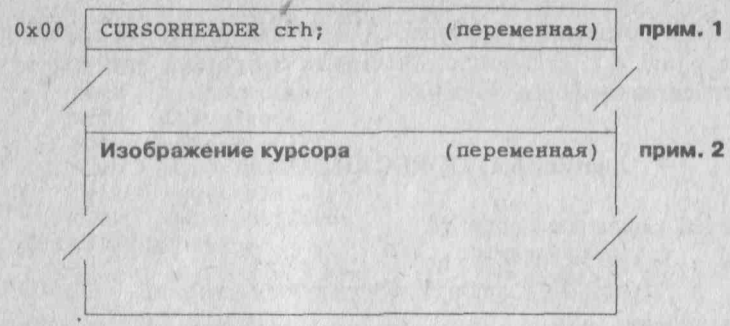
Формат файла

На рис. 6.1 показан общий формат курсорного файла, имя которого, как правило, имеет расширение .CUR, а сам он может содержать один или несколько курсоров.

Примечания к рис. 6.1

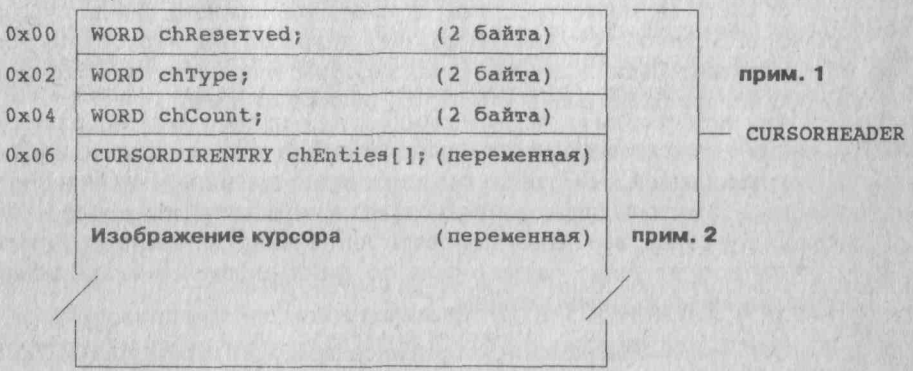
1. В Windows не объявляется структура CURSORHEADER, которая иногда называется CURSORDIR. См. рис. 6.2 для более подробной информации об этом заголовке.
2. Изображения курсоров хранятся в многосекционном DIB-формате, как и пиктограммы, но всегда являются монохромными (черно-белыми). Подобно пиктограммам, изображения курсоров не могут быть сжатыми.

Рис. 6.1. Формат файла курсоров



Как и в файлах пиктограмм, заголовок CURSORHEADER содержит несколько полей, из которых сформирован каталог хранимых курсорных изображений. Программа обращается к нему, чтобы определить местоположение в файле какого-либо курсора. На рис 6.2 показаны заголовок и каталог.

Рис. 6.2. Структура CURSORHEADER



Примечания к рис. 6.2

1. Структура CURSORHEADER заканчивается массивом переменной длины записей типа CURSORDIRENTRY, каждая из которых описывает изображение одного курсора в файле. Значение поля chCount равно числу записей в этом массиве.
2. Это тот же самый DIB-формат изображения, что и на рис 6.1.

Интерфейс с языком Си

В листинге 6.1 показана структура CURSORHEADER на языке Си, соответствующая рис. 6.2. Размерность элемента chEntries в этой структуре зависит от количества курсоров в файле.

Листинг 6.1. CURSORHEADER

```
typedef struct tagCURSORHEADER {
    WORD            chReserved;
    WORD            chType;
    WORD            chCount;
    CURSORDIRENTRY  chEntries[1];
} CURSORHEADER;
```

- chReserved — не документировано и не используется. Должно равняться нулю. Как и в случае файла пиктограмм, это слово всегда равно нулю и, видимо, никак не используется.
- idType — должно равняться 2 (в файлах пиктограмм равно 1). Поскольку в микропроцессоре i86 фирмы Intel старший и младший байты в 16-разрядном слове меняются местами, третий байт каждого файла курсоров равняется 2. Это обстоятельство можно использовать для грубой проверки того, содержит ли данный CUR-файл курсоры. В правильном CUR-файле первые четыре байта равны 00, 00, 02, 00 соответственно.
- chCount — число курсорных изображений в файле. Как и в случае пиктограмм, для CUR-файлов не характерно хранение большого числа курсоров. Так что если это значение чрезмерно велико (например, 10000), то файл курсоров, вероятно, испорчен или вообще не является таковым. Если данное значение равно нулю, то файл определенно не является нормальным курсорным файлом.
- chEntries — массив переменной длины, содержащий структуры CURSORDIRENTRY, каждая из которых описывает различные признаки одного из курсорных изображений. В данном массиве имеется chCount элементов, а индексы изменяются от 0 до chCount - 1.

CURSORDIRENTRY

Каждый элемент массива chEntries описывает характеристики изображения одного из курсоров в файле. В листинге 6.2 показан формат структуры CURSORDIRENTRY, занимающей ровно 16 байтов.

Листинг 6.2. CURSORDIRENTRY

```
typedef struct tagCURSORDIRENTRY {
    BYTE   bWidth;
    BYTE   bHeight;
    BYTE   bColorCount;
    BYTE   bReserved;
    WORD   wXHotSpot;
    WORD   wYHotSpot;
    DWORD  dwBytesInRes;
    DWORD  dwImageOffset;
} CURSORDIRENTRY;
```

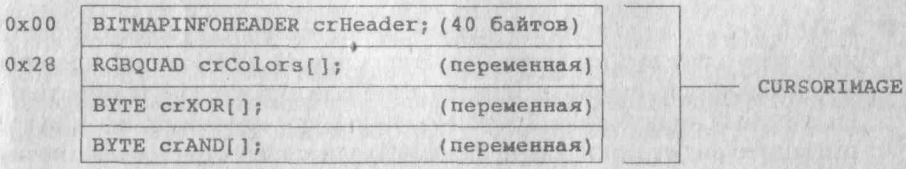
- bWidth — ширина изображения курсора в пикселах.
- bHeight — высота изображения курсора в пикселах. Часто равняется bWidth, хотя это не является обязательным.
- bColorCount — число цветов, используемых в изображении курсора. Равняется двум, т.к. все курсоры в Windows могут быть только черно-белыми.
- bReserved — не документировано и не используется. Должно равняться нулю.
- wXHotSpot — расположение горячей точки (hot spot) курсора относительно левого края его изображения. Эта и другая (wYHotSpot) координата определяют единственный пиксел изображения курсора как активный, называемый также *горячей точкой*. Например, горячая точка у курсора, имеющего вид стрелки, как правило, находится на острие ее наконечника. Когда пользователь с помощью мыши подводит эту стрелку к некоторому объекту на экране и щелкает кнопкой, программа принимает сообщение, содержащее координаты пиксела, который находится под горячей точкой курсора.
- dwBytesInRes — размер ресурса курсора, который состоит из изображений всех курсоров в файле минус заголовок CURSORHEADER. (См. описание этого поля в структуре ICONDIRENTRY в предыдущей главе.)
- dwImageOffset — байтовое смещение от начала файла до начала изображения курсора.

Совет. В отличие от пиктограмм размеры курсоров не ограничены значениями 16, 32 или 64 — теоретически в Windows курсор может быть любого размера. Однако некоторые редакторы ресурсов могут создавать курсоры только одного размера (32x32 пиксела), имеющего наиболее широкое распространение. Кроме того, данный формат аппаратно поддерживается некоторыми видеоадаптерами с помощью специальной схемы высокоскоростного отображения курсора. По этим причинам целесообразно, вероятно, ограничить размеры изображения курсора данным форматом (32x32 пиксела), за исключением специальных случаев — например, увеличенный курсор для портативного дисплея или курсор для плохо видящих пользователей.

CURSORIMAGE

Как и пиктограммы, курсоры хранятся в многосекционном DIB-формате, состоящем из четырех секций, показанных на рис. 6.3.

Рис. 6.3. CURSORIMAGE



В листинге 6.3 показан формат изображения курсора в виде псевдоструктуры языка Си. Размерности массивов crXor и crAND зависят от ширины и высоты курсора.

Предупреждение. Структура в листинге 6.3 носит чисто информационный характер и не может быть использована для определения переменных в программе.

Листинг 6.3. CURSORIMAGE

```
typedef struct tagCURSORIMAGE {
  BITMAPINFOHEADER crHeader;
  RGBQUAD crColors[2];
  BYTE crXOR[1];
  BYTE crAND[1];
} CURSORIMAGE;
```

- crHeader — 40-байтовый заголовок, начинающий описание каждого курсора. Его формат рассматривается в гл. 4. Для курсоров в этой структуре важны только следующие элементы: biSize, biWidth, biHeight, biPlanes, biBitCount и biSizeImage. Все остальные поля должны быть нулевыми.
- crColors — массив, состоящий из двух структур RGBQUAD, содержащих значения цвета, используемые маской icXOR, рассматриваемой ниже. В отличие от пиктограмм, которые могут быть многоцветными, изображения курсоров всегда монохромные, и, следовательно, crColors всегда состоит из двух элементов — обычно черного и белого. (Некоторые устройства могут даже игнорировать этот массив и сразу выводить на экран черно-белое изображение курсора, не полагаясь на то, что массив crColors способен воспроизвести цветной курсор. Теоретически это должно быть возможным. Практически — вероятно, нет.
- crXOR — байтовый массив переменной длины, представляющий собой исключающую ИЛИ-маску, а проще говоря, изображение переднего плана пиктограммы. Хотя границы этого массива переменны, он всегда имеет монохромный формат (один бит на пиксел).
- crAND — байтовый массив переменной длины, всегда имеющий формат монохромного (один бит на пиксел) изображения. Как и в случае пиктограмм, crAND-маска "пробивает дыру" в фоновой картинке, куда затем накладывается crXOR-маска.

Замечание. В структуре CURSORIMAGE имеется элемент crHeader, в котором поле biHeight должно быть в два раза больше поля bHeight структуры CURSORDIRENTRY, что объясняется наличием двух растровых масок — crXor и crAND. Кроме того, поля biPlanes и biBitCount должны равняться 1. Очень важно понимать, что у всех курсоров поле bColorCount структуры CURSORDIRENTRY должно равняться 2 (указывая, что изображение курсора может быть только монохромным), но поле biBitCount элемента crHeader должно равняться 1 (указывая, что в данном изображении использован формат один бит на пиксел). Заметьте, что эти два поля по существу имеют один и тот же смысл, несмотря на различные значения, — изображение курсора всегда монохромное.

Как Windows отображает курсоры

В Windows используется один и тот же двухшаговый метод при отображении пиктограмм и курсоров. Сначала дисплейный драйвер (возможно, поддерживаемый специальной видеоаппаратурой) "пробивает дыру" в фоновой картинке с помощью поразрядной конъюнкции crAND-маски и фоновых пикселов. Затем в эту "дыру" помещается crXOR-маска курсора с помощью

операции исключающего ИЛИ, создавая видимость того, будто изображение наплывает на фон независимо от сложности последнего.

Большинство редакторов курсоров (или универсальных редакторов ресурсов) автоматически создают crAND- и crXOR-маски. При этом от вас требуется лишь нарисовать собственно изображение курсора. В некоторых редакторах используются термины "прозрачный" для областей, сквозь которые должен просвечивать фон, и "инверсный" для областей, которые должны комбинироваться с фоном, производя эффект тени при движении курсора по экрану.

Замечание. Несмотря на то, что Windows отображает пиктограммы и курсоры одним и тем же способом, курсоры могут отображаться аппаратно с помощью специализированных видеоадаптеров. См. гл. 5.

Глава 7
ШРИФТОВЫЕ ФАЙЛЫ (.FNT)

Шрифтовой файл Windows, имя которого имеет расширение FNT, фактически является графическим файлом. Это значит, что он содержит изображения растровых шрифтов и *штриховые команды* (команды вычерчивания линий) для *векторных* шрифтов. Из этих двух типов растровые шрифты явно предпочтительнее. Они обеспечивают высокую скорость вывода на дисплей и хорошо выглядят при определенной разрешающей способности. Однако растровые шрифты далеки от совершенства, поскольку они трудно масштабируются на меньшие или большие размеры, а для печати необходимы дополнительные принтерные шрифты, соответствующие экранным. Векторные шрифты хуже — они отображаются медленно, выглядят невзрачно и плохо печатаются. Немногие пользователи Windows обращают внимание на векторные шрифты, за исключением, быть может, тех, кто применяет их для текстового вывода на перьевой графопостроитель.

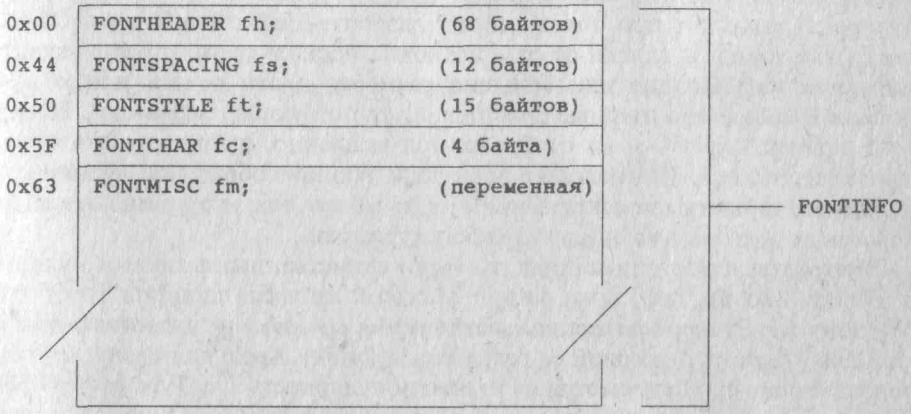
Растровые и векторные шрифты были единственными типами шрифтов в Windows до тех пор, пока фирма Microsoft не ввела шрифты TrueType в Windows 3.1. Эта *технология масштабируемых шрифтов* не является уникальной для Windows — первой ее разработала фирма Apple для своих компьютеров Macintosh. Но несмотря на то, что популярность TrueType растет среди пользователей Windows, растровые и векторные шрифты все еще есть повсюду и важно знать формат их файлов. Вероятно, растровые шрифты уцелеют вместе со шрифтами TrueType. У векторных шрифтов менее определенное будущее. В данной главе рассматриваются оба формата.

К сожалению, формат файла TrueType настолько сложен, что лишь для перечисления его элементов потребовалась бы книга вдвое толще этой. Кроме того, сомнительно, что многие программисты будут обрабатывать шрифтовые файлы TrueType непосредственно. По этим причинам этот формат здесь не рассматривается.

Совет. Отдельные растровые и векторные шрифты хранятся в FNT-файлах и известны также как физические шрифты. Библиотеки шрифтов (файлы с расширением .FON) содержат один или несколько шрифтов в формате, описанном в данной главе. Эти библиотеки фактически являются динамически присоединяемыми библиотеками (DLL), содержащими многочисленные шрифтовые ресурсы и, как правило, информацию о версии. Для извлечения определенного шрифта из библиотеки используйте редактор ресурсов, чтобы открыть FON-файл, а затем сохраните ресурсы требуемого шрифта в FNT-файле.

Шрифтовой файл начинается множеством полей, описывающих характеристики шрифта. В некоторых документах эта информация перечисляется в одной гигантской структуре, но для простоты мы будем рассматривать их по логическим группам. На рис. 7.1 показан общий формат главных пяти групп полей шрифтового файла.

Рис. 7.1. Формат шрифтового файла

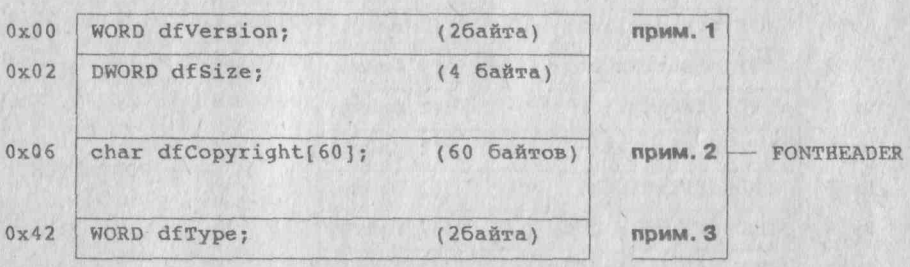


Каждая из показанных на рис. 7.1 пяти секций шрифтового файла содержит различные дополнительные поля с общей информацией о шрифте. Имеются следующие секции: заголовок шрифта (FONTHEADER), межсимвольные промежутки (FONTSPACING), информация о стиле (FONTSTYLE), описания символов (FONTCHAR) и прочая информация (FONTMISC).

Замечание. Структуры, приведенные в данной главе, являются моими собственными разработками, но их элементы согласуются с форматом шрифта из официальной документации, где он обычно представляется в виде одной гигантской структуры FONTINFO.

На рис. 7.2—7.6 показаны форматы всех пяти групп, каждая из которых, за исключением последней, имеет фиксированный размер. Эта последняя группа, FONTMISC, завершается массивом переменной длины, содержащим значения различного характера (см. раздел "Таблица символов шрифта" далее в этой главе).

Рис. 7.2. FONTHEADER



Примечания к рис. 7.2

1. Если это поле равно 0x200, показывая тем самым, что файл содержит шрифт Windows 2.x, то элементы dfFlags, dfAspace, dfBspace, dfCspase, dfColorPointer и dfReserved2 в структуре FONTMISC не представлены. Перечисленные элементы присутствуют в файле только в том случае, если данное поле равно 0x300, показывая тем самым, что файл содержит шрифт Windows 3.x.

2. В официальной документации не указано, должен ли этот массив содержать ASCIIZ-строку (строка с нулем на конце), но, по-видимому, должен.

3. Определяет тип хранимого в файле шрифта — растрового или векторного. См. раздел "Интерфейс с языком Си" для информации о типах шрифтов, хранимых в FNT-файлах.

Рис. 7.3. FONTSPACING

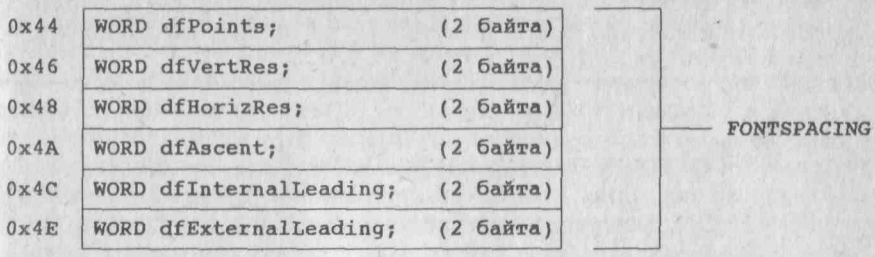


Рис. 7.4. FONTSTYLE

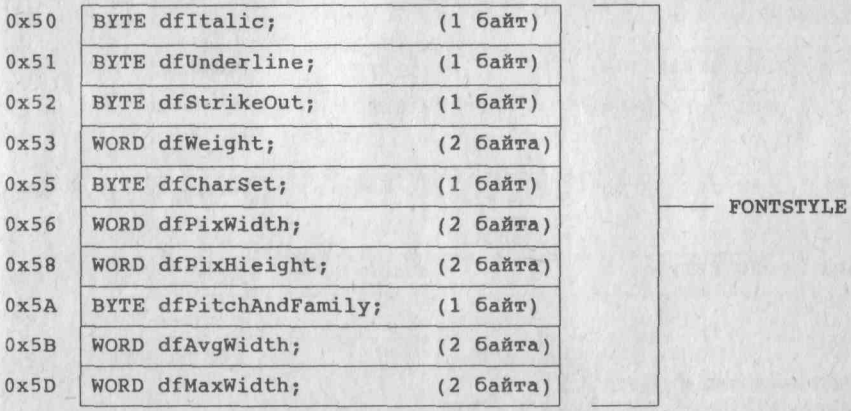


Рис 7.5. FONTCHAR

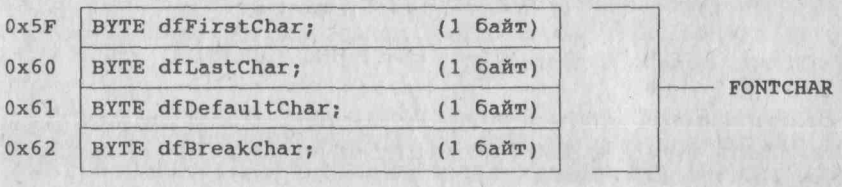
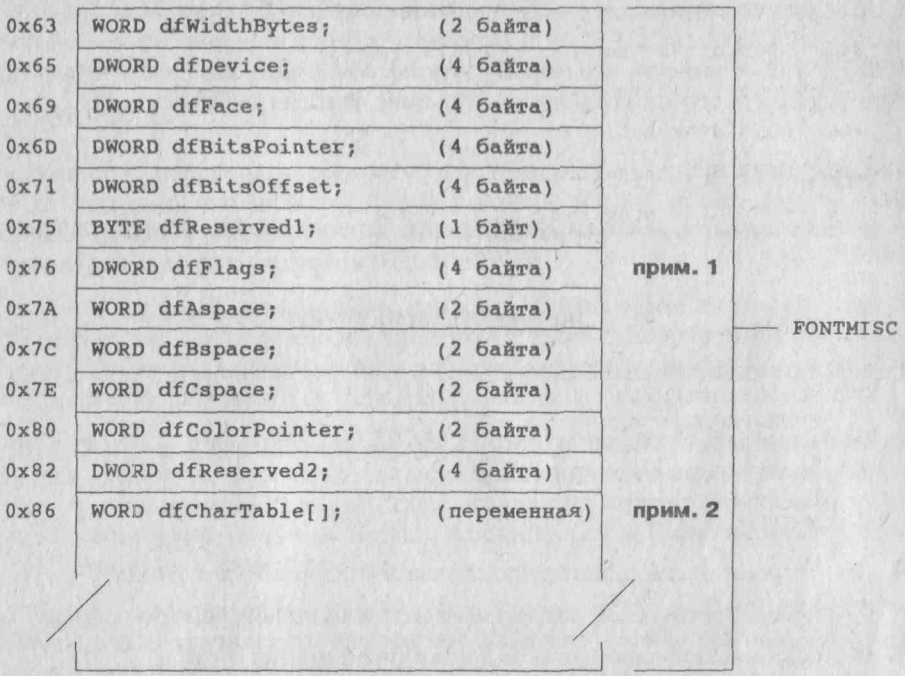


Рис. 7.6. FONTMISC



Примечания к рис. 7.6

1. Поля dfFlags, dfAspace, dfBspace, dfCspase, dfColorPointer и dfReserved2 присутствуют только в шрифтовых файлах Windows 3.x (значение поля dfVersion равно 0x300). В шрифтовых файлах Windows 2.x (значение поля dfVersion равно 0x200) эти поля отсутствуют.
2. Точный формат этого массива переменной длины зависит от типа хранимого шрифта (поле dfType структуры FONTHEADER). Этот массив содержит начертания (glyphs) символов — слова, описывающие в общем виде видимые изображения символов или команды, синтезирующие эти изображения. В Windows 2.x 16-битовые смещения в массиве ограничивали максимальный размер файла до 64K. В Windows 3.x 32-битовые смещения позволяют превысить этот предел. Тем не менее не все драйверы под Windows 3.0 или 3.1 поддерживают 3.x-формат шрифтовых файлов.

Интерфейс с языком Си

Как уже упоминалось, в официальной документации по Windows шрифтовой файл представлен одной огромной структурой, обычно называемой FONTINFO. Я нахожу, что формат этого файла будет легче восприниматься, если разделить его на несколько категорий, рассматриваемых ниже.

FONTINFO

В листинге 7.1 показан формат шрифтового файла в виде структуры FONTINFO.

Листинг 7.1. FONTINFO

```
typedef struct tagFONTINFO {
    FONTHEADER   fh;
    FONTSPACING  fs;
    FONTSTYLE    ft;
    FONTCHAR     fc;
    FONTMISC     fm;
} FONTINFO;
```

- fh — версия, размер, авторские права и информация о типе.
- fs — высота в пунктах, разрешение и другая информация о размере.
- ft — курсив, подчеркивание и другая информация о стиле.
- fc — первый и последний символы шрифта, символ-заместитель и символ-разделитель.
- fm — прочая информация и изображения символов шрифта.

Совет. Сверяйтесь со структурными диаграммами файлов, приведенными в начале данной главы, для уточнения смещений элементов структуры FONTINFO. Для того чтобы прочитать или записать некоторую структуру, позиционируйте на нее внутренний указатель файла с помощью команды seek и прочитайте байты структуры с диска, используя приемы из гл. 3.

FONTHEADER

Каждая из структурных категорий, показанных в листинге 7.1, содержит различные элементы, которые полностью описывают шрифт. В листинге 7.2 представлена первая из них — FONTHEADER — содержащая основные характеристики шрифта. Элементы этой 68-байтовой структуры расположены в начале каждого шрифтового файла.

Листинг 7.2. FONTHEADER

```
typedef struct tagFONTHEADER {
    WORD   dfVersion;
    DWORD  dfSize;
    char   dfCopyright[60];
    WORD   dfType;
} FONTHEADER;
```

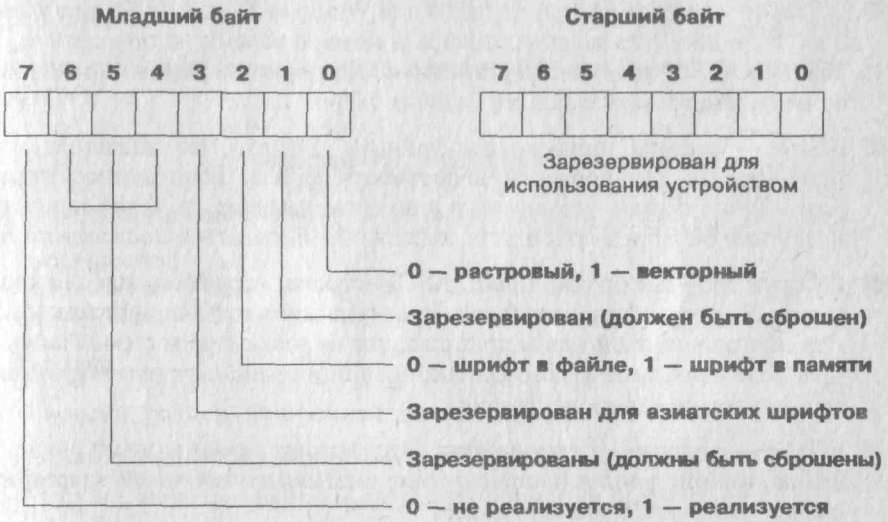
- dfVersion — всегда равняется 0x200 для Windows 2.x или 0x300 для Windows 3.x. Очевидно, что младшая цифра в номере версии не определена, хотя значение 0x301 технически допустимо. До поры до времени присваивайте этим полям только значения 0x200 и 0x300.
- dfSize — размер шрифтового файла в байтах. Это значение можно использовать для проверки целостности файла. Если данное число *не равно* длине файла, указанной в дисковом каталоге, то файл испорчен и не должен обрабатываться (или должен обрабатываться с осторожностью).
- dfCopyright — авторские права. ASCII-строка, вероятно, завершающаяся нулем. Я говорю "вероятно", т.к. проверка множества шрифтовых файлов при подготовке этой главы показала, что во всех случаях строка авторских прав заканчивалась нулем. Однако в официальной документации данное требование не определено явно.
- dfType — определяет тип шрифта. Это одно из самых важных значений в файле, т.к. оно влияет на присутствие и смысл других полей в шрифтовом файле, что будет пояснено далее.

Большинство элементов структуры FONTHEADER имеют очевидное назначение. Последний элемент — dfType — может содержать несколько битовых флажков, показанных на рис. 7.7. Старший байт этого слова рассматривается по-разному в зависимости от того, как шрифт *реализуется*, т.е. как он используется в определенном дисплейном контексте. Когда шрифт реализуется с помощью интерфейса графических устройств (GDI), старший байт обнуляется. Однако драйвер устройства может использовать его для своих собственных целей, запоминая здесь дополнительные флажки, имеющие особый смысл. GDI не изменяет старший байт в dfType, если шрифт реализуется драйвером устройства.

Младший байт слова содержит несколько значимых битовых полей. Если бит 0 равен нулю, то файл содержит растровый шрифт. Если бит 0 установлен, файл содержит векторный шрифт. Если бит 2 (третий по счету) сброшен, элемент dfBitOffset, рассматриваемый далее в этой главе, определяет смещение растрового изображения шрифта в файле. Если бит 2 равен 1, dfBitOffset представляет собой абсолютный адрес изображения, которое может храниться в ПЗУ. Если бит 7 в слове dfType установлен, шрифт реализуется

драйвером устройства. (В шрифтовом файле бит 7 всегда равен нулю, т.к. шрифт должен быть загружен в память, прежде чем быть реализованным.) Бит 3 зарезервирован для использования в азиатских шрифтах и не должен применяться для каких-либо других целей.

Рис. 7.7. Элемент dfType структуры FONTHEADER



FONTSPACING

Начиная с байта 0x44, шрифтовой файл содержит данные о шрифтовых интервалах. В листинге 7.3 эта информация представлена в виде 12-байтовой структуры FONTSPACING.

Листинг 7.3. FONTSPACING

```
typedef struct tagFONTSPACING {
    WORD  dfPoints;
    WORD  dfVertRes;
    WORD  dfHorizRes;
    WORD  dfAscent;
    WORD  dfInternalLeading;
    WORD  dfExternalLeading;
} FONTSPACING;
```

- dfPoints — оптимальный размер в пунктах (1 пункт = 1/72 дюйма), т.е. тот, при котором шрифт выглядит наилучшим образом. Можно использовать и другие размеры, однако результат в этом случае не гарантируется. Поскольку разрешающая способность устройства также влияет на внешний вид шрифта, некоторые шрифты непригодны для отображения даже при оптимальном размере. Желательно, чтобы этот факт учитывался в приложениях (чего обычно не бывает).
- dfVertRes — рекомендуемая разрешающая способность по вертикали в пикселах на дюйм.
- dfHorizRes — рекомендуемая разрешающая способность по горизонтали в пикселах на дюйм. В том случае, когда для данного стиля имеются шрифты различного размера, значения dfVertRes и dfHorizRes можно использовать для подбора шрифта, наиболее подходящего к разрешающей способности выходного устройства. При этом увеличивается вероятность того, что внешний вид отображаемого шрифта будет приемлем по указанному в dfPoints размеру.
- dfAscent — расстояние от верха символьной ячейки до базовой линии (воображаемая горизонтальная линия, на которой стоят прописные буквы и под которой находятся хвостики некоторых строчных литер, например g и y). Используйте поле dfPixHeight структуры FONTSTYLE для определения спуска литеры — расстояния от базовой линии до низа символьной ячейки.
- dfInternalLeading — размер, определяющий высоту диакритических значков (акцентов). Отсчитывается по вертикали от верха символьной ячейки вниз.
- dfExternalLeading — рекомендуемая высота зоны над верхом символьной ячейки. Определяет минимальное расстояние между соседними строками. Может равняться нулю. Никакие выносные элементы литер не должны залезать в эту зону.

FONTSTYLE

Далее идет другая структура, описывающая стиль шрифта и содержащая некоторую дополнительную информацию о ширине и высоте символов. Данная структура показана в листинге 7.4.

Листинг 7.4. FONTSTYLE

```
typedef struct tagFONTSTYLE {
    BYTE  dfItalic
    BYTE  dfUnderline;
    BYTE  dfStrikeOut;
    WORD  dfWeight;
    BYTE  dfCharSet;
    WORD  dfPixWidth;
    WORD  dfPixHeight;
    BYTE  dfPitchAndFamily;
    WORD  dfAvgWidth;
    WORD  dfMaxWidth;
} FONTSTYLE;
```

- dfItalic — равняется 1, если шрифт курсивный. В противном случае равняется нулю. В этом байте используется только младший бит, остальные биты должны быть сброшены.
- dfUnderline — равняется 1, если шрифт с подчеркиванием. В противном случае равняется нулю. В этом байте используется только младший бит, остальные биты должны быть сброшены. Все подчеркивания уже присутствуют в изображении символов на одном и том же расстоянии от базовой линии или непосредственно на ней.
- dfStrikeOut — равняется 1, если шрифт перечеркнутый. В противном случае равняется нулю. В этом байте используется только младший бит, остальные биты должны быть сброшены. Все перечеркивания уже присутствуют в изображении символов. Обычно это короткие горизонтальные черточки, расположенные посередине между верхушкой литеры и базовой линией.

Замечание. Может быть установлены любые комбинации значений dfItalic, dfUnderline и dfStrikeOut. Чтобы обеспечить все возможные комбинации, требуется семь версий одного и того же шрифтового стиля — нормальная, курсивная, с подчеркиванием, с перечеркиванием, курсивная с подчеркиванием, с подчеркиванием и перечеркиванием и курсивная с подчеркиванием и перечеркиванием.

- dfWeight — относительный вес шрифта, или плотность, изменяющаяся от 0 (вес не определен) до 1000 (сверхжирный). Малые значения представляют светлые начертания, большие — жирные. Значение 400 соответствует нормальному начертанию, 700 — как правило, жирному. В табл. 7.1 перечислен полный набор весовых констант, объявленных в windows.h. Большинство поставляемых с Windows шрифтов имеют веса FW_DONTCARE, FW_NORMAL (синоним FW_REGULAR) и FW_BOLD. Другие веса используются редко.

Табл. 7.1.

Весовые константы шрифта

Константа	Значение
FW_DONTCARE	0
FW_THIN	100
FW_EXRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_BLACK	900
FW_HEAVY	900

- dfCharSet — набор символов шрифта. Этот байт равен ANSI_CHARSET (0) для шрифтов Windows, SYMBOL_CHARSET (2) для символических шрифтов (например, набор математических символов) или OEM_CHARSET (255) для шрифтов, совместимых с MS-DOS.

- dfPixWidth — ширина символьной ячейки. Для моноширинных растровых шрифтов значение этого поля представляет собой максимальную ширину изображения для всех символов. Для пропорциональных растровых шрифтов содержит нуль, а для каждой литеры определяется своя собственная ширина. Для векторных шрифтов значение этого поля представляет собой горизонтальный шаг сетки, применяемой для оцифровывания шрифта. Вы можете использовать его как контрольное значение при отображении шрифта в его исходной форме, но это не имеет особого смысла для векторных символов.

- dfPixHeight — высота символьной ячейки в линиях развертки. Значение этого поля всегда отлично от нуля и постоянно даже для пропорциональных шрифтов. Для векторных шрифтов это поле, как и в случае dfPixWidth, представляет собой вертикальный шаг сетки, по которой оцифровываются литеры.

- dfPitchAndFamily — любая константа или комбинация констант из табл. 7.2, которая описывает шаг шрифта (иными словами, его горизонтальный интервал) и семейство — дескриптивный термин, описывающий внешний вид шрифта приближенно.

Табл. 7.2.

Константы, описывающие шаг и семейство шрифта

Константа	Значение	Назначение
FF_DONTCARE	0x00	Семейство не определено.
без имени	0x01	Шрифт с переменным шагом. Может комбинироваться с другими константами из данной таблицы, хотя, видимо, ее использование вместе с FF_MODERN — явное противоречие.
FF_ROMAN	0x10	Пропорциональный шрифт с засечками (поперечные черточки на концах вертикальных элементов литеры).
FF_SWISS	0x20	Пропорциональный шрифт без засечек.
FF_MODERN	0x30	Шрифт с фиксированным шагом, или моноширинный
FF_SCRIPT	0x40	Курсивный шрифт, напоминающий рукописный.
FF_DECORATIVE	0x50	Специальные символы или новейшие шрифты

Предупреждение. Не изобретайте своих собственных имен и значений констант, описывающих шаг и семейство шрифта, а применяйте только перечисленные в табл. 7.2. Ожидается, что в будущих версиях Windows будут новые дополнительные имена шрифтов, которые могут конфликтовать с вашими. Если ваш шрифт не может быть классифицирован приведенными выше константами, используйте FF_DONTCARE.

- dfAvgWidth — имеет тот же смысл, что и dfPixWidth, для шрифтов с фиксированным шагом, или моноширинных. Для шрифтов с переменным шагом, или пропорциональных, равняется ширине прописной буквы "X".
- dfMaxWidth — имеет тот же смысл, что и dfPixWidth, для шрифтов с фиксированным шагом, или моноширинных. Для пропорционального шрифта равняется ширине самой широкой его литеры.

FONTCHAR

Многие шрифты содержат рисунки всех (или, по крайней мере, большинства) символов с ASCII-кодами от 0 до 255. Тем не менее возможен ограниченный набор литер, например в шрифте, содержащем специальные символы. Структура FONTCHAR показывает, какие символы содержит шрифт, а также включает в себя пару спецсимволов (листинг 7.5). Эта структура занимает 4 байта.

Листинг 7.5. FONTCHAR

```
typedef struct tagFONTCHAR {
    BYTE  dfFirstChar;
    BYTE  dfLastChar;
    BYTE  dfDefaultChar;
    BYTE  dfBreakChar;
} FONTCHAR;
```

- dfFirstChar — значение ASCII-кода первой литеры шрифта из диапазона 0—255. Так как в некоторых шрифтах определены не все символы, программа всегда должна использовать dfFirstChar для вычисления индексов в таблице dfCharTable структуры FONTMISC, рассматриваемой ниже.
- dfLastChar — значение ASCII-кода последней литеры шрифта из диапазона 0—255, большее или равное dfFirstChar. Шрифт должен определять изображения всех символов с ASCII-кодами от dfFirstChar до dfLastChar.
- dfDefaultChar — относительный код символа, используемого драйвером устройства для отображения литер с кодами вне диапазона dfFirstChar—dfLastChar. ASCII-код этого символа-заместителя вычисляется как сумма dfFirstChar + dfDefaultChar. Символ-заместитель должен быть видимым, а не пробелом. Традиционно является точкой для растровых и векторных шрифтов и квадратиком для TrueType.
- dfBreakChar — символ, используемый как межсловный разделитель (обычно пробел). ASCII-код этого символа-разделителя вычисляется как сумма dfFirstChar + dfBreakChar. Так как первая литера во многих шрифтах является пробелом (ASCII-код 32), dfBreakChar обычно равен нулю, определяя таким образом пробел как символ обрыва строки для переноса слов и тому подобных применений.

Предупреждение. Не думайте, что все Windows-программы относятся уважительно к определяемому в dfBreakChar символу-разделителю. Текстовые редакторы могут определить свои собственные ASCII-коды для разделителей и игнорировать значение dfBreakChar.

FONTMISC

Последняя структура в шрифтовом файле, называемая FONTMISC, содержит прочие характеристики шрифта, а также изображения его символов. Эта структура представлена в листинге 7.6 и имеет переменную длину благодаря своему последнему элементу — массиву dfCharTable.

Совет. Не представляет особых затруднений чтение и запись структуры отдельными полями или в специально созданную новую структуру фиксированной длины без массива dfCharTable в конце. Программа должна читать этот массив отдельно, после определения его размера и некоторых других характеристик.

Листинг 7.6. FONTMISC

```
typedef struct tagFONTMISC {
    WORD   dfWidthBytes;
    DWORD  dfDevice;
    DWORD  dfFace;
    DWORD  dfBitsPointer;
    DWORD  dfBitsOffset;
    BYTE   dfReserved1;
    DWORD  dfFlags;
    WORD   dfAspace;
    WORD   dfBspace;
    WORD   dfCspace;
    WORD   dfColorPointer;
    WORD   dfReserved2;
    WORD   dfCharTable[1];
} FONTMISC;
```

- dfWidthBytes — число байтов в строках растрового изображения символов. Это значение всегда четно, что гарантирует выравнивание адресов строк изображения на границу слова. Векторные шрифты не используют это поле.
- dfDevice — байтовое смещение от начала файла до ASCIIZ-строки с именем устройства, находящейся в массиве dfCharTable. Эта строка определяет устройства, для которого предназначен данный шрифт, например Plotter или Printer. Поле dfDevice должно равняться нулю для обобщенных шрифтов, у которых нет имен устройств, и обязательно присутствовать в тех шрифтах, которые реализуются драйвером.
- dfFace — байтовое смещение от начала файла до ASCIIZ-строки с наименованием гарнитуры, находящейся в массиве dfCharTable. Имя гарнитуры должно как можно более кратко описывать шрифт и принадлежать типовому списку шрифтов. Например, Courier или Roman. В отличие от dfDevice данное поле не должно быть нулевым.
- dfBitsPointer — в шрифтовых файлах не используется и равняется нулю. Когда GDI загружает некоторый шрифт в память, он устанавливает этот указатель на адрес растровых изображений символов шрифта. Этот адрес всегда четный.

- dfBitsOffset — обычно содержит байтовое смещение от начала файла до изображений символов растрового шрифта или до штриховых команд векторного шрифта. Если бит 2 поля dfType структуры FONTHEADER установлен (см. листинг 7.2), dfBitsOffset содержит абсолютный адрес (вероятно, в ПЗУ) начала растровых изображений символов.
- dfReserved1 — этот байт зарезервирован для выравнивания на четный адрес идущую сразу вслед за ним таблицу символов шрифта Windows 2.x. Хотя данное поле, очевидно, не используется для других целей, хранение здесь посторонних данных не рекомендуется.

Предупреждение. Следующие шесть элементов, начиная с dfFlags и кончая dfReserved2, присутствуют только в шрифтах Windows 3.0 или 3.1 (поле dfVersion структуры FONTHEADER равно 0x300). Не пытайтесь читать или записывать их в шрифтах Windows 2.x.

- dfFlags — определяет формат растровых изображений символов шрифта согласно перечисленным в табл. 7.3. Однако в настоящее время поддерживаются только два значения — DFF_FIXED и DFF_PROPORTIONAL.

Табл. 7.3.

Значения поля dfFlags в Windows 3.x

Константа	Значение	Назначение
DFF_FIXED	0x0001	Шрифт с фиксированным шагом (моноширинный).
DFF_PROPORTIONAL	0x0002	Пропорциональный шрифт.
DFF_ABCFIXED	0x0004	Шрифт с фиксированным шагом, в котором ширина символов равняется сумме dfAspace + dfBspace + dfCspace.
DFF_ABCPROPORTIONAL	0x0008	Пропорциональный шрифт, где в каждом рисунке символа определяются свои собственные размеры A, B и C, а общая ширина символьной ячейки равняется их сумме.
DFF_1COLOR	0x0010	Монохромный (одноцветный) шрифт.
DFF_16COLOR	0x0020	16-цветный шрифт.
DFF_256COLOR	0x0040	256-цветный шрифт.
DFF_RGBCOLOR	0x0080	Полноцветный (true color) шрифт.

- dfAspace — определяет размер "A" для символа — расстояние от текущей позиции вывода до левого края символьной ячейки. Может иметь отрицательное значение для подстрочных символов.
- dfBspace — определяет размер "B" для символа — ширину символьной ячейки.
- dfCspace — определяет размер "C" для символа — расстояние от правого края символьной ячейки до позиции вывода следующего символа. Может иметь отрицательное значение для надстрочных символов.

Замечание. Сумма dfAspace + dfBspace + dfCspace равна общему горизонтальному приращению для всех символов моноширинного шрифта.

- dfColorPointer — в настоящее время не используется. Данное поле предназначено для хранения байтового смещения от начала файла до таблицы цветов шрифта. При пустом значении этого указателя шрифт не имеет таблицы цветов. (На самом деле это поле должно быть нулевым, т.к. цветные шрифты не поддерживаются в настоящих версиях Windows.)
- dfReserved2 — не документировано и не используется.

Совет. Чтобы проверить, поддерживает ли устройство вывода шрифты Windows 3.x, содержащие предыдущие шесть полей, передайте аргумент типа RASTERCAPS в функцию GetDeviceCaps() и проверьте, содержит ли возвращенное значение бит RC_BIGFONT. Если это так, устройство поддерживает 3.x-шрифты, превышающие 64K. Если нет — поддерживаются лишь шрифты Windows 2.x, а шрифты, большие чем 64K, не могут использоваться на данном устройстве.

- dfCharTable — массив переменной длины, содержащий ASCII-строки, растровые изображения или векторные смещения. Формат, размер и содержимое этого массива, описанного ниже, зависит от типа шрифта.

Таблица символов шрифта

Последнее поле в шрифтовом файле является массивом переменной длины, который для удобства был объявлен в предыдущем разделе как

```
WORD dfCharTable[1];
```

Такое объявление даже отдаленно не отражает истинного характера этой таблицы. На самом деле этот массив является сложной коллекцией структур-описателей, растровых изображений, векторных штриховых команд,

ASCII-строк и другой информации, форматы которых зависят от значений других полей файла.

Замечание. Во многих случаях не представляется возможным прочитать таблицу символов шрифтового файла как массив слов, т.к. часто он бывает больше 64K — максимального размера, который способны обрабатывать многие компиляторы Си. Перед тем как попытаться загрузить массив dfCharTable в память, прочитайте все приведенные ниже описания его форматов. Обработка этой информации требует внимательного программирования.

Если изображения символов находятся не в ПЗУ (что определяется полем dfType), массив dfCharTable содержит до пяти типов полей, перечисленных в табл. 7.4.

Табл. 7.4.

Типы полей в таблице символов шрифта

Тип информации	Назначение
Таблица описателей	Серия структур-описателей, определяющих расположение и размер растровых изображений символов и векторные штриховые команды, которые также хранятся в таблице символов.
Растровые изображения символов шрифта	Рисунки символов растрового шрифта.
Векторные штриховые команды	Штриховые команды для векторного шрифта.
Имя устройства	ASCIIZ-строка с именем устройства, для которого предназначен данный шрифт. Присутствует, если dfDevice не равен нулю.
Имя шрифта	ASCIIZ-строка, содержащая наименование гарнитуры. Адресуется элементом dfFace. Должна всегда присутствовать.

Замечание. Байтовый размер строки, содержащей имя устройства или наименование гарнитуры, на единицу больше длины соответствующей строки и не ограничен. Однако, как правило, в этих строках содержится не более 15—20 символов.

Моноширинные шрифты

Чтобы разобраться в сложностях таблицы символов шрифтового файла, начнем с простейшего случая: растровый шрифт Windows 2.x с фиксированным шагом, составленный из монохромных точечных рисунков. В шрифтах данного вида dfCharTable начинается таблицей описателей, представляющей собой ряд структур, показанных в листинге 7.7. В dfCharTable имеется по одной структуре типа GLYPH2 для каждого символа с кодом из диапазона dfFirstChar—dfLastChar (см. листинг 7.5) плюс два дополнительных элемента, отмечающих конец таблицы. Таким образом, общее число структур-описателей определяется выражением

```
dfLastChar - dfFirstChar + 2
```

Листинг 7.7. GLYPH2

```
typedef struct tagGLYPH2 {
  WORD gWidth;
  WORD gOffset;
} GLYPH2;
```

- gWidth — ширина растрового изображения символа в пикселах.
- gOffset — байтовое смещение от начала файла до изображения данного символа.

Замечание. В таблицу символов включены две дополнительные структуры-описателя. Первый дополнительный символ должен быть пробелом. Второй дополнительный описатель, использующийся в качестве метки конца таблицы, должен содержать только нулевые байты. Векторный шрифт использует второй дополнительный описатель для определения размера предшествующей структуры, а растровый шрифт использует только первый дополнительный описатель. Тем не менее оба описателя всегда должны присутствовать в файле.

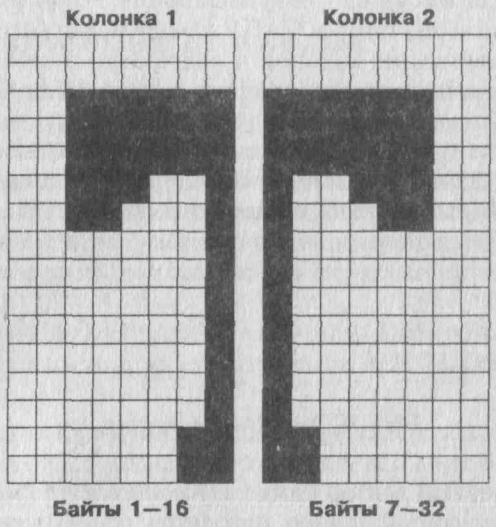
Растровые изображения символов шрифта хранятся иначе, нежели обычные растровые изображения. Чтобы представить себе этот формат, вообразите ряд вертикальных столбцов шириной 8 битов каждый, как показано на рис. 7.8. Здесь показан символ шириной 16 битов и высотой 16 битов. Первый байт этого изображения определяет первые 8 разрядов верхней линии развертки. Второй байт определяет 8 битов второй строки того же самого столбца. Таким образом, байты с 1-го по 16-й определяют левую половину изображения. Байты с 17-го по 32-й определяют правую часть изображения. Неиспользуемые пикселы справа от рисунка символа (в данном случае биты

9—16) сбрасываются и игнорируются. Пропорциональные шрифты используют тот же самый формат, но каждый рисунок потенциально может иметь собственную ширину, отличную от определенной в структуре GLYPH2.

Замечание. Изображения узких символов хранятся в полных байтах с нулевыми битами справа от рисунка. Например, на рис. 7.8 столбец 1 может определять изображение символа, имеющее ширину от 1 до 8 пикселов. Разумеется, однобитовые символы не слишком полезны, однако маленькие шрифты с шириной символов 4—5 пикселов могут реально использоваться.

Вслед за таблицей описателей и растровыми изображениями символов идут дополнительные байты, содержащие имена устройства и гарнитуры, соответственно адресуемые элементами dfDevice и dfFace структуры FONTMISC (см. листинг 7.6). Однако точный порядок следования этих строк и растровых изображений не определен. Для поиска нужных данных используйте элементы gOffset структур-определителей и значения соответствующих байтовых смещений.

Рис. 7.8. Пример растрового символа



Растровые шрифты Windows 3.x весьма похожи на 2.x-шрифты, но имеют другой формат таблицы описателей. Если элемент dfVersion структуры FONTHEADER равен 0x300 (см. листинг 7.2), таблица описателей символов состоит из структур типа GLYPH3, показанного в листинге 7.8.

Листинг 7.8. GLYPH3

```
typedef struct tagGLYPH3 {
  WORD gWidth;
  DWORD gOffset;
} GLYPH3;
```

Данная структура весьма похожа на GLYPH2, за исключением того, что элемент gOffset имеет тип DWORD (32-разрядное число). Это позволяет шрифтовым файлам Windows 3.x быть более длинными, т.к. смещения растровых изображений не ограничены пределом 65535, как в шрифтах Windows 2.x. В остальном растровые изображения 3.x-шрифтов ничем не отличаются от описанных выше.

Векторные шрифты

Существует два вида векторных шрифтов — пропорциональные и моноширинные. Многие пользователи Windows полагают, что все векторные шрифты пропорциональные, однако возможен и моноширинный шрифт, в котором все символы имеют одну и ту же ширину. Например, разработанный для графопостроителя на станции САПР шрифт может быть моноширинным для простоты выравнивания колонок и подготовки таблиц.

В моноширинных векторных шрифтах массив dfCharTable для каждого символа содержит пару байтов, хранящую смещение до начала соответствующих штриховых команд, определяющих все перемещения, необходимые для вычерчивания данного символа. Примерный фрагмент векторной таблицы символов может выглядеть следующим образом:

```
00 10 (0x1000)
80 10 (0x1080)
. . . . . .
00 20 (0x2000)
60 20 (0x2060)
```

Поскольку в процессорах фирмы Intel байты в словах меняются местами, хранимые числа указаны в круглых скобках рядом с каждой парой байтов. В данном примере первый вектор символа находится по смещению 0x1000 от начала файла. Следующий вектор находится по смещению 0x1080 и т.д. Длины векторов могут быть найдены посредством вычитаний этих смещений. Так, в данном примере длина первого вектора равна 0x80 байтам (0x1080 - 0x1000). Нулевая пара байтов отмечает конец таблицы символов. Пропорциональный векторный шрифт использует похожий формат, но в этом случае dfCharTable состоит из структур типа VECTORGLYPHS (листинг 7.9).

Листинг 7.9. VECTORGLYPHS

```
typedef struct tagVECTORGLYPH {
  WORD vgOffset;
  WORD vgWidth;
} VECTORGLYPH;
```

- vgOffset — байтовое смещение от начала файла до векторных штриховых команд для данного символа.
- vgWidth — ширина символа в пикселах, использующаяся для определения позиции следующего символа на этой же строке.

Векторные штриховые команды, расположенные в таблице символов по смещению vgOffset, состоят из цепочек байтов или слов в зависимости от размера оцифровывающей сетки. (См. элементы dfPixWidth и dfPixHeight структуры FONTSTYLE в листинге 7.4.) Они представляют собой команды относительного перемещения при вычерчивании изображения символа. Поскольку векторные символы вычерчиваются сегментами линий, они могут серьезно уменьшать скорость вывода в программах. Поэтому векторные шрифты приемлемы лишь для относительно медленных устройств, например для перьевых графопостроителей.

Неподдерживаемые структуры

Шрифтовые файлы Windows 3.x разработаны таким образом, что могут состоять еще и из других структур-описателей в зависимости от значения dFlags структуры FONTMISC (см. листинг 7.6). В настоящее время этот флажок может принимать лишь значения DFF_FIXED и DFF_PROPORTIONAL, и на практике рассматриваемые ниже структуры не используются. Тем не менее они являются частью официального определения шрифтового файла и приводятся здесь для справки на случай их активизации в будущих версиях Windows.

Если dFlags равен DFF_ABCFIXED или DFF_ABCPROPORTIONAL, таблица описателей состоит из структур типа ABCGLYPH (листинг 7.10).

Листинг 7.10. ABCGLYPH

```
typedef struct tagABCGLYPH {
    WORD   agWidth;
    DWORD  agOffset;
    FIXED  agAspace;
    FIXED  agBspace;
    FIXED  agCspace;
} ABCGLYPH;
```

- agWidth — ширина символа в пикселах.

- agOffset — смещение до изображения символа от начала файла.
- agAspace — действительное число с фиксированной точкой, представляющее собой расстояние от текущей позиции вывода до левого края символьной ячейки.
- agBspace — действительное число с фиксированной точкой, представляющее собой ширину символьной ячейки.
- agCspace — действительное число с фиксированной точкой, представляющее собой расстояние от правого края символьной ячейки до позиции вывода следующего символа.

Замечание. Последние три элемента в пропорциональных шрифтах эквивалентны соответствующим полям dfAspace, dfBspace и dfCspace моноширинных 3.x-шрифтов (см. структуру FONTMISC в листинге 7.6). Общая ширина символа равна сумме этих трех ABC-значений.

В листинге 7.11 показано, как структура FIXED, используемая в полях agAspace, agBspace и agCspace структуры ABCGLYPH, объявляется в windows.h.

Листинг 7.11. FIXED

```
typedef struct tagFixed {
  UINT fract;
  int  value;
} FIXED;
```

- fract — дробная часть действительного числа с фиксированной точкой.
- value — целая часть действительного числа с фиксированной точкой.

Замечание. В структуре FIXED предусмотрен метод представления дробной части числа в форме целого. Помимо того, что данная структура используется здесь в определении ширин символов 3.x-шрифтов, она переходит по наследству к шрифтам TrueType. Для более подробной информации см. структуры TTPOLYCURVE и TTPOLYGONHEADER, а также функцию GetGlyphOutline() в документации по Windows.

Если dFlags равен DFF_1COLOR, DFF_16COLOR или DFF_RGBCOLOR, таблица описателей состоит из структур типа COLORABCGLYPH, показанного в листинге 7.12.

Листинг 7.12. COLORABCGLYPH

```
typedef struct tagCOLORABCGLYPH {
    WORD    cgWidth;
    DWORD   cgOffset;
    WORD    cgHeight;
    FIXED   cgAspace;
    FIXED   cgBspace;
    FIXED   cgCspace;
} COLORABCGLYPH;
```

Элементы этой структуры имеют тот же смысл, что и в структуре ABCGLYPH, за исключением поля cgHeight, содержащего высоту символа в строках развертки. В этих шрифтах число битов на пиксел зависит от числа цветов, определяемых шрифтом. Одноцветные шрифты используют 1 бит на пиксел, 16-цветные — 4 бита на пиксел, 256-цветные — 8 битов на пиксел и полноцветные — 32 бита на пиксел в форме структуры RGBQUAD, как описано в гл. 4 для обычных растровых изображений.

Глава 8
МЕТАФАЙЛЫ (.WMF)

Файл метафайла (далее просто метафайл) очень похож на музыкальную запись, где вместо музыки записаны команды интерфейса графических устройств (GDI). Для того чтобы отобразить метафайл, программа передает эти команды функции PlayMetaFile(), которая, подобно магнитофону, воспроизводит визуальную "мелодию".

Метафайлы обеспечивают независимые от устройства средства хранения и выборки графической информации. Эти файлы создаются без особого труда и показывают прекрасные результаты на устройствах с различной разрешающей способностью и прочими возможностями. Кроме того, поскольку команды метафайла остаются на диске, значительно экономится память, что особенно важно для программ, отображающих большое количество сложных изображений. Так как длина метафайла практически не ограничена (верхний предел равен 4 гигабайтам), он может также дополнять память компьютера.

В отличие от растровых изображений, хранящих графическую информацию непосредственно в виде пикселов, метафайлы идеально подходят для таких изображений, как карты, диаграммы, архитектурные чертежи и другие картинки, состоящие из перекрывающихся фрагментов. Так, например в САПР, метафайлы могут применяться для запоминания данных. Они также полезны при передаче изображений в их собственных форматах в системный буфер Windows (clipboard) для использования их другими приложениями. Если изображение может быть нарисовано с помощью команд GDI, оно может быть передано другой программе как метафайл. При этом подразумевается, что программа знает, как интерпретировать команды метафайла.

Несколько популярных приложений Windows используют WMF-файлы для хранения графической информации. Текстовый процессор Word for Windows (с помощью которого был создан американский оригинал этой книги) имеет много готовых иллюстраций (clip-art), оформленных в виде

метафайлов Windows. Графический редактор Micrographix Designer также может импортировать и экспортировать метафайлы.

Замечание. Имена метафайлов Windows не обязательно должны иметь рекомендуемое расширение WMF. Другое популярное расширение — MET. Вы можете также использовать любое другое расширение имени, включая пустое, если есть такое желание.

Формат файла

Существует три вида метафайлов: *стандартный*, *размещаемый* (placeable) и *буферный* (clipboard). (О буферном формате метафайла см. также гл. 12 "Файлы редактора Write".) Стандартный метафайл содержит 18-байтовый заголовок, вслед за которым идут одна или несколько графических команд. В начале размещаемого метафайла дополнительно имеется 22-байтовый заголовок, содержащий информацию о размерах, которая полезна для отображения метафайлов в относительно одинаковых размерах на различных устройствах. Сразу же вслед за этим заголовком идет стандартный метафайл.

Формат стандартного метафайла

Структура стандартного метафайла приведена на рис. 8.1.

Примечания к рис. 8.1

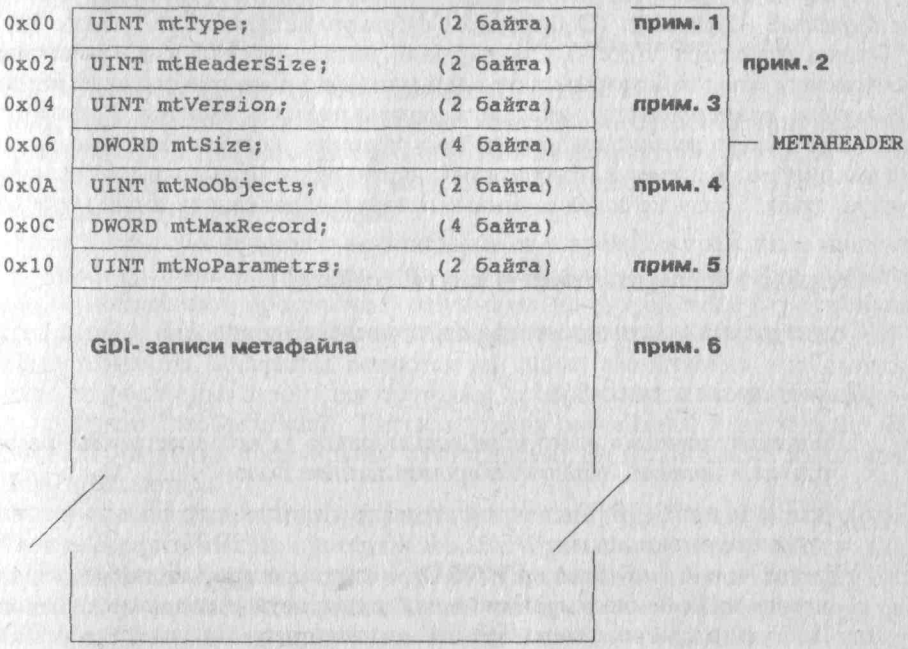
1. В файле значение этого поля всегда равно 1. Когда метафайл формируется в памяти, Windows обнуляет данное поле.
2. Все поля типа UINT в данной структуре иногда неправильно указываются как имеющие тип WORD. В Windows тип UINT определяется как целое число без знака, а WORD — как короткое число без знака. С практической точки зрения между этими двумя типами нет разницы. По крайней мере, с точки зрения компиляторов Си, используемых для программирования в 16-битовом режиме Windows. Тем не менее правильным типом данных в этом случае является UINT.
3. В официальной документации указано, что значение этого поля равно 0x300 для Windows 3.0 и более поздних версий. Правильное значение — 0x0300. К тому же трудно представить, что метка версии метафайла останется неизменной в будущих версиях Windows.
4. Это поле представляет собой *число* объектов метафайла (кисти, перья, растровые изображения и шрифты), существующих одновременно при

его воспроизведении. Данное поле не означает "объектов нет", что можно было бы предположить, судя по его имени.

5. Это поле не используется и, вероятно, должно было бы называться mtReserved. Тем не менее на диаграмме указано официальное имя, которое объявлено в windows.h.

6. Каждая GDI-запись метафайла содержит один вызов GDI-функции. См. раздел "GDI-записи метафайла" в этой главе.

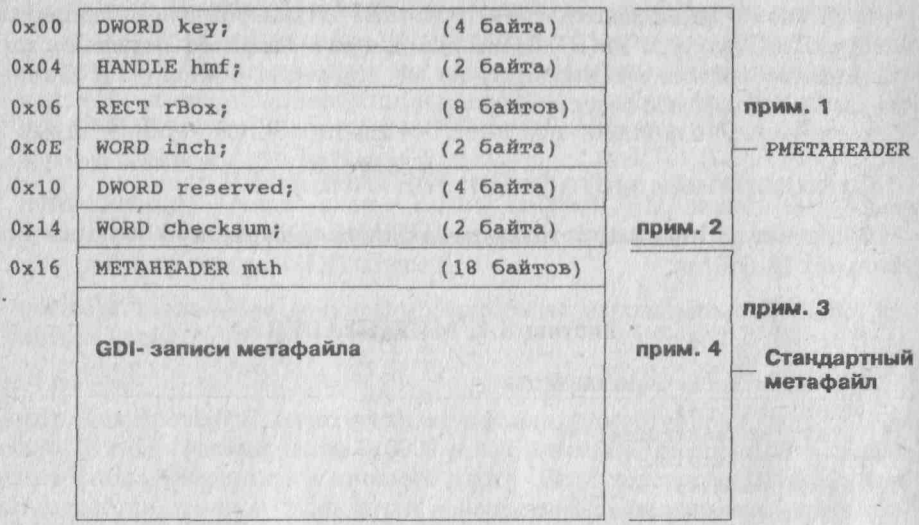
Рис. 8.1. Формат стандартного метафайла



Размещаемый формат метафайла

Размещаемый метафайл начинается 22-байтовым заголовком, показанным на рис. 8.2. Сразу вслед за этим заголовком, содержащим информацию о разрешении и контрольную сумму, следует стандартная структура METAHEADER.

Рис. 8.2. Размещаемый формат метафайла



Примечания к рис. 8.2

1. Метафайловые функции Windows не поддерживают размещаемый формат. Перед тем как вызвать функцию GetMetaFile() для загрузки размещаемого метафайла, необходимо удалить дополнительную 22-байтовую структуру PMETAHEADER. Для этого скопируйте размещаемый метафайл в другой, временный файл, начиная с 16-го байта. Затем вы можете загрузить этот временный файл с помощью функции GetMetaFile().

2. Контрольная сумма есть результат операции исключающего ИЛИ над первыми десятью словами файла. Используется для проверки целостности файла. Удостоверьтесь в том, что правильно установили контрольную сумму при записи или перезаписи размещаемого метафайла. Windows не сделает этого за вас.

3. См. рис. 8.1. Вслед за заголовком размещаемого метафайла идут заголовок стандартного метафайла, а также одна или несколько GDI-записей.

4. Как и в стандартном метафайле, каждая запись размещаемого метафайла содержит вызов одной GDI-функции. См. раздел "GDI-записи метафайла" в этой главе.

Интерфейс с языком Си

В Windows определяется структура METAHEADER для стандартного метафайла. Структура PMETAHEADER для размещаемого метафайла описывается во многих публикациях, но не объявлена в windows.h. Данный раздел охватывает обе структуры.

Стандартный метафайл: METAHEADER

В листинге 8.1 показана структура METAHEADER из windows.h, которая занимает 18 байтов.

Листинг 8.1. METAHEADER

```
typedef struct tagMETAHEADER {
  UINT   mtType;
  UINT   mtHeaderSize;
  UINT   mtVersion;
  DWORD  mtSize;
  UINT   mtNoObjects;
  DWORD  mtMaxRecord;
  UINT   mtNoParametrs;
} METAHEADER;
```

- mtType — Если равняется 0x0000, метафайл формируется в памяти. Если равняется 0x0001, метафайл хранится на диске. Windows присваивает этому полю правильное значение автоматически, но если вы создаете дисковый метафайл вручную (без помощи метафайловых функций Windows), убедитесь, что значение данного поля равно 0x0001.
- mtHeaderSize — длина структуры METAHEADER в словах (а не в байтах). Если значение данного слова не равно 0x0009, то это не стандартный метафайл.
- mtVersion — номер версии Windows, используемой для создания метафайла. Для Windows 3.0 и 3.1 значение этого поля должно равняться 0x0300.
- mtSize — число слов в метафайле. Используется для проверки целостности метафайла. Если значение этого поля не соответствует длине файла, указанной в каталоге диска, то данный файл либо не является метафайлом, либо его содержимое испорчено.

Замечание. Длина файла, указанная в каталоге магнитного диска, измеряется в байтах, а размер mtSize — в словах. Таким образом, он должен быть равен *половине* значения длины файла, указанной в каталоге.

- mtNoObjects — число объектов метафайла (кисти, перья, растровые изображения и шрифты), существующих одновременно при его воспроизведении. Поскольку некоторые объекты могут оказаться удаленными во время воспроизведения, значение этого поля не является *общим* количеством используемых в метафайле объектов, а представляет собой число лишь тех объектов, которые должны существовать в данный момент. Windows использует mtNoObjects при создании таблицы дескрипторов для ссылки на эти объекты.
- mtMaxRecord — число слов в самой длинной GDI-записи метафайла. Очевидно, Windows использует значение этого поля для выделения памяти под буфер чтения GDI-записей.
- mtNoParametrs — не документировано и не используется. Должно равняться нулю.

Совет. Для проверки целостности метафайла, удостоверьтесь, что mtType равно 1, mtHeaderSize равно 0x0009 и mtSize равно измеренной в словах длине файла, указанной в каталоге диска. Если перечисленные условия выполняются, данный файл почти наверняка является корректным метафайлом.

Размещаемый метафайл: PMETAHEADER

Если файл является размещаемым метафайлом, он начинается 22-байтовой структурой PMETAHEADER, показанной в листинге 8.2.

Предупреждение. Windows не распознает размещаемый формат метафайла, и программа сама должна непосредственно работать со структурой PMETAHEADER, которую нужно удалять из файла, прежде чем загружать его с помощью функции GetMetaFile(). Эта структура не объявлена в windows.h.

Листинг 8.2. PMETAHEADER

```
typedef struct tagPMETAHEADER {
    DWORD   key;
    HANDLE  hmf;
    RECT    rBox;
    WORD    inch;
    DWORD   reserved;
    WORD    checksum;
} PMETAHEADER;
```

- key — используется для проверки формата метафайла. Значение этого поля должно равняться 0x9AC6CDD7L.
- hmf — предназначено, очевидно, для хранения дескриптора метафайла, однако согласно официальной документации данный элемент не используется. Во время загрузки метафайла в память здесь можно временно запоминать его дескриптор.
- rBox — описывает наименьшую прямоугольную область, в которой могут быть показаны изображения метафайла. Значения даны в единицах измерения, определяемых полем inch.
- inch — число единиц измерения, используемых при создании метафайла, на одном дюйме. Должно быть меньше 1440 и обычно равно либо 576 (нормальное разрешение), либо 1000 (высокое разрешение).
- reserved — не документировано и не используется. Должно равняться нулю.
- checksum — 16-разрядный результат операции исключающего ИЛИ над первыми десятью словами размещаемого метафайла. Проверяйте с помощью этой суммы целостность метафайла. Удостоверьтесь, что правильно вычислили ее, когда создаете или обновляете на диске свои собственные размещаемые метафайлы.

Совет. Чтобы определить, хранится ли метафайл в размещаемом формате, прочитайте первые два слова в переменную типа DWORD и сравните его с 0x9AC6CDD7L. После этого проверьте корректность параметра checksum. Если перечисленные проверки были успешными, данный файл почти наверняка является размещаемым метафайлом. Если вы обнаружили ошибки, проверьте стандартный метафайл, как описано в предыдущем разделе.

GDI- записи метафайла

Основную часть метафайла составляют одна или более GDI-записей, каждая из которых описывает одну графическую функцию. Метафайлы могут использовать лишь некоторое подмножество полного набора GDI-функций Windows. В общем случае только те функции, которые производят графический вывод и принимают в качестве аргумента дескриптор контекста устройства, могут использоваться в метафайле.

GDI- функции метафайла

В табл. 8.1 перечислены GDI-функции, которые программа может вызывать при создании метафайла. В данной таблице также перечислены связанные с этими функциями идентификаторы и значения констант метафайла в том виде, как они объявлены в windows.h. Значения констант хранятся в записях метафайла и являются ссылками на соответствующие GDI-команды.

Замечание. Существуют функции, которые на самом деле не запоминаются в метафайле, хотя некоторые из них перечислены в табл. 8.1 для справки, а другие и вовсе здесь не упоминаются. Так, например, можно вызвать функцию CreateSolidBrush(), чтобы создать для метафайла объект типа кисть. Однако сама функция не запоминается в метафайле. Вместо этого команда создания (повторного создания) данного объекта генерируется, когда кисть выбирается в контекст устройства метафайла. Можно также вызвать функцию CreateDIBPatternBrush() и другие функции создания объектов, не упоминаемые в таблице. Но именно выбор объектов в контекст устройства метафайла генерирует команду для создания (повторного создания) данного объекта.

Табл. 8.1.

GDI-функции и константы метафайла

GDI-функция	Константа метафайла	Значение константы
AbortDoc()	META_ABORTDOC	0x0052
AnimatePalette()	META_ANIMATEPALETTE	0x0436
Arc()	META_ARC	0x0817
BitBlt()	META_BITBLT	0x0922
Chord()	META_CHORD	0x0830
CreateBitmap()	META_CREATEBITMAP	0x06FE
CreateBitmapIndirect()	META_CREATEBITMAPINDIRECT	0x02FD
Функции кисти	META_CREATEBRUSH	0x00F8
CreateBrushIndirect()	META_CREATEBRUSHINDIRECT	0x02FC
CreateFontIndirect()	META_CREATEFONTINDIRECT	0x02FB
CreatePenIndirect()	META_CREATEPENINDIRECT	0x02FA
CreatePalette()	META_CREATEPALETTE	0x00F7
CreatePatternBrush()	META_CREATEPATTERNBRUSH	0x01F9
Функции области	META_CREATEREGION	0x06FF
нет	META_DELETEOBJECT	0x01F0
нет	META_DIBBITBLT	0x0940
нет	META_DIBCREATEPATTERNBRUSH	0x0142
нет	META_DIBSTRETCHBLT	0x0B41
DrawText()	META_DRAWTEXT	0x062F
Ellipse()	META_ELLIPSE	0x0418
EndDoc()	META_ENDDOC	0x005E

Табл. 8.1. (продолжение)

GDI-функция	Константа метафайла	Значение константы
EndPage()	META_ENDPAGE	0x0050
Escape()	META_ESCAPE	0x0626
ExcludeClipRect()	META_EXCLUDECLIPRECT	0x0415
ExtFloodFill()	META_EXTFLOODFILL	0x0548
ExtTextOut()	META_EXTTEXTOUT	0x0A32
FillRgn()	META_FILLREGION	0x0228
FloodFill()	META_FLOODFILL	0x0419
FrameRgn()	META_FRAMEREGION	0x0429
IntersectClipRect()	META_INTERSECTCLIPRECT	0x0416
InvertRgn()	META_INVERTREGION	0x012A
LineTo()	META_LINETO	0x0213
MoveTo()	META_MOVETO	0x0214
OffsetClipRgn()	META_OFFSETCLIPRGN	0x0220
OffsetViewportOrg()	META_OFFSETVIEWPORTORG	0x0211
OffsetWindowOrg()	META_OFFSETWINDOWORG	0x020F
PaintRgn()	META_PAINTREGION	0x012B
PatBlt()	META_PATBLT	0x061D
Pie()	MTA_PIE	0x081A
Polygon()	META_POLYGON	0x0324
Polyline()	META_POLYLINE	0x0325
PolyPolygon()	META_POLYPOLYGON	0x0538
RealizePalette()	META_REALIZEPALETTE	0x0035
Rectangle()	META_RECTANGLE	0x041B
ResetDC()	META_RESETDC	0x014C
ResizePalette()	META_RESIZEPALETTE	0x0139
RestoreDC()	META_RESTOREDC	0x0127
RoundRect()	META_ROUNDRECT	0x061C
SaveDC()	META_SAVEDC	0x001E
ScaleViewportExt()	META_SCALEVIEWPORTEXT	0x0412
ScaleWindowExt()	META_SCALEWINDOWEXT	0x0410
SelectClipRgn()	META_SELECTCLIPREGION	0x012C
SelectObject()	META_SELECTOBJECT	0x012D
SelectPalette()	META_SELECTPALETTE	0x0234
SetBkColor()	META_SETBKCOLOR	0x0201
SetBkMode()	META_SETBKMODE	0x0102
SetDIBitsToDevice()	META_SETDIBTODEV	0x0D33
SetMapMode()	META_SETMAPMODE	0x0103
SetMapperFlags()	META_SETMAPPERFLAGS	0x0231
SetPaletteEntries()	META_SETPALENTRIES	0x0037
SetPixel()	META_SETPIXEL	0x041F
SetPolyFillMode()	META_SETPOLYFILLMODE	0x0106
нет	META_SETRELABS	0x0105
SetROP2()	META_SETROP2	0x0102
SetStretchBltMode()	META_SETSTRETCHBLTMODE	0x0107
SetTextAllign()	META_SETTEXTALLIGN	0x012E
SetTextCharacterExtra()	META_SETTEXTCHAREXTRA	0x0108

Табл. 8.1. (продолжение)

GDI-функция	Константа метафайла	Значение константы
SetTextColor()	META_SETTEXTCOLOR	0x0209
SetTextJustification()	META_SETTEXTJUSTIFICATION	0x020A
SetViewportExt()	META_SETVIEWPORTEXT	0x020E
SetViewportOrg()	META_SETVIEWPORTORG	0x020D
SetWindowExt()	META_SETWINDOWEXT	0x020C
SetWindowOrg()	META_SETWINDOWORG	0x020B
StartDoc()	META_STARTDOC	0x014D
StartPage()	META_STARTPAGE	0x004F
StretchBlt()	META_STRETCHBLT	0x0B23
StretchDIBits()	META_STRETCHDIB	0x0F43
TextOut()	META_TEXTOUT	0x0521

Замечание. Значения констант в табл. 8.1 состоят из двух частей. Младший байт константы идентифицирует GDI-функцию, а старший байт представляет собой число слов, составляющих суммарную длину параметров, которые передаются этой функции. Например, число 0x0512 идентифицирует функцию TextOut() номером 0x12 и определяет, что для ее параметров потребуется 0x05 слов (10 байтов). Параметр дескриптора контекста устройства не входит в их число и не хранится в метафайле, т.к. различные контексты устройств обеспечиваются самой программой во время воспроизведения метафайла.

METARECORD

Все метафайлы, включая размещаемые, завершаются одной или несколькими записями, которые имеют показанный в листинге 8.3 тип. Это объявленная в windows.h вместе с двумя своими указателями структура METARECORD.

Предупреждение. METARECORD — структура переменной длины, которая не может непосредственно использоваться для определения объектов в программе. Чтобы создать объект типа METARECORD, программа должна прочитать первое двойное слово записи, а затем с помощью описанных в гл. 2 приемов выделить достаточную для объекта память. Однако при использовании в метафайле таких функций, как CreateMetaFile() и GetMetaFile(), Windows автоматически читает и записывает объекты типа METARECORD, и вы смело можете игнорировать данное предупреждение.

Листинг 8.3. METARECORD

```
typedef struct tagMETARECORD {
  DWORD rdSize;
  UINT rdFunction;
  UINT rdParm[1];
} METARECORD;
typedef METARECORD* PMETARECORD;
typedef METARECORD FAR* LPMETARECORD;
```

- rdSize — длина данного объекта типа METARECORD, *включая* rdSize, в 16-разрядных словах. Размеры GDI-объектов метафайла зависят от вызываемой функции (идентифицируется следующим полем) и ее параметров.
- rdFunction — код функции из таблицы 8.1. Это число однозначно идентифицирует GDI-функцию, связанную с данной записью метафайла. *Последняя запись отмечается числом* 0x0000.
- rdParm — передаваемые функции параметры. Хранятся в обратном порядке по сравнению с объявляемым в заголовке функции. Так, например, для функции f(x, y) параметры запоминаются в порядке y, x. Элемент rdParm объявлен как массив 16-разрядных целых слов без знака. Его фактическое содержимое зависит от типов вызываемых функций. Дескриптор контекста устройства никогда не запоминается в метафайле, потому что он будет другим во время воспроизведения метафайла.

Совет. В результате проверок реальных метафайлов оказалось, что Windows записывает в конце файла специальную маркирующую запись. В этой записи поле rdSize равно 0x00000003L, а поле rdFunction равно 0x0000. Элемент rdParm отсутствует. Короче говоря, метафайлы всегда заканчиваются шестью байтами — 03 00 00 00 00 00, — следующими именно в таком порядке.

HANDLETABLE

Когда Windows читает или создает GDI-записи метафайла, подготавливает массив дескрипторов, идентифицирующих каждый объект (например, перо или кисть), используемый функциями метафайла. Размер массива дескрипторов определяется элементом mtNoObjects структуры METAHEADER (см. *листинг 8.1*). *Объект включается в эту таблицу, когда программа вызывает* функцию SelectObject() для выбора объекта в контекст метафайлового устройства.

Маловероятно, что вам когда-либо потребуется самим создавать таблицу дескрипторов метафайла — Windows берет эту заботу на себя. Однако для

сведения структура HANDLETABLE приводится в листинге 8.4 именно так, как она объявлена в windows.h. Эта таблица не запоминается в дисковом метафайле.

Предупреждение. HANDLETABLE — структура переменной длины. Не используйте ее для непосредственного определения переменных в программах. Если вам необходимо создать таблицу дескрипторов, нужно выделить для структуры достаточное количество памяти, как показано в гл. 2. Немногим (если вообще каким-либо) программам когда-нибудь потребуется делать это.

Листинг 8.4. HANDLETABLE

```
typedef struct tagHANDLETABLE {
  HGDIOBJ objectHandle[1];
} HANDLETABLE;
typedef HANDLETABLE* PHANDLETABLE;
typedef HANDLETABLE FAR* LPHANDLETABLE;
```

- objectHandle — единственный элемент структуры. Представляет собой массив 16-разрядных значений типа HGDIOBJ, каждое из которых является дескриптором пера, кисти, растрового изображения или шрифта. Не все используемые метафайлом объекты присутствуют одновременно. Во время воспроизведения метафайла команда META_DELETEOBJECT удаляет существующий объект, чтобы освободить место для нового, сохраняя, таким образом, память для необходимых только в данный момент объектов.

Создание и использование метафайлов

Чтобы создать метафайл на диске, вызовите функцию CreateMetaFile(). В Windows эта функция объявляется следующим образом:

```
HDC CreateMetaFile(LPCSTR filename);
```

- filename — литеральная строка с именем файла или указатель на символьную строку, содержащую имя файла. Имеет значение NULL, если метафайл формируется в памяти, а не на диске.

Функция CreateMetaFile() пытается создать новый метафайл с указанным именем. В случае успеха она возвращает дескриптор контекста устройства того же самого типа, что и функция GetDC() для рисования в окне. Однако дескриптор метафайла не относится к окну или к какому-нибудь реальному

устройству. На самом деле *контекст устройства метафайла* обозначает область памяти, куда помещаются вызовы GDI-функций. Вызывайте CreateMetaFile() следующим образом:

```
HDC hmfDC;
hmfDC = CreateMetaFile((LPSTR)"anyfile.wmf");
```

Если переменная hmfDC равна нулю, то функция CreateMetaFile() не смогла создать файл с указанным именем. При ином результате можно использовать дескриптор контекста устройства для последующих обращений к GDI-функциям, перечисленным в табл. 8.1. Например, для создания и использования пера в метафайле, можно написать следующее:

```
HPEN hPen;
hPen = CreatePen(PS_SOLID, 0, RGB(0, 0, 255));
SelectObject(hmfDC, hPen);
Rectangle(hmfDC, 10, 10, 125, 80);
```

Перо может быть создано только что рассмотренным способом, но может быть уже созданным ранее. Чтобы использовать перо, выберите его дескриптор в контекст устройства метафайла, указав hmfDC при вызове функции SelectObject(). Выбор объекта в контекст устройства генерирует в метафайле команду повторного создания этого объекта во время воспроизведения. После выбора пера или иного объекта можно вызвать графическую функцию, например Rectangle(), снова указав дескриптор метафайла.

Контексты устройства метафайла не имеют предварительно определенных атрибутов. Они используют любой атрибут, существующий для окна, в котором создается метафайл. В большинстве случаев вы будете создавать перо, кисть или иные объекты, выбирать их в контекст устройства метафайла, а затем вызывать GDI-функцию, чтобы рисовать этим объектом. Если вы не создавали и не выбирали объект в контекст устройства метафайла, прорисовка будет осуществляться с помощью любого объекта, ранее выбранного в контекст воспроизводящего окна.

Предупреждение. Передача выражений GDI-функциям, используемым при создании метафайлов, может работать не так, как ожидалось. Например, если в предыдущем программном фрагменте цвет пера определить как RGB(rc+10, gc+20, bc+30), выражения будут вычисляться при создании объекта. Во время воспроизведения метафайла используются эти же самые константы, а исходные выражения повторно не вычисляются. Как правило, для любой функции f() оператор f(expr) запоминает в метафайле результат выражения expr. Выражение не вычисляется повторно во время воспроизведения метафайла.

После создания объектов и вызовов GDI-функций для прорисовки на контексте устройства метафайла обратитесь к функции CloseMetaFile() и запомните полученный дескриптор в переменной типа HMETAFILE. Этот шаг информирует Windows о том, что программа завершила подборку GDI-функций в метафайле.

```
HMETAFILE hmf;
hmf = CloseMetaFile(hmfDC);
```

Замечание. Различайте дескриптор типа HMETAFILE и дескриптор контекста устройства метафайла (тип HDC). HDC-дескриптор используется только при создании метафайла. HMETAFILE-дескриптор используется для ссылки на метафайл, после того как последний был создан и закрыт.

Чтобы закрыть метафайл, передайте функции CloseMetaFile() дескриптор контекста устройства метафайла, полученный от CreateMetaFile(). После этого вам больше не нужна переменная hmfDC — фактически *этот дескриптор более не является действительным и не должен использоваться.* Новый дескриптор, возвращаемый функцией CloseMetaFile(), идентифицирует законченный метафайл, сформированный в памяти (в нашем примере он сохранен также на диске). Для воспроизведения метафайла и просмотра изображения в окне, передайте этот дескриптор функции PlayMetaFile(), как правило, в ответ на сообщение WM_PAINT:

```
PlayMetaFile(hDC, hmf);
```

Дескриптор hDC, обычно получаемый от функции GetDC(), идентифицирует оконный контекст устройства. Дескриптор hmf идентифицирует метафайл в памяти. Вызов функции PlayMetaFile() заставляет Windows выполнять каждую запись метафайла так, как будто программа сама вызывает соответствующие GDI-функции.

Воспроизведение метафайла может влиять на атрибуты текущего контекста устройства. Для сохранения этих атрибутов было бы неплохо окружить PlayMetaFile() вызовами функций SaveDC() и RestoreDC(), как показано ниже (аргумент -1 восстанавливает самый последний сохраненный набор атрибутов).

```
SaveDC(hDC);
PlayMetaFile();
RestoreDC(hDC, -1);
```

Это дает тот же самый эффект, что и повторный выбор по умолчанию пера, кисти и других объектов в контекст устройства. Объекты, созданные с помощью функции PlayMetaFile(), автоматически удаляются, т.е. вам не

нужно вызывать для этого функцию DeleteObjects(). Вы должны удалять объекты, построенные во время создания метафайла. Другими словами, если вы создаете перо и используете его в метафайле для прорисовки, необходимо удалить этот объект, как вы делаете это с контекстом устройства окна. Во время воспроизведения метафайла Windows создает и удаляет объекты автоматически.

Замечание. Метафайл можно целиком создавать в памяти и не сохранять его на диске. Для этого следует передать NULL функции CreateMetaFile(). При этом способе метафайл собирает множество GDI-функций в компактный пакет для многократного воспроизведения.

Финальный этап создания и использования метафайла — удаление его дескриптора, перед тем как программа закончится. Делайте это всегда следующим образом:

```
DeleteMetaFile(hmf);
```

Чтобы удалить метафайл, передайте его дескриптор, полученный от функции CloseMetaFile(), функции DeleteMetaFile(). Данный оператор освобождает выделенную метафайлу память и аннулирует его дескриптор. Не используйте переменную hmf после передачи ее функции DeleteMetaFile().

Предупреждение. *Метафайлы принадлежат GDI, а не программе и не* удаляются автоматически после ее завершения. Чтобы предотвратить утечку памяти, всегда удаляйте дескрипторы метафайлов, вызывая DeleteMetaFile(). Вопреки своему имени эта функция не стирает метафайл с диска, а удаляет лишь его копию в памяти. Для удаления файла на диске (что вы обязаны сделать, например для временного файла) вызовите OpenFile() (см. гл. 3) или используйте стандартную библиотечную функцию _unlink().

Чтобы взять метафайл, хранящийся на диске, вызовите функцию GetMetaFile(). Она возвратит дескриптор типа HMETAFILE.

```
HMETAFILE hmf;
hmf = GetMetaFile((LPSTR)"anyfile.wmf");
```

Можно также передать указатель на символьную строку, содержащую имя файла. Если полученный дескриптор не нулевой, передайте его вместе с дескриптором контекста устройства в функцию PlayMetaFile() для отображения команд метафайла:

```
PlayMetaFile(hDC, hmf);
```

Не забудьте удалить дескриптор метафайла после завершения работы с ним.

```
DeleteMetaFile(hmf);
```

Совет. Для большего спокойствия всегда проверяйте существование метафайла *до* вызова функции GetMetaFile(). Если вы не запретили сообщение об отсутствии файла, выдаваемое по умолчанию (см. функцию SetErrorMode() в разделе "Другие полезные файловые функции" гл. 3), то в случае отсутствия искомого метафайла на диске функция GetMetaFile() выдаст диалоговую панель с просьбой вставить диск в дисковод A:, что может смутить пользователя. Избегайте этого сообщения, проверяя существование файла до вызова GetMetaFile(). Для этого используйте функции _lopen() или OpenFile(), как показано в гл. 3.

Другие метафайловые функции

Ниже дано еще несколько функций метафайла, которые могут оказаться полезными.

CopyMetaFile()

Копирует метафайл из памяти на диск. Windows объявляет данную функцию следующим образом:

```
HMETAFILE CopyMetaFile(HMETAFILE hmfSrc, LPCSTR lpszFile);
```

- hmfSrc — дескриптор, идентифицирующий область памяти, которая содержит подлежащий передаче на диск метафайл. Этот дескриптор обычно получают с помощью оператора hmfSrc = CreateMetaFile(NULL);
- lpszFile — литеральная ASCIIZ-строка или указатель на символьную строку с именем файла и маршрутом. Любой уже существующий файл с таким же именем будет перезаписан.

Совет. Если вы присвоите переменной lpszFile значение NULL, то функция CopyMetaFile() создаст в памяти копию метафайла, содержащую все команды первоисточника. Это может пригодиться для передачи метафайлов функции, которая как-либо изменяет их содержимое. В этом случае передавайте не оригинальный метафайл, а его копию, созданную с помощью CopyMetaFile().

GetMetaFileBits()

Как правило, вы будете использовать функцию CreateMetaFile() и функцию CopyMetaFile() для записи метафайла на диск. Но вы также можете обрабатывать метафайлы в двоичной форме, т.е. как блоки неструктурированных байтов. Это может понадобиться, например для присоединения структуры PMETAHEADER при создании размещаемого метафайла.

Чтобы получить дескриптор блока памяти, занимаемого метафайлом, вызовите функцию GetMetaFileBits() и запомните полученный дескриптор типа HGLOBAL для дальнейшего использования с файловыми функциями. Windows объявляет GetMetaFileBits() следующим образом:

```
HGLOBAL GetMetaFileBits(HMETAFILE hmf);
```

- hmf — дескриптор метафайла, тот же, что и возвращаемый функцией CloseMetaFile().

Предупреждение. После вызова функции GetMetaFileBits() первоначальный дескриптор метафайла hmf более недействителен и не должен использоваться. Для освобождения памяти, занимаемой метафайлом, вызывайте функцию GlobalFree(). Никогда не обращайтесь для этого к DeleteMetaFile().

SetMetaFileBits()

Эта функция преобразует блок памяти в метафайл, т.е. является "обратной" по отношению к GetMetaFileBits(). Во многих местах она документируется неправильно, что особенно заметно в файле оперативной помощи по Microsoft Windows SDK, где перепутаны типы параметров. Правильное объявление в windows.h выглядит следующим образом:

```
HMETAFILE SetMetaFileBits(HGLOBAL hMem);
```

- hMem — дескриптор области глобальной динамической памяти, возвращаемый обычно функцией GetMetaFileBits(). Однако может относиться к области памяти, содержащей метафайл, построенный нестандартными методами.

Замечание. После вызова функции SetMetaFileBits() метафайл можно удалить с помощью функции DeletemetaFile(), передав ей полученный дескриптор обычным образом. Однако не освобождайте память с помощью GlobalFree().

SetMetaFileBitsBetter()

Программы, которые поддерживают протокол встраивания и связывания объектов (OLE), должны вызывать функцию SetMetaFileBitsBetter(), а не SetMetaFileBits(). Обе эти функции объявляются похожим образом и выполняют идентичные задачи преобразования блока глобальной памяти в метафайл, возвращая дескриптор типа HMETAFILE. Однако данная функция назначает GDI владельцем метафайла, создавая, таким образом, постоянный метафайл, который остается действительным даже после завершения породившей его программы.

EnumMetaFile()

Для исчерпывающего управления воспроизведением метафайла следует вызвать функцию EnumMetaFile(), которая передает GDI-записи метафайла одну за одной некоторой функции-обработчику (callback) в вашей программе. Таким образом, вы получаете полный контроль над воспроизведением метафайла. Например, можно использовать функцию EnumMetaFile() для воспроизведения лишь части файла, для организации паузы в середине воспроизведения, для отладки метафайла в замедленном темпе или для генерации отчетов о его содержимом. EnumMetaFile() объявляется следующим образом:

```
BOOL EnumMetaFile(HDC hdc, HMETAFILE hmf,
  MFENUMPROC mfEnumProc, LPARAM lParam);
```

- hdc — дескриптор контекста устройства, обычно связанный с окном, в котором должен прорисовываться метафайл.
- hmf — дескриптор метафайла, записи которого будут перебираться. В Windows 3.0 тип этого параметра указан неправильно как HLOCAL. Эта ошибка была исправлена в Windows 3.1. Во избежание ошибки компилятора под Windows 3.0 приведите этот параметр к типу HLOCAL.
- mfEnumProc — адрес экземпляра процедуры для функции-обработчика в вашей программе. Этот адрес можно получить с помощью вызова MakeProcInstance().
- lParam — 32-разрядное число или указатель, передаваемый функции-обработчику в вашей программе. Это число можно использовать каким угодно образом. Например, передать флажок или аргумент функции-обработчику или передать указатель на структуру. Если вам не нужно ничего передавать функции-обработчику, установите lParam равным нулю.

Функция-обработчик, идентифицируемая параметром mfEnumProc должна объявляться следующим образом:

```
int EnumMetaFileProc(HDC hdc, HANDLETABLE FAR* lpht,
  METARECORD FAR* lpmr, int cObj, LPARAM lParam);
```

- hdc — контекст устройства, переданный функции EnumMetaFile(). Вы можете передавать этот контекстный дескриптор любой GDI-функции или использовать его для воспроизведения отдельных записей метафайла с помощью функции PlayMetaFileRecord().
- lpht — указатель на таблицу дескрипторов объектов, содержащую кисти, перья и т.д., используемую различными GDI-функциями метафайла.
- lpmr — указатель на некоторую запись метафайла. Вы можете модифицировать эту запись и, таким образом, изменять его содержимое.
- cObj — число кистей, перьев и других объектов, используемых данной записью метафайла.
- lParam — необязательный параметр, представляющий собой 32-разрядное число или указатель, передаваемый функции EnumMetaFile().

PlayMetaFileRecord()

Во время выполнения функции EnumMetaFile() программа-обработчик может обращаться к функции PlayMetaFileRecord() для воспроизведения отдельной записи метафайла. Данная функция объявляется в Windows следующим образом:

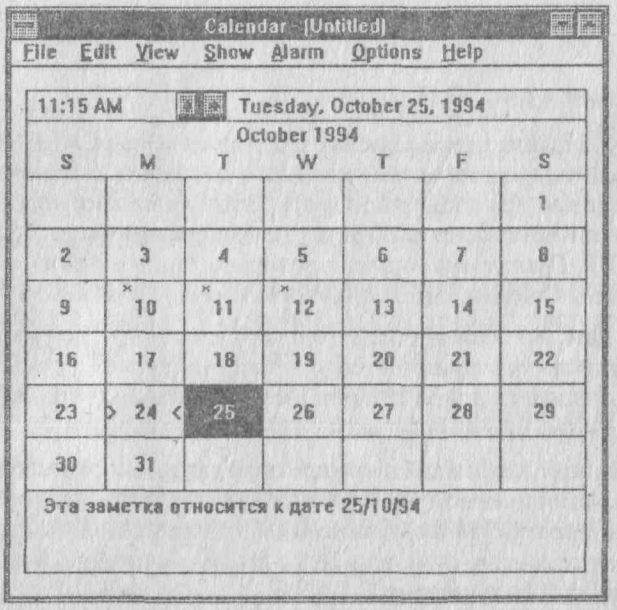
```
void PlayMetaFileRecord(HDC hdc, HANDLETABLE FAR* lpht,
  METARECORD FAR* lpmr, UINT cHandles);
```

- hdc — см. аналогичный параметр функции-обработчика.
- lpht — см. аналогичный параметр функции-обработчика.
- lpmr — см. аналогичный параметр функции-обработчика.
- cHandles — см. параметр cObj функции-обработчика.

Глава 9
ФАЙЛЫ КАЛЕНДАРЯ (.CAL)

Одним из первых приложений Windows, с которым знакомятся пользователи, является программа Calendar (см. рис 9.1.). Поскольку это приложение весьма широко используется, всякая программа, имеющее дело с датой и временем, должна распознавать формат CAL-файлов, где программа-календарь хранит заметки, маркеры, распорядок делового дня и установки будильника.

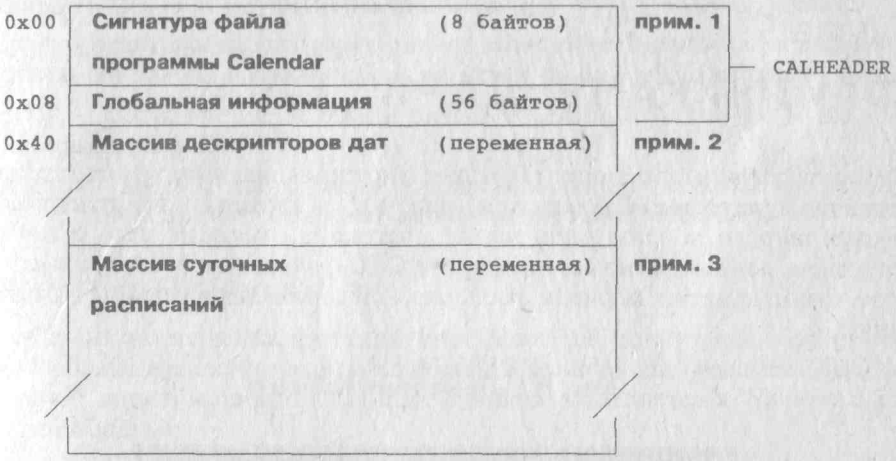
Рис. 9.1. Программа Calendar



Формат файла

Файл программы Calendar состоит из нескольких секций. Некоторые из них содержат глобальные данные, относящиеся ко всему файлу, другие — даты и суточные расписания. Общий формат CAL-файла показан на рис. 9.2.

Рис. 9.2. Формат файла программы Calendar



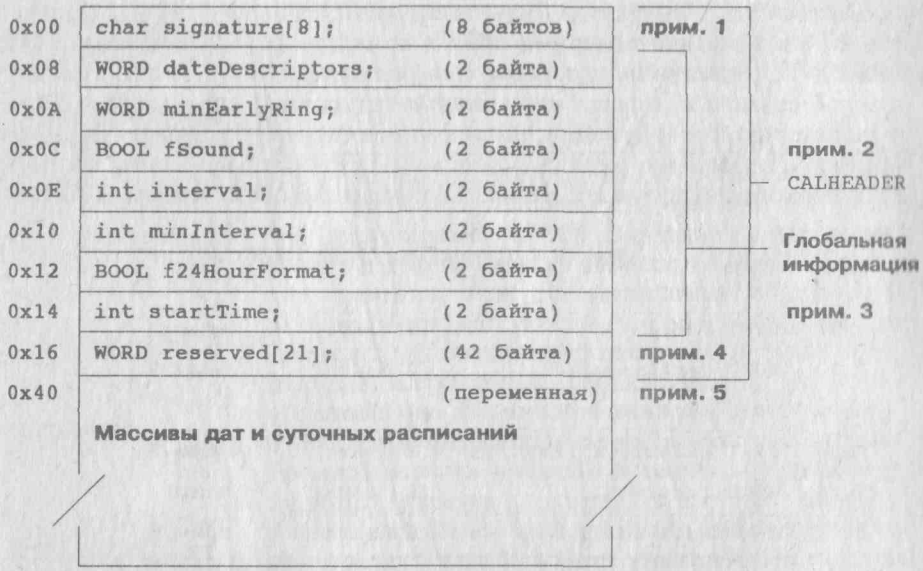
Примечания к рис. 9.2

1. Сигнатура файла равна цепочке прописных букв CALENDAR, зашифрованной посредством побуквенного сложения с цепочкой строчных букв radnelac (calendar наоборот). Этот довольно необычный прием шифровки порождает байтовую последовательность B5 A2 B0 B3 B3 B0 A2 B5. Программа может проверить, принадлежит ли некий файл программе Calendar, анализируя эти восемь байтов. См. также рис. 9.3.
2. Каждая дата, имеющая не менее одного связанного с ней маркера, заметки, назначенного мероприятия или сигнала будильника, описывается записями, хранящимися в этой секции. Вместе с *массивом дескрипторов дат* этот массив формирует каталог активных календарных дат. См. также рис. 9.4.
3. Информация, связанная с конкретной датой, запоминается в суточной записи, выровненной на 64-байтовую границу относительно начала файла. *Массив суточных расписаний* содержит для каждой даты текст заметки плюс ноль, одну или более строк расписания, а также установки будильника. См. также рис. 9.5 и 9.6.

Заголовок файла программы Calendar

Файл программы Calendar начинается 64-байтовым заголовком, содержащим уникальную сигнатуру файла и некоторую глобальную информацию о файле. На рис. 9.3 показана структура заголовка календарного файла, детализирующая первые две секции на рис. 9.2.

Рис. 9.3. Заголовок файла программы Calendar



Примечания к рис. 9.3

1. Это символьная строка, не имеющая оконечного нулевого байта. См. листинг 9.1 в разделе "Интерфейс с языком Си" настоящей главы для более подробной информации о сигнатуре.
2. Полная структура CALHEADER занимает ровно 64 байта. Вы можете читать и записывать ее поэлементно или целым блоком, как показано здесь.
3. Храните эти значения в глобальных переменных. Они понадобятся вам для обработки другой информации из календарного файла.
4. Эта 42-байтовая зарезервированная область не используется и должна содержать только нули. Если заголовок читается поэлементно, эту

область, служащую для выравнивания последующих данных на 64-байтовую границу, можно пропустить.

5. Вслед за заголовком CAL-файла идет секция переменной длины, содержащая даты и суточные расписания. См. рис. 9.4, 9.5 и 9.6, а также листинги 9.2, 9.3 и 9.4.

Массив дескрипторов дат

Массив дескрипторов дат начинается со смещения 0x40 в календарном файле. Этот массив содержит одну или несколько 12-байтовых структур DATEINFO, каждая из которых описывает одну активную дату, т.е. дату, с которой связана некоторая введенная пользователем информация. Даты, не имеющие такой информации, не запоминаются, что позволяет CAL-файлам сохранять небольшие размеры, даже если суточные расписания составлены для широкого диапазона дат. На рис. 9.4 показан формат структуры DATEINFO.

Рис. 9.4. Структура DATEINFO

0x00	UINT date;	(2 байта)	прим. 1
0x02	WORD markSymbol;	(2 байта)	прим. 2
0x04	WORD numAlarms;	(2 байта)	
0x06	UINT fileBlockOffset;	(2 байта)	прим. 3
0x08	WORD reserved1;	(2 байта)	прим. 4
0x0A	int reserved2;	(2 байта)	прим. 5

DATEINFO

Примечания к рис. 9.4

1. Это поле содержит число дней, прошедших с 1 января 1980 года до даты, указанной в данном элементе массива дескрипторов дат. Другими словами, нулевое значение этого поля соответствует 1/01/80. Первая структура массива располагается по смещения 0x40 относительно начала файла. См. также рис. 9.2.
2. Даты могут маркироваться одним или несколькими ASCII-символами, закодированными в этом слове как флажки. Символы не имеют заранее определенного значения. Пользователь может помечать ими любые даты. См. листинг 9.2 для информации о флажках этого поля.
3. Поле fileBlockOffset определяет относительное местоположение в файле информации о дне и назначенных на него мероприятиях, связанных с данной структурой DATEINFO. Его значение равно числу 64-байтовых

блоков, предшествующих этой информации. Так, если fileBlockOffset равно 8, суточное расписание на данную дату находится по смещению 8*64=512 байтов относительно начала файла. Значение 0xFFFF указывает на отсутствие заметки, мероприятий и установок будильника, связанных с этой датой.

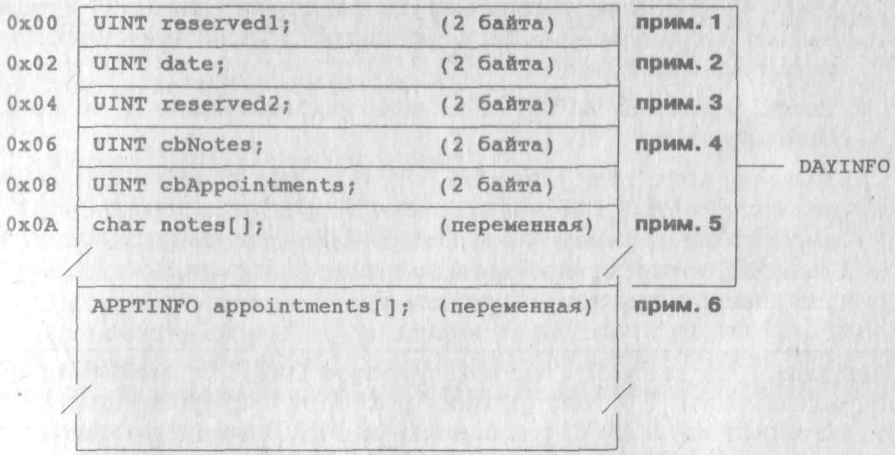
4. Всегда равняется 0xFFFF. Это поле зарезервировано и не должно использоваться.
5. Значение этого поля равно 0xFFFE, если дата не имеет связанной с ней пользовательской информации. В противном случае это поле имеет то же значение, что и fileBlockOffset или fileBlockOffset+0x80. Данное поле зарезервировано и не должно использоваться. Его точное значение, по-видимому, не важно.

Замечание. Поля reserved1 и reserved2 структуры DATEINFO весьма скудно документированы. При тестировании реальных CAL-файлов оказалось, что значение поля reserved1 всегда равно 0xFFFF. Значение поля reserved2 иногда равно 0xFFFE (если с датой не связана никакая информация), но может принимать другие значения, если дополнительная информация присутствует. Эти факты позволяют предположить, что программа Calendar использует reserved1 и reserved2 для своих внутренних целей, запоминая, вероятно, результат умножения переменной fileBlockOffset на 64, чтобы вычислить относительное смещение связанной с данной структурой информации о распорядке дня. Все вышеизложенное достаточно умозрительно, и, вероятно, можно использовать reserved1 и reserved2 по вашему усмотрению. Однако, видимо, небезопасно сохранять частные данные в этих полях на диске, т.к. они официально зарезервированы для возможного использования в будущих версиях. Во избежание противоречий с существующими CAL-файлами, устанавливайте в поле reserved1 значение 0xFFFF, а в поле reserved2 — 0xFFFE.

Массив мероприятий суточного расписания

Каждая структура DATEINFO может указывать на структуру DAYINFO, которая содержит заметку и распорядок дня, связанные с этой датой. Записи типа DAYINFO выровнены на 64-байтовую границу относительно начала файла. Формат структуры DAYINFO показан на рис. 9.5.

Рис 9.5. Структура DAYINFO



Примечания к рис 9.5

1. Это поле зарезервировано и всегда равно 0x0000. Смещение 0x00 этого поля указано относительно начала структуры, а не файла. Смещение внутри файла вычисляется посредством умножения поля fileBlockOffset структуры DATEINFO на 64, как указано в примечаниях к рис 9.4.
2. Это поле содержит число дней, прошедших с 1 января 1980 года до данной даты. Всегда должно быть таким же, как и поле date соответствующей структуры DATEINFO.
3. Запишите в это зарезервированное поле число 0x0001.
4. Префикс "cb" в именах этого и следующего полей означает "счетчик байтов" (count of bytes). Поле cbNotes определяет байтовую длину строки, содержащей текст заметки, включая оконечный нуль. Поле cbAppointments определяет число записей типа APPTINFO, следующих за структурой DAYINFO.
5. ASCIIZ-строка, содержащая текст заметки, связанный с данной датой. Текст заметки отображается в нижней части окна программы Calendar (см. рис. 9.1) при выборе этой даты. Поле cbNotes содержит длину строки в байтах, включая оконечный нуль. Если cbNotes содержит нуль, текст заметки отсутствует.
6. Этот массив содержит ноль, одно или несколько мероприятий, запланированных на данную дату. Мероприятия могут назначаться на любое

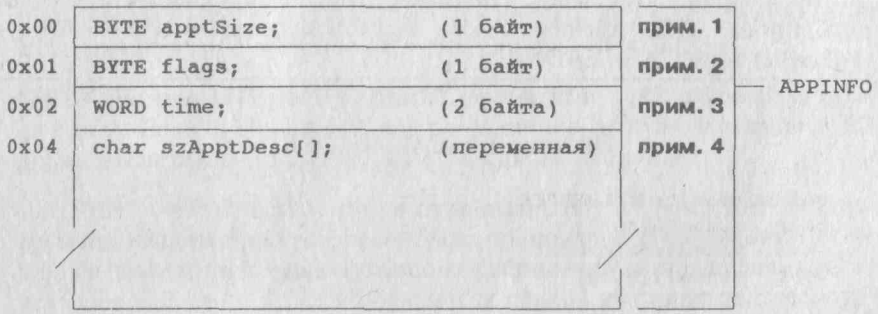
время суток, и, следовательно, этот массив потенциально может состоять из 1440 элементов — по одному на каждую минуту суток. Записи типа APPTINFO идут сразу же вслед за полем notes (или за полем cbAppointments, если с датой не связан текст заметки) структуры DAYINFO.

APPTINFO

Вслед за структурой DAYINFO располагается массив записей типа APPTINFO с расписанием на данные сутки (рис. 9.6).

Предупреждение. Дни, на которые не назначено ни одного мероприятия (неплохо было бы иметь таких дней побольше?!), не сопровождаются записями типа APPTINFO. По этой причине массив запланированных на день мероприятий не включен в структуру DAYINFO, как это сделано в некоторых публикациях по формату календарного файла.

Рис. 9.6. Структура APPTINFO



Примечания к рис. 9.6

1. Значение этого поля определяет длину данной структуры в байтах. Ограничивает длину строки szApptDesc до 255 байтов, включая оконечный нуль.
2. Если это поле нулевое, данное мероприятие не сопровождается установкой будильника. Если его значение равно 0x01, сигнал будильника устанавливается на время назначения мероприятия. Если это поле содержит число 0x02, сигнал будильника установлен на иное время (например, на десять минут ранее назначенного срока).

3. Время мероприятия, закодированное как количество минут, прошедшее с полуночи.
4. Эта ASCIIZ-строка содержит текстовое описание мероприятия. Если мероприятие отмечено в календаре только установкой будильника и не содержит никакого текстового описания, эта строка отсутствует.

Интерфейс с языком Си

В следующих разделах перечисляются Си-структуры, которые соответствуют структурным диаграммам на рис. 9.2—9.6.

CALHEADER

Календарный файл начинается структурой CALHEADER, показанной в листинге 9.1. Эта структура занимает ровно 64 байта.

> **Совет.** Вы можете попробовать изобрести более простые структуры, чем те, которые приводятся здесь. Например, без полей signature и reserved. Однако проще всего прочитать весь заголовок целиком в 64-байтовую переменную типа CALHEADER.

Листинг 9.1. CALHEADER

```
typedef struct tagCALHEADER {
    char  signature[8];
    WORD  dateDescriptors;
    WORD  minEarlyRing;
    BOOL  fSound;
    int   interval;
    int   minInterval;
    BOOL  f24HourFormat;
    int   startTime;
    WORD  reserved[21];
} CALHEADER;
```

- signature — восьмибайтовая последовательность B5 A2 B0 B3 B3 B0 A2 B5. При любом другом значении этого поля файл испорчен или не является календарным. Эта последовательность получается посредством суммирования цепочки прописных букв CALENDAR и цепочки строчных букв radnelac (calendar наоборот). Другими словами, сумма ASCII-кодов C и r равна 0xB5, A+a — 0xA2 и т.д.

- dateDescriptors — количество дескрипторов дат в файле. Дескрипторы типа DATEINFO следуют непосредственно за заголовком календаря, начиная с байта 0x40 относительно начала файла.
- minEarlyRing — количество минут, на которое звонок будильника опережает время назначения мероприятия. Устанавливается в программе Calendar с помощью команд Alarm | Controls...
- fSound — Если 0 (FALSE), звуковой сигнал подавляется. Если 1 (TRUE), сигнал будильника слышен. Также устанавливается с помощью команд Alarm | Controls...
- interval — определяет нормальный временной интервал между двумя мероприятиями суточного расписания для каждого дня. Принимает значения 0 (15-минутный интервал), 1 (30-минутный интервал) или 2 (30-минутный интервал). Это поле используется (вместе с полем minInterval, описанного далее) для отображения времен суточного расписания, на которые не назначены никакие мероприятия. Назначенные мероприятия определяются своим собственным временем и не ограничены действующим временным интервалом. Устанавливается с помощью команд Options | Day Settings...
- minInterval — то же самое, что и interval, но выражено в минутах. Устанавливается с помощью команд Options | Day Settings...
- f24HourFormat — Если 0 (FALSE), время дано в 12-часовом утро/вечер формате. Если 1 или любое другое отличное от нуля значение (TRUE), время представлено в 24-часовом "военном" формате.
- startTime — количество минут относительно полуночи для начального времени ежедневника. Если не было специального назначения на более раннее время, то строка суточного расписания с определяемым полем startTime временем будет отображаться первой на странице ежедневника.
- reserved — Не используется. Должно содержать только нули.

Замечание. Поле reserved фактически не принадлежит структуре CALHEADER, хотя так удобнее ее показывать. Эта 42-байтовая зарезервированная область служит просто для выравнивания следующего далее массива дескрипторов дат на 64-байтовую границу. Зарезервированные байты сидят между этим массивом и заголовком файла. Вероятно, будет лучше обнулить все байты поля reserved. Если вы попытаетесь запомнить здесь какие-либо частные данные, ваш файл может оказаться несовместимым с программой Calendar.

DATEINFO

Вслед за структурой CALHEADER идут одна или несколько записей типа DATEINFO (листинг 9.2). Каждой активной дате соответствует один элемент типа DATEINFO. Под активной датой подразумевается та, с которой связаны маркер или текст заметки, или не менее одной установки будильника, или непустой распорядок дня. Все вместе эти записи образуют каталог дат в файле и называются дескрипторами даты.

Листинг 9.2. DATEINFO

```
typedef struct tagDATEINFO {
    UINT  date;
    WORD  markSymbol;
    WORD  numAlarms;
    UINT  fileBlockOffset;
    WORD  reserved1;
    int   reserved2;
} DATEINFO;
```

- date — количество дней, прошедшее от 1/01/80 до определяемой даты.
- markSymbol — одна или несколько флаговых констант из табл. 9.1. Объединяет несколько флажков с помощью логического ИЛИ. Эти маркеры помечают определенные даты в режиме просмотра месячного календаря (команды View | Month). К сожалению, маркеры представляются ASCII-символами, а не пиктограммами, что является очевидным недостатком программы.

Замечание. Константы табл. 9.1 не объявлены в windows.h. Используйте их литеральные значения, показанные в шестнадцатеричной форме и в скобках в десятичной системе, или определите их с помощью оператора #define в программном модуле или в заголовке файла. В колонке *Номер* показаны номера маркирующих символов в программе Calendar (команды Options | Mark).

Табл. 9.1.

Константы календарных маркеров

Константа	Значение	Символ	Номер
CMARK_NONE	0x0000 (0)	Нет	Нет
CMARK_BOX	0x0080 (128)	Прямоугольник	1
CMARK_PAREN	0x0100 (256)	Скобки	2
CMARK_CIRCLE	0x0200 (512)	Маленькое о или точка	3
CMARK_CROSS	0x0400 (1024)	Маленькое x	4
CMARK_UNDERSCORE	0x0800 (2048)	Подчеркивание	5

- numAlarms — общее число установок будильника в суточном расписании. Равно нулю, если таких установок не было.
- fileBlockOffset — имеет значение 0xFFFF, если с датой не связаны тексты заметок, мероприятия и установки будильника. В противном случае представляет собой число 64-байтовых блоков, предшествующих ассоциированным с данным дескриптором даты структурам DAYINFO и APPTINFO. Байтовое смещение этой информации в файле вычисляется посредством умножения fileBlockOffset на 64.
- reserved1 — не документировано и не используется. Всегда равняется 0xFFFF.
- reserved2 — не документировано и не используется.

Замечание. По словам Microsoft, значения переменных reserved1 и reserved2 должны равняться 0xFFF. Очевидно, что это трехзначное шестнадцатеричное число является некорректным. При тестировании реальных CAL-файлов оказалось, что значение поля reserved1 всегда равно 0xFFFF, а изменение его на 0x0000 не имело никакого эффекта. Аналогично поле reserved2 обычно содержит 0xFFFE, но ввод других чисел в это поле снова не оказал сколь-нибудь заметного влияния. Таким образом, можно с большой долей вероятности предположить, что эти два поля используются программой Calendar для своих нужд.

DAYINFO

С любой датой, имеющей текст заметки, непустое расписание или установки будильника, связана структура DAYINFO, показанная в листинге 9.3. Выражение fileBlockOffset*64 (это поле принадлежит структуре DATEINFO) вычисляет смещение в файле записи типа DAYINFO, относящейся к заданной дате.

Замечание. Структура DAYINFO имеет переменную длину.

Листинг 9.3. DAYINFO

```
typedef struct tagDAYINFO {
    UINT reserved1;
    UINT date;
    UINT reserved2;
    UINT cbNotes;
    UINT cbAppointments;
    char notes[1];
} DAYINFO;
```

- reserved1 — не документировано и не используется, должно содержать нуль.
- date — содержит число дней, прошедших с 1 января 1980 года до данной даты. Всегда должно быть таким же, как и поле date соответствующей структуры DATEINFO.

Совет. Для проверки целостности календарных файлов сравнивайте поле date структуры DAYINFO с аналогичным полем в связанной с ней записи типа DATEINFO. В случае несоответствия значений этих двух полей файл может оказаться испорченным.

- reserved2 — не документировано и не используется, должно содержать 0x0001.
- cbNotes — определяет байтовую длину строки, содержащей текст заметок, включая оконечный нуль. Содержит 0x0000, если таковые отсутствуют.
- cbAppointments — определяет размер записей типа APPTINFO, следующих за структурой DAYINFO. Содержит 0x0000, если на данный день не назначено ни одного мероприятия и нет установок будильника.
- notes — ASCIIZ-строка, содержащая текст заметки, связанный с данной датой. Каждая дата в календарном файле может иметь одну и только одну заметку, текст которой отображается в нижней части окна программы Calendar (см. рис. 9.1). Размер этого символьного массива, т.е. длину символьной строки в байтах, включая оконечный нуль, определяет поле cbNotes.

Предупреждение. Если cbNotes содержит нуль, массив notes не существует. В этом случае *программа никоим образом не должна обращаться к массиву notes.*

APPTINFO

Последняя структура в календарном файле содержит информацию о о запланированных мероприятиях и установках будильника. Ноль, одна или более структур APPTINFO (см. листинг 9.4) следуют непосредственно за структурой DAYINFO, описанной в предыдущем разделе. Общий размер в байтах всех записей типа APPTINFO, относящихся к определенной дате, содержится в поле cbAppointments структуры DAYINFO.

Замечание. Структура APPTINFO имеет переменную длину.

Листинг 9.4. APPTINFO

```
typedef struct tagAPPTINFO {
    BYTE  apptSize;
    BYTE  flags;
    WORD  time;
    char  szApptDesc[1];
} APPTINFO;
```

- apptSize — длина данной структуры в байтах.
- flags — установки будильника. Имеет следующие значения: 0 (нет сигнала), 1 (сигнал будильника устанавливается на время, хранящееся в поле time данной структуры, или 2 (сигнал будильника установлен на иное время, например, на десять минут ранее назначенного срока).
- time — время мероприятия, закодированное как количество минут, прошедшее с полуночи. Мероприятие и сигнал будильника могут быть назначены на любую минуту суток.
- szApptDesc — ASCIIZ-строка, содержащая текстовое описание запланированного мероприятия.

Предупреждение. Благодаря тому, что длина поля apptSize равна одному байту, суммарный размер записей типа APPTINFO ограничен 255 байтами. Это означает, что строка szApptDesc никогда не должна превышать 255 символов, включая оконечный нулевой байт.

Глава 10

ФАЙЛЫ КАРТОТЕКИ (.CRD)

Программа Windows Cardfile может хранить организованные в виде картотеки записи, содержащие текст или графику (см. рис. 10.1). Несмотря на недостаток интеллектуальных возможностей, эта программа полезна для управления простой базой данных. Поскольку Cardfile может еще и набирать телефонные номера по модему, многие пользователи Windows хранят базы данных имен и адресов в картотеках. Карточки могут также содержать графику или смесь графики и текста, как показывается далее.

Формат файла

Формат файла картотеки, приведенный в этой главе, совместим с Windows 3.0 и 3.1. В настоящее время доступны только структуры данных версии 3.0, и рассматриваемый здесь формат файла может быть изменен в будущем. Имейте это в виду, когда будете читать данную главу.

Заголовок

Картотечный файл, имя которого обычно имеет расширение CRD, начинается трехбуквенной сигнатурой. Следующие два байта показывают, сколько карточек хранится в файле. На рис. 10.2 эти элементы показаны в виде заголовочной структуры CARDHEADER.

Рис. 10.1. Программа Cardfile

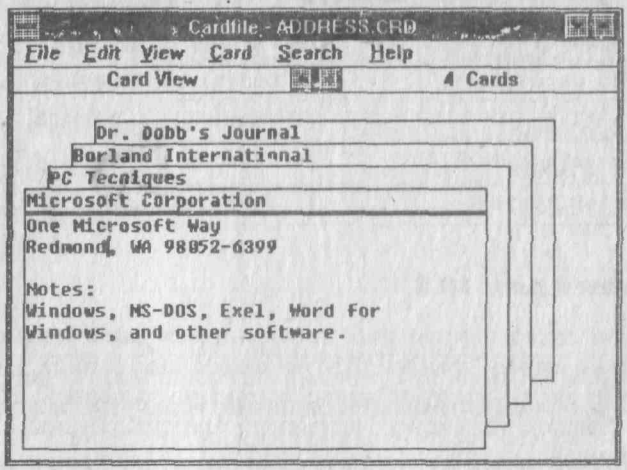


Рис. 10.2. Заголовок картотеки

0x00	char signature[3];	(3 байта)	прим. 1	CARDHEADER
0x03	WORD numCards;	(2 байта)	прим. 2	

Примечания к рис. 10.2

1. Трехбуквенная строка MGC. Это символьная строка без нулевого байта на конце.

2. Определяет число карточек в файле. Максимальное значение равно 65535. Однако на практике картотеки содержат значительно меньше записей.

Индекс

Каждая карточка файла имеет связанный с ней индексный элемент. Вместе эти элементы формируют каталог записей файла. Первый индекс начинается со смещения 0x05. Каждый последующий 52-байтовый индекс идет сразу же вслед за предыдущим. На рис. 10.3 показан формат индексной структуры.

Рис. 10.3. Индекс картотеки

0x05	WORD reserved[3];	(6 байтов)	прим. 1
0x0B	DWORD cardOffset;	(4 байта)	прим. 2
0x0F	BYTE flag;	(1 байт)	прим. 3
0x10	char szIndexText[40];	(40 байтов)	прим. 4
0x38	char nullByte;	(1 байт)	

CARDINDEX

Примечания к рис. 10.3

1. Смещения этого и остальных полей индекса даны только для первой индексной записи файла. Другие индексные записи, если таковые вообще имеются, следуют непосредственно за первой. Первые шесть байтов каждого индексного элемента зарезервированы и должны содержать нули.
2. Значение этого поля представляет собой байтовое смещение связанной с данным индексом карточки относительно начала файла. Устанавливайте внутренний указатель файла на эту позицию, чтобы прочитать определенные карточки.
3. Хотя этот байт и назван флажком (flag) в официальной документации, он, по-видимому, ни на что не влияет и должен устанавливаться в нуль.
4. Текст индексного элемента. Эта 40-байтовая символьная строка отображается как заголовок карточки. Строка должна завершаться нулевым байтом. Поскольку следующий байт, названный пустым (nullByte), всегда равен нулю, szIndexText может содержать от 0 до 40 значащих символа.

Текстовая карточка

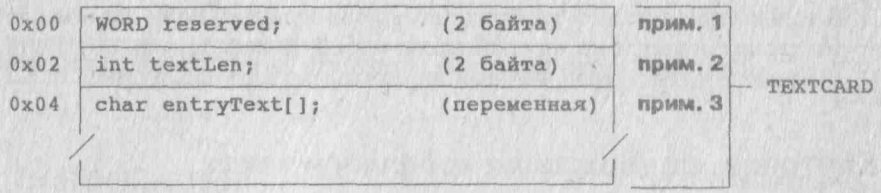
Каждая индексная запись картотеки указывает на карточку, содержащую текст или графику (или и то, и другое). На рис. 10.4 показан формат карточки, содержащей только текст.

Примечания к рис. 10.4

1. Смещения этого и остальных полей структуры даны относительно начала карточки. Если первое слово карточки равно нулю, она содержит только текст. В противном случае карточка содержит графические данные в одном из показанных на рис. 10.5 и 10.6 форматах.
2. Длина массива entryText в байтах.

3. Это символьная строка без оконечного нулевого байта. Может содержать коды возврата каретки и перевода строки.

Рис. 10.4. Текстовая карточка картотеки

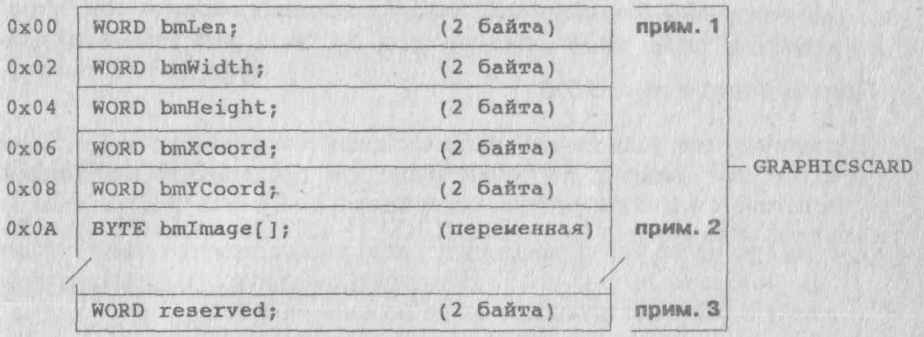


Предупреждение. При чтении массива entryText не пытайтесь найти завершающий нулевой байт, а при записи не добавляйте к нему оконечный нуль. Имейте также в виду, что массив entryText может содержать множество строк, разделенных символами возврата каретки и перевода строки. Последняя строка может не содержать этих управляющих кодов.

Графическая карточка

Карточка может также содержать растровую графику. В этом случае карточка имеет формат, показанный на рис. 10.5.

Рис. 10.5. Графическая карточка картотеки



Примечания к рис. 10.5

1. Смещения этого и остальных полей структуры даны относительно начала карточки. Если первое слово карточки равно нулю, она содер-

жит только текст (рис. 10.4). В противном случае карточка является графической и может также содержать текст (см. рис. 10.6).

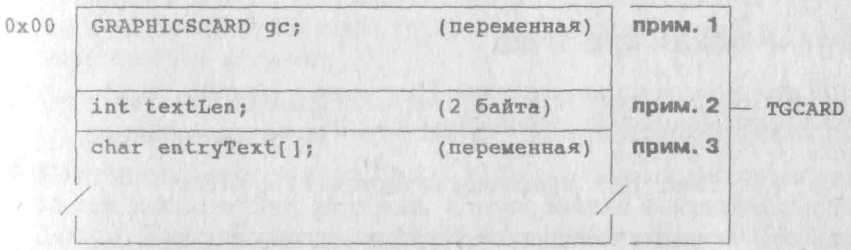
2. Длина этого поля дана в слове bmLen, расположенного в начале карточки.

3. В только графической карточке это слово нулевое. Если данное слово не равно нулю, оно представляет собой длину текстового поля в смешанной текстографической карточке, показанной на рис. 10.6.

Карточка, содержащая графику и текст

Карточка может содержат смесь текста и графики, например иллюстрацию с подписью. Этот тип карточек похож на только графический, но сопровождается дополнительной текстовой информацией, как показано на рис. 10.6.

Рис. 10.6. Карточка, содержащая графику и текст



Примечания к рис. 10.6

1. Включает все поля из рис. 10.5, исключая последний элемент reserved. Если этот элемент не равен нулю, он представляет собой длину текстового поля в карточке, содержащей смесь текста и графики.

2. Если это целое число равно нулю, карточка содержит только графику, как показано на рис. 10.5. Ненулевое значение представляет собой длину в байтах следующего далее массива entryText.

3. Содержит ASCII-текст для смешанной текстографической карточки. Отсутствует, если textLen равно нулю. Эта символьная строка содержит столько байтов, сколько указано в поле textLen, и не имеет оконечного нуля. Содержащийся текст может иметь несколько строк, разделенных управляющими кодами возврата каретки и перевода строки.

Интерфейс с языком Си

Следующие Си-структуры соответствуют структурным диаграммам на рис. 10.2—10.6.

Замечание. Картотечные структуры не объявлены в windows.h.

CARDHEADER

Картотечные файлы начинаются небольшим заголовком, идентифицирующим файл и показывающим число хранимых в нем карточек. Этот заголовок приводится в листинге 10.1 в виде структуры CARDHEADER.

Листинг 10.1. CARDHEADER

```
typedef struct tagCARDHEADER {
    char  signature[3];
    WORD  numCards;
} CARDHEADER;
```

- signature — содержит трехбуквенную цепочку MCG. Эта символьная строка не имеет оконечного нулевого байта.
- numCards — число карточек в файле. Все картотеки имеют хотя бы одну карточку, так что, если это поле нулевое, файл имеет нестандартный формат или испорчен.

CARDINDEX

Вслед за структурой CARDHEADER следуют одна или несколько элементов типа CARDINDEX, формирующих каталог карточек в файле. Структура CARDINDEX показана в листинге 10.2.

Совет. Чтобы отсортировать картотеку, вы должны переупорядочить ее индексы. Вам не нужно перемещать сами карточки, хранящиеся сразу же после индексного каталога. Порядок следования карточек может изменяться только программой Cardfile.

Листинг 10.2. CARDINDEX

```
typedef struct tagCARDINDEX {
    WORD reserved[3];
    DWORD cardOffset;
    BYTE flag;
    char szIndexText[40];
    char nullByte;
} CARDINDEX;
```

- reserved — не используется и не документировано. Должно содержать нули во всех байтах.
- cardOffset — байтовое смещение связанной с данным индексом карточки относительно начала файла. Устанавливайте внутренний указатель файла на эту позицию, чтобы прочитать определенную карточку.
- flag — не используется и не документировано. Должно содержать нуль.
- szIndexText — ASCII-текст, отображаемый в виде верхнего заголовка карточки (см. рис. 10.1). Эта строка может иметь, а может и не иметь оконечный нулевой байт.
- nullByte — гарантирует, что строка szIndexText будет всегда завершаться нулевым байтом. Всегда обнуляйте этот байт.

Замечание. Массив szIndexText может содержать от нуля до 40 символов. Чтобы получить наилучший результат, устанавливайте все неиспользуемые в этом массиве символы в 0x00. Если в тексте ровно 40 символов, то далее идущий байт nullByte гарантирует, что текстовая строка будет оформлена так, как надо, т.е. иметь завершающий нулевой байт.

TEXTCARD

Типовая карточка содержит одну или несколько текстовых строк, хранимых в виде структуры TEXTCARD, показанной в листинге 10.3.

Замечание. Структура TEXTCARD имеет переменную длину.

Листинг 10.3. TEXTCARD

```
typedef struct tagTEXTCARD {
    WORD reserved;
    int textLen;
    char entryText[1];
} TEXTCARD;
```

- reserved — не используется. Записывайте в это поле нуль.
- textLen — длина массива entryText в байтах.
- entryText — ASCII-текст данной карточки. Может содержать встроенные коды возврата каретки и перевода строки. Этот текст не является ASCIIZ-строкой и содержит ровно столько символов, сколько определено полем textLen.

Замечание. Карточки хранятся одна за другой без каких-либо промежуточных байтов. Они не выровнены по какому-либо адресу в файле. Важно понимать, что строки типа entryText не содержат оконечные нулевые байты, хотя во многих случаях из-за начальных нулевых байтов последующих текстовых карточек может показаться, что это не так. На самом деле — это чистое совпадение.

GRAPHICSCARD

Чисто графические карточки хранятся в виде структур GRAPHICSCARD (см. листинг 10.4).

Предупреждение. Структура GRAPHICSCARD имеет переменный размер. Поскольку элемент переменной длины bmImage не является последним в структуре, она не может использоваться в программах для непосредственного определения переменных, а имеет лишь информационный характер. Чтобы обойти это ограничение, читайте и записывайте элементы данной структуры индивидуально или объявите некоторую модифицированную структуру без поля reserved на конце.

Листинг 10.4. GRAPHICSCARD

```
typedef struct tagGRAPHICSCARD {
    WORD bmLen;
    WORD bmWidth;
    WORD bmHeight;
    WORD bmXCoord;
    WORD bmYCoord;
    BYTE bmImage[1];
    WORD reserved;
} GRAPHICSCARD;
```

- bmLen — длина байтового массива bmImage, содержащего растровое изображение, принадлежащее данной карточке.
- bmWidth — ширина растрового изображения в пикселах.
- bmHeight — высота растрового изображения в пикселах.

- bmXCoord — горизонтальная координата левого верхнего угла растрового изображения относительно левого края карточки.
- bmYCoord — вертикальная координата левого верхнего угла растрового изображения относительно верхнего края карточки.
- bmImage — монохромное растровое изображение на данной карточке.
- reserved — для чисто графических карточек равно нулю.

Совет. Для того чтобы определить, содержит ли карточка текст или графику, прочитайте первое слово ее внутреннего представления. Если оно нулевое, то карточка содержит только текст в форме структуры TEXTCARD (см. листинг 10.3). Если это слово не равно нулю, карточка может быть графической (листинг 10.4). Однако, если при этом поле reserved не равно нулю, то данная карточка также содержит текст в формате структуры TGCARD, который описывается ниже.

TGCARD

Последний формат, объединяющий в себе предыдущие два, используется для карточек, содержащий как текст, так и графику. Этот формат показан в листинге 10.5 в виде структуры TGCARD.

Предупреждение. Структура TGCARD, как и GRAPHICSCARD, имеет переменную длину. Поскольку ее элемент переменной длины bmImage объявлен не последним, эта структура не может использоваться в программах для непосредственного определения переменных, а имеет лишь информационный характер. Чтобы обойти это ограничение, читайте и записывайте элементы данной структуры индивидуально или объявите две новых структуры, заканчивающимися полями bmImage и entryText соответственно.

Листинг 10.5. TGCARD

```
typedef struct tagTGCARD {
    WORD bmLen;
    WORD bmWidth;
    WORD bmHeight;
    WORD bmXCoord;
    WORD bmYCoord;
    BYTE bmImage[1];
    int textLen;
    char entryText[1];
} TGCARD;
```

- bmLen—bmImage — то же самое, что и первые шесть элементов структуры GRAPHICSCARD.
- textLen — размерность идущего далее массива entryText в байтах. Это целое число соответствует полю reserved в структуре GRAPHICSCARD. Если textLen равно нулю, то карточка чисто графическая и элемент entryText отсутствует. Только когда textLen не равно нулю, карточка содержит смесь из текста и графики.
- entryText — то же самое, что и соответствующий элемент в структуре TEXTCARD, но располагается в другой относительной позиции. Текст в этом массиве может содержать управляющие коды возврата каретки и перевода строки. Данный массив не является строкой с нулевым оконечным байтом.

Глава 11

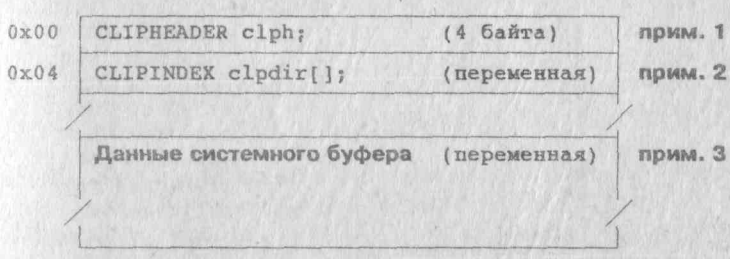
ФАЙЛЫ ПРОСМОТРЩИКА СИСТЕМНОГО БУФЕРА (.CLP)

Многие программисты и пользователи Windows путают системный буфер Windows (clipboard) с программой просмотра системного буфера (Clipboard Viewer). Это происходит, вероятно, потому, что системный буфер не осязаем, т.е. не есть нечто такое, что можно потрогать и сказать: "Это системный буфер". Фактически это протокол, в соответствии с которым программы разделяют информацию. Просмотрщик системного буфера является программой, поставляемой с Windows, позволяющей пользователям просматривать эту информацию, а также читать и записывать ее в CLP-файлы на диске, рассматриваемые в данной главе.

Формат файла

Из всех рассматриваемых в этой книге файловых форматов, формат файла системного буфера является простейшим. Его общая структура показана на рис. 11.1.

Рис. 11.1. Формат файла системного буфера



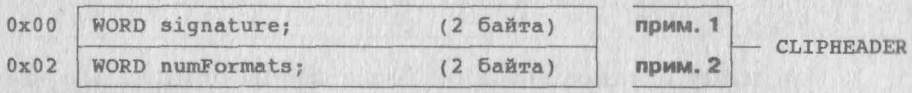
Примечания к рис. 11.1

1. Первые два байта заголовка равны 0x50 и 0x3C (0x3C50, если они рассматриваются как 16-разрядное слово). Следующие два байта определяют количество форматов данных, содержащихся в файле. См. также рис. 11.2 и листинг 11.1.
2. Каталог файла системного буфера содержит одну или несколько 89-байтовых структур CLIPINDEX. Каждая из этих индексных записей идентифицирует блок хранящихся в файле системного буфера данных. См. также рис. 11.3 и листинг 11.2.
3. Данные, хранящиеся в файле, могут быть в одном из нескольких форматов: текст, растровое изображение, метафайл и т.д. См. раздел этой главы "Форматы данных системного буфера".

Заголовок системного буфера

Файлы системного буфера начинаются небольшим двухсловным заголовком, который идентифицирует файл и определяет количество хранящихся в нем записей. Структура этого заголовка приведена на рис. 11.2.

Рис. 11.2. Заголовок файла системного буфера



Примечания к рис. 11.2

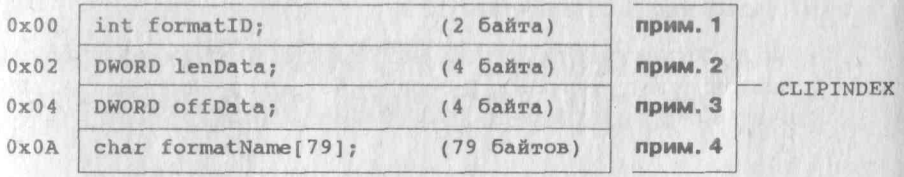
1. Это поле имеет значение 0x3C50 — сигнатура файла системного буфера. Единственным путем проверки того, содержит ли файл данные системного буфера, является анализ значения этого поля.
2. Определяет количество хранящихся в файле форматов данных, а также число индексов в каталоге системного буфера (см. рис. 11.3).

Индекс системного буфера

Вслед за заголовком идет как минимум одна индексная запись, формат которой показан на рис. 11.3. Множество индексных записей формирует каталог файла системного буфера.

Замечание. Часто файл системного буфера содержит более одной формы одних и тех же данных. Например, просмотрщик системного буфера обычно запоминает текстовую информацию двумя способами — первый раз с использованием символьного набора ANSI (Windows), а второй — с использованием символьного набора OEM (MS-DOS).

Рис. 11.3. Индекс системного буфера



Примечания к рис. 11.3

1. Байтовые смещения, показанные здесь, отсчитываются относительно начала структуры, а не относительно начала файла. Первый индексный элемент всегда имеет смещение 0x04. В файле со многими индексами индексные записи следуют друг за другом непосредственно.
2. Определяет длину в байтах блока данных, связанного с этим индексным элементом.
3. Определяет байтовое смещение блока данных относительно начала файла. Устанавливайте внутренний указатель файла на это значение, чтобы прочитать или записать определенный блок данных.
4. ASCIIZ-строка, определяющая тип данных, связанных с этим индексом.

Интерфейс с языком Си

В следующих далее листингах показан формат файла системного буфера в виде двух структур языка Си — CLIPHEADER и CLIPINDEX.

Замечание. Структуры, рассматриваемые в данном разделе, не объявлены в windows.h.

CLIPHEADER

Всякий CLP-файл начинается структурой CLIPHEADER (листинг 11.1) независимо от количества типов информации, содержащихся в файле. Эта структура может быть полезна для проверки того, что файл содержит системный буфер, и для определения количества хранимых записей.

Листинг 11.1. CLIPHEADER

```
typedef struct tagCLIPHEADER {
    WORD  signature;
    WORD  numFormats;
} CLIPHEADER;
```

- signature — всегда равно 0x3C50. Это поле можно также читать или записывать побайтно: 0x50 и 0x3C (байты в словах запоминаются в обратном порядке на компьютерах с процессорами фирмы Intel).
- numFormats — количество хранящихся в файле форматов данных. Столько же индексных элементов следуют непосредственно за заголовком. См. следующий раздел по структуре CLPINDEX.

Замечание. Можно считать это моим сугубо личным мнением, но сигнатура системного буфера весьма похожа на комбинацию ASCII-кодов трех букв: C(0x43), L(0x4C) и P(0x50). Вероятно, не случайно младшие четыре бита в кодах первых двух букв равны старшей и младшей тетрадам второго байта сигнатуры (0x3C), а ASCII-код буквы P равен первому байту сигнатуры (0x50).

CLIPINDEX

Каждая запись в файле системного буфера идентифицируется структурой CLIPINDEX, показанной в листинге 11.2. Первый индекс располагается в файле сразу же за структурой CLIPHEADER. Остальные индексы идут далее друг за другом без каких-либо промежутков.

Листинг 11.2. CLIPINDEX

```
typedef struct tagCLIPINDEX {
    int    formatID;
    DWORD  lenData;
    DWORD  offData;
    char   formatName[79];
} CLIPINDEX;
```

- formatID — тип данных, связанный с этим индексным элементом. Поместите в это поле одну из констант табл. 11.1 (см. раздел "Форматы

данных системного буфера") или значение из диапазонов CF_PRIVATE-FIRST–CF_PRIVATELAST или CF_GDIOBJFIRST–GDIOBJLAST.

- lenData — длина в байтах связанных с этим индексным элементом данных.
- offData — смещение относительно начала файла данных, связанных с этим индексным элементом. Устанавливайте внутренний указатель файла на это значение и читайте lenData байтов в буфер, чтобы прочитать определенный блок данных.
- formatName — ASCIIZ-строка, определяющая тип данных, связанных с этим индексом. Максимальная длина составляет 79 байтов, включая оконечный нуль.

Форматы данных системного буфера

Данные в системном буфере могут иметь много форматов, в том числе и специальные частные форматы, понятные лишь некоторым приложениям. В табл. 11.1 перечислены форматы данных системного буфера, идентифицируемые константами, объявленными в windows.h и начинающимися префиксом CF_.

Замечание. Константы CF_PRIVATEFIRST и CF_PRIVATELAST обозначают диапазон значений, который программы могут использовать для запоминания в системном буфере данных неописанных типов. Константы CF_GDIOBJFIRST и CF_GDIOBJLAST определяют подобный диапазон для объектов интерфейса графических устройств (GDI).

Табл. 11.1.

Константы форматов данных системного буфера

Константа	Значение	Назначение
CF_TEXT	0x0001	Одна или несколько строк завершающегося нулевым байтом ANSI-текста (называемым также символьным набором Windows), разделенных управляющими кодами возврата каретки и перевода строки. Данные, хранимые в формате CF_TEXT, обычно дублируются также в формате CF_OEMTEXT.
CF_BITMAP	0x0002	Растровое изображение в формате, зависящем от *устройства вывода*.
CF_METAFILEPICT	0x0003	Метафайловая картинка, сходная по формату с обычным метафайлом (см. гл. 8), но начинающаяся дополнительной структурой METAFILEPICT (см. листинг 11.3).

Табл. 11.1. (продолжение)

Константа	Значение	Назначение
CF_SYLK	0x0004	Данные символьной связи фирмы Microsoft (Microsoft Symbolic Link data). Подобны CF_TEXT, но предназначены для совместного использования со средствами программирования фирмы Microsoft. По всей видимости, вам вряд ли понадобится этот формат.
CF_DIF	0x0005	Формат обмена данными (Data Interchange Format) — текстовое представление табличной информации, первоначально разработанное фирмой Software Arts.
CF_TIFF	0x0006	TIFF-формат — Формат файла размеченного изображения. Обширная спецификация для представления графической информации в TIF-файлах.
CF_OEMTEXT	0x0007	Одна или несколько строк завершающегося нулевым байтом ASCII-текста, разделенных управляющими кодами возврата каретки и перевода строки. Совместим с текстовым режимом MS-DOS. Термин OEM означает *производитель оригинального оборудования* (original equipment manufacturer) и в данном случае относится к аппаратуре отображения, поддерживающей текстовые режимы.
CF_DIB	0x0008	Двухсекционный формат независимого от устройства растрового изображения, начинающийся структурой BITMAPINFO, описанной в гл. 4 (см. листинг 4.2) и заканчивающийся пикселами изображения.
CF_PALETTE	0x0009	Палитра цветов, обеспечивающая дополнительную цветовую информацию для формата CF_DIB.
CF_PENDATA	0x000A	Данные пера Windows.
CF_RIFF	0x000B	Формат файла обмена ресурсами — протокол, позволяющий запоминать блоки данных в разных форматах. Например, блок данных в формате RIFF может содержать растровое изображение, а также текстовую информацию. Как правило, используется с мультимедиа Windows для описания формы сигнала и прочих звуковых данных.
CF_WAVE	0x000C	Форма сигнала, описывающая звуковые данные для мультимедиа Windows.

Табл. 11.1. (продолжение)

Константа	Значение	Назначение
CF_OWNERDISPLAY	0x0080	Частные данные, которые должны быть отображены программами-владельцами этих данных. Как правило, не используются в файлах системного буфера, т.к. связь между данными и программами-владельцами не сохраняется в файле.
CF_DSPTEXT	0x0081	Текстовое представление частных данных.
CF_DSPBITMAP	0x0082	Растровое графическое представление частных данных.
CF_DSPMETAFILEPICT	0x0083	Метафайловое представление частных данных.
CF_PRIVATEFIRST	0x0200	Первое значение из допустимого диапазона, зарезервированного для частных типов данных.
CF_PRIVATELAST	0x02FF	Последнее значение из допустимого диапазона, зарезервированного для частных типов данных.
CF_GDIOBJFIRST	0x0300	Первое значение из допустимого диапазона, зарезервированного для частных GDI-объектов.
CF_GDIOBJLAST	0x03FF	Последнее значение из допустимого диапазона, зарезервированного для частных GDI-объектов.

Замечание. Значения констант из табл. 11.1 соответствуют форматам, *распознаваемым функцией SetClipboardData()*. См. справочник по программированию в среде Windows для более подробной информации об этой функции и о типах данных системного буфера.

Форматы большинства типов данных в файлах системного буфера согласуются с теми же типами данных в памяти. Например, текстовые данные хранятся просто как множество строк символов, разделенных управляющими кодами возврата каретки и перевода строки.

Тем не менее метафайл немного изменяется, когда помещается в системный буфер или записывается в файл системного буфера. Поскольку почти любое изображение, которое может быть нарисовано с помощью GDI-команд, точно представляется в метафайле, этот формат является близким к идеальному для передачи графической информации из одной программы в другую. К сожалению, в метафайлах недостает информации о размере и режиме отображения, необходимой для точного воспроизведения картинки. Программа просмотра системного буфера добавляет эту информацию, записывая в начало метафайла структуру METAFILEPICT, объявленной в windows.h и показанной в листинге 11.3.

Листинг 11.3. METAFILEPICT

```
typedef struct tagMETAFILEPICT {
  int mm;
  int xExt;
  int yExt;
METAFILE hMF;
} METAFILEPICT;
```

- mm — режим отображения, используемый для создания метафайла. Значение этого поля требуется для точного воспроизведения метафайла или для преобразования в другие единицы измерения. Режимы отображения выражаются константами MM_ANISOTROPIC, MM_LOMETRIC, MM_TEXT и другими константами из windows.h, начинающимися на MM_.
- xExt — ширина метафайла в единицах режима отображения, определяемого полем mm. Для режимов MM_ISOTROPIC и MM_ANISOTROPIC поле xExt представляет собой предполагаемую ширину в единицах MM_HIMETRIC. Может равняться нулю для режима MM_ANISOTROPIC, в котором программа свободна в выборе размера картинки. В режиме отображения MM_ISOTROPIC положительные значения полей xExt и yExt (следующее поле) определяют размеры картинки, одновременно предполагая коэффициент сжатия дисплейного изображения. При отрицательных xExt и yExt в режиме MM_ISOTROPIC их абсолютные значения представляют собой *только* коэффициент сжатия дисплейного изображения, а информация о размере отсутствует.
- yExt — высота метафайла в единицах режима отображения, определяемого полем mm. Все комментарии, касающиеся поля xExt, справедливы и для yExt.
- hMF — дескриптор метафайла в памяти. Не используется в метафайле, записанном в файл системного буфера.

Глава 12

ФАЙЛЫ РЕДАКТОРА WINDOWS WRITE (.WRI)

Одним из наиболее полезных приложений, поставляемых с Windows, является Windows Write. Хотя его возможности не так обширны, как у других текстовых процессоров (WordPerfect или Word for Windows), Write прекрасно подходит для компьютерной документации, писем и небольших рукописей.

Основное преимущество Write по сравнению с такими простыми текстовыми редакторами, как Windows Notepad, — возможность сохранять вместе с текстом информацию о его формате. Вы можете выбрать шрифты и прочие атрибуты для отображения текста на экране или для печати его на принтере в жирном и курсивном начертаниях, а также с подчеркиваниями. Кроме того, вы можете устанавливать поля для абзацев, центрировать абзацы (что особенно полезно для заголовков), изменять абзацные втяжки и устанавливать табуляции.

Чтобы поддерживать эти и другие возможности, необходима сложная структура файла. Даже если вы не планируете разработку программ для непосредственного чтения или записи WRI-файлов (обычное расширение имен файлов, подготовленных с помощью Write), вы лучше поймете приемы форматирования данных, изучив диаграммы этих файлов и соответствующие структуры языка Си в этой главе.

Формат файлов

Типичный WRI-файл может иметь более дюжины различных структур, таблиц, битовых флажков и прочих элементов. Чтобы понять, как это все взаимодействует, сначала просмотрите все представленные в этой главе структурные диаграммы, а затем изучайте их более подробно по мере продвижения по разделу "Интерфейс с языком Си".

WRHEADER

Файлы Windows Write начинаются с заголовка, который содержит некоторые сведения о файле. Формат этого заголовка показан на рис. 12.1.

Рис. 12.1. Заголовок WRI-файла

0x00	WORD wIdent;	(2 байта)	прим 1.	
0x02	WORD dty;	(2 байта)		
0x04	WORD wTool;	(2 байта)		
0x06	WORD reserved1[4];	(8 байтов)	прим. 2	
0x0E	DWORD fcMax	(4 байта)		
0x12	WORD pnPara	(2 байта)		прим. 3
0x14	WORD pnFntb	(2 байта)		WRHEADER
0x16	WORD pnSep	(2 байта)		
0x18	WORD pnSetb	(2 байта)		
0x1A	WORD pnPgtb	(2 байта)		
0x1C	WORD pnFfntb	(2 байта)		
0x1E	WORD reserved2[33]	(66 байтов)		
0x60	WORD pnMax	(2 байта)		
0x62	WORD reserved3[15]	(30 байтов)		

Примечания к рис. 12.1

1. Этим полем начинается всякий WRI-файл. Его значение должно быть равным 0xBE31. Если же оно равно 0xBE32, то файл содержит объекты OLE (протокол связывания и встраивания объектов).

2. Так много зарезервированных байтов в этой и других структурах предусмотрено для того, чтобы сделать WRI-файлы хотя бы частично совместимыми с DOC-файлами текстового процессора Word for DOS. Но, к сожалению, WRI- и DOC-файлы непосредственно не совместимы с документами Word for Windows.

3. См. листинг 12.1 для более подробной информации о структуре WRHEADER.

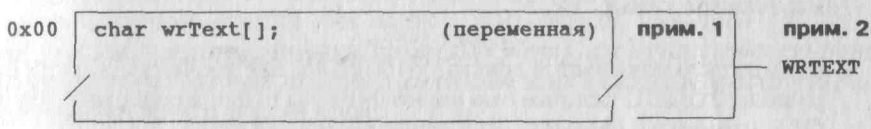
WRTEXT

Текст в WRI-файлах хранится в коде ASCII по абзацам, завершающимися кодами возврата каретки и перевода строки. Файлы могут также содержать управляющие коды прогона листа (ASCII-код 0x0C), принудительного конца страницы и табуляции (ASCII-код 0x09). За исключением этих немногочисленных управляющих кодов, Write запоминает информацию о форматировании отдельно, в отличие от некоторых текстовых процессоров, смешивающих эту информацию с текстом.

Совет. Преобразование WRI-файла в чисто текстовый несложно. Найдите лишь местоположение текстовых страниц (термин программы Write для обозначения 128-байтовых секторов диска) и прочитайте находящийся там ASCII-текст. Чтобы представить текст построчно, вам нужно будет расставить в соответствующие места коды возврата каретки и перевода строки, которыми отмечаются концы абзацев в документах, подготовленных редактором Write. См. подраздел "WRTEXT" в разделе "Интерфейс с языком Си" о способе поиска ASCII-текста в WRI-документах.

На рис. 12.2 в чисто иллюстративных целях приведен формат хранения текста редактора Write в виде структуры WRTEXT. Однако нет необходимости читать и записывать текстовую часть WRI-файлов именно в такой форме. Вы можете обрабатывать символы индивидуально или читать текст в буфер абзацами.

Рис. 12.2. Формат хранения текста



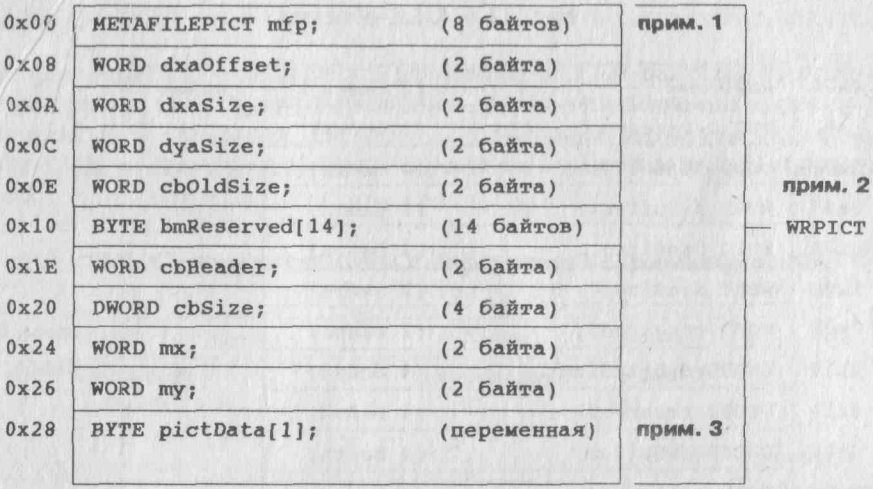
Примечания к рис. 12.2

1. Текст хранится в 128-байтовых секторах, называемых страницами. Первая страница текста расположена во втором секторе файла.
2. См. листинг 12.2 и сопутствующие комментарии для более подробной информации о способе хранения текста в WRI-файлах.

WRPICT

Картинки могут непосредственно запоминаться в WRI-файлах в формате метафайла, как показано на рис. 12.3.

Рис. 12.3. Формат хранения картинок



Примечания к рис. 12.3

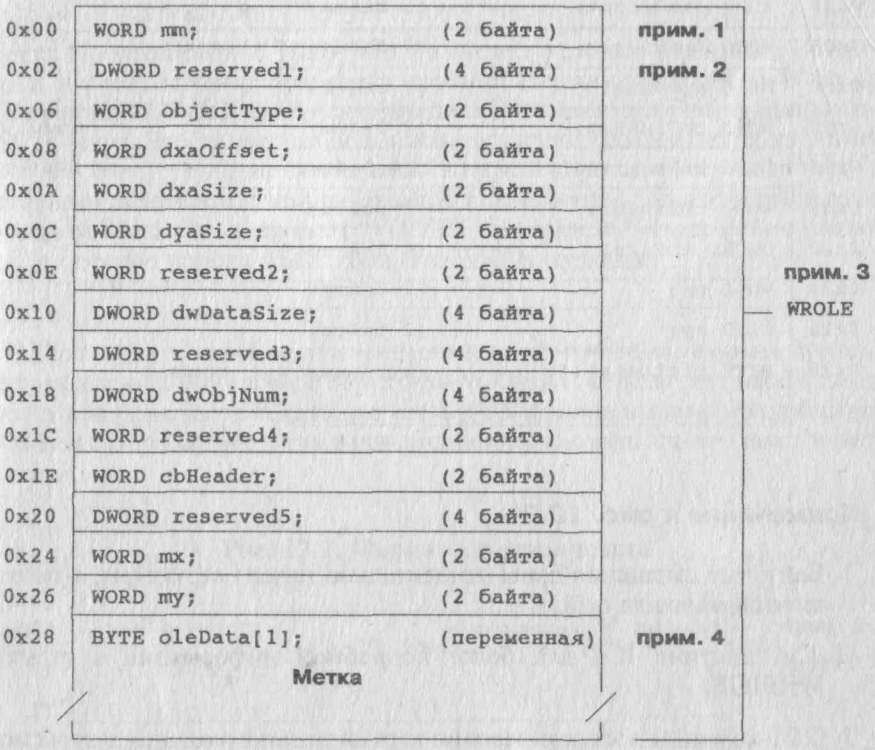
1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.3 для более подробной информации о структуре WRPICT.
3. GDI-команды воспроизведения картинки запоминаются в этом месте, представленном здесь в виде байтового массива переменной длины.

WROLE

Кроме растровых изображений, Write может также запоминать объекты OLE (*протокол связывания и встраивания объектов*). Формат этих объектов, которые обычно (но не обязательно) являются картинками, показан на рис. 12.4.

Совет. Чтобы создать OLE-объект, просто перетащите документ в окно текстового процессора Write. Например, с помощью Менеджера файлов захватите мышью PCX-картинку или DOC-файл текстового процессора Word for Windows и перетащите его в WRI-документ, открытый в окне Write. OLE-объект представляется на экране в виде пиктограммы приложения, создавшего данный объект.

Рис. 12.4. OLE-объекты



Примечания к рис. 12.4

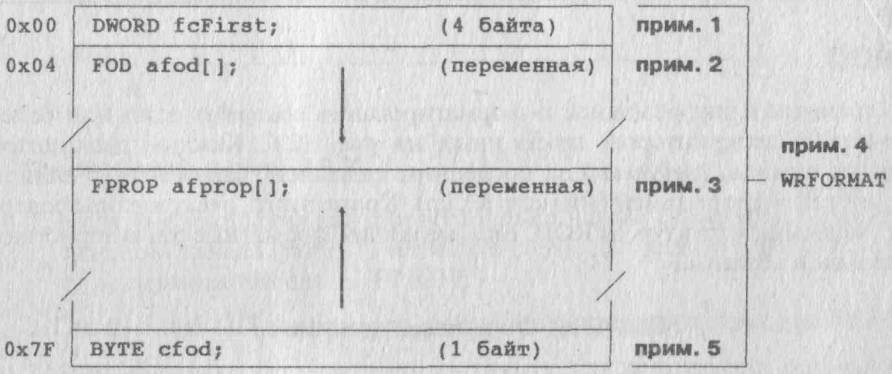
1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. Структура имеет резервные элементы, подобные этому, для того чтобы другие поля хотя бы частично соответствовали аналогичным полям структуры WRPICT. Тем не менее эти две структуры не взаимозаменяемы.

3. См. листинг 12.4 для более подробной информации о структуре WROLE.
4. Точный формат OLE-объектов зависит от создавшего их приложения. Программы, читающие или записывающие WRI-файлы, должны рассматривать эти данные как блок байтов, размер которого дан в элементе dwDataSize.

WRFORMAT

Информация о форматировании запоминается в 128-байтовых страницах, содержащих массив структур переменной длины, описывающих характеристики абзацев и символов. На рис. 12.5 показана структура форматных страниц, самая первая из которых адресуется элементом pnPara заголовка файла.

Рис. 12.5. Структура страницы с информацией о форматировании



Примечания к рис. 12.5

1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла. В элементе fcFirst дано смещение первого из тех символов, к которым относится информация о форматировании данной страницы.
2. Этот массив переменной длины хранит форматные дескрипторы типа FOD. См. рис. 12.6 и листинг 12.6 для более подробной информации о структуре FOD. Массив растет вниз, т.е. в сторону увеличения адресов памяти (новые элементы будут иметь большее смещение).

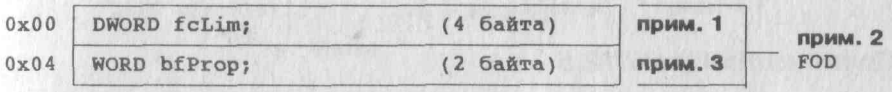
3. Этот массив переменной длины хранит форматные характеристики типа FPROP. См. рис. 12.7 и листинг 12.7 для более подробной информации о структуре FPROP. Массив растет вверх, т.е. в сторону уменьшения адресов памяти (новые элементы будут иметь меньшее смещение).
4. См. 12.5 для более подробной информации о структуре WRFORMAT.
5. Последний байт на 128-байтовой странице информации о форматировании содержит число структур типа FOD в массиве afod.

Совет. Длина каждой страницы с информацией о форматировании всегда равна 128 байтам. Чтобы прочитать их из WRI-файла, выделите 128-байтовый буфер и загружайте эти страницы с диска по одной. К сожалению, структуру WRFORMAT нельзя непосредственно использовать в программах, т.к. она имеет два массива переменной длины, что недопустимо в Си-структурах. Однако структурная диаграмма файла может пригодиться для доступа к индивидуальным элементам в буфере посредством индексирования последнего, как массива байтов.

FOD

Страницы с информацией о форматировании содержат один или более форматных дескрипторов, показанных на рис. 12.6. Каждый дескриптор адресует символ, *следующий* за последним символом, связанным с данной форматной информацией (элемент fcLim). Кроме этого, дескрипторы содержат смещения структур FPROP, описывающих форматные характеристики символов и абзацев.

Рис. 12.6. Форматный дескриптор



Примечания к рис. 12.6

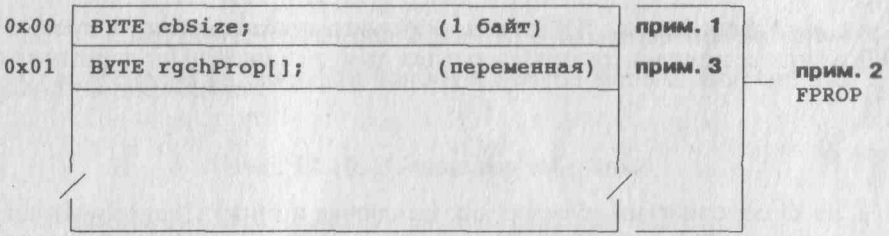
1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.6 для более подробной информации о структуре FOD.
3. Это байтовое смещение относится к началу массива afod. Оно указывает на структуру FPROP, связанную с данным форматным дескрип-

тором. См. рис. 12.7 и листинг 12.7 для более подробной информации о структуре FPROP.

FPROP

Форматные характеристики запоминаются в структурах FPROP. Каждая структура FPROP, показанная на рис. 12.7, адресуется соответствующим форматным дескриптором (FOD) и может содержать характеристики символов и абзацев.

Рис. 12.7 Форматные характеристики



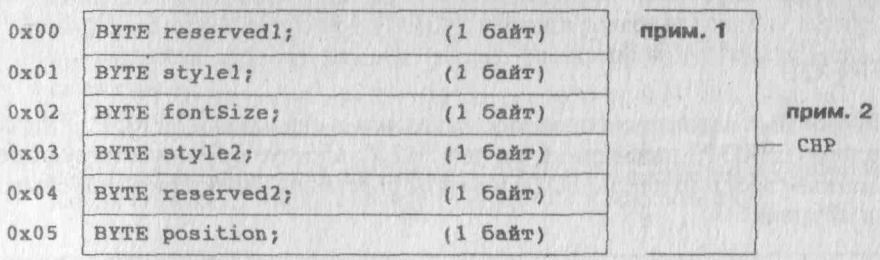
Примечания к рис. 12.7

1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла. В элементе cbSize дан размер в байтах структуры переменной длины FPROP.
2. См. листинг 12.7 для более подробной информации о структуре FPROP.
3. Этот массив может содержать частичную или полную структуру характеристик символов (CHP) или абзацев (PAP). См. рис. 12.8 и 12.9 о формате этих структур.

CHP

Такие характеристики символов, как шрифтовые стили и размеры, запоминаются в структурах CHP (листинг 12.8). Эти структуры либо частично, либо полностью хранятся в записях типа FPROP. Частичная структура CHP замещает собой начальные байты принятого по умолчанию объекта типа CHP, как поясняется в замечании к структуре CHP в разделе "Интерфейс с языком Си" далее в этой главе.

Рис. 12.8. Характеристики символов



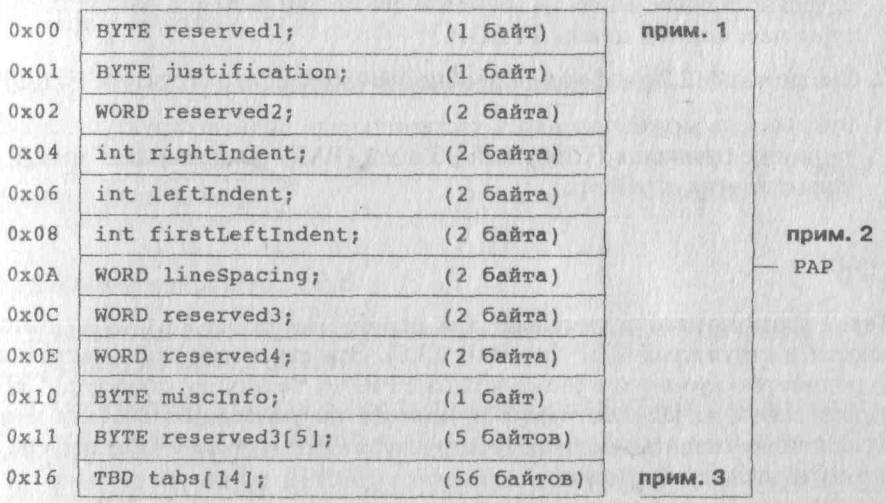
Примечания к рис. 12.8

1. Байтовые смещения даны относительно начала структуры.
2. См. листинг 12.8 для более подробной информации о структуре CHP.

PAP

Такие характеристики абзацев, как выключка и втяжка, запоминаются в структурах типа PAP (рис. 12.9). Эти структуры либо частично, либо полностью хранятся в записях типа FPROP. Частичная структура PAP замещает собой начальные байты принятого по умолчанию объекта типа PAP.

Рис. 12.9. Характеристики абзацев



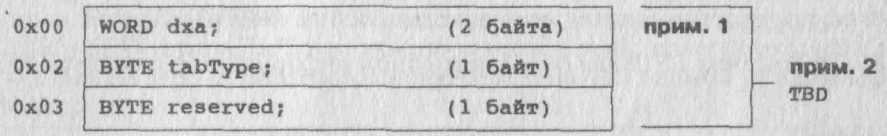
Примечания к рис. 12.9

1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.9 для более подробной информации о структуре PAP.
3. Структура характеристик абзаца может содержать до 14 табуляторных дескрипторов типа TBD. См. рис. 12.10 для информации о формате этого дескриптора.

TBD

Характеризующая абзац структура типа PAP может содержать до 14 дескрипторов табуляций. Формат этого дескриптора в виде структуры *TBD* показан на рис. 12.10.

Рис. 12.10. Дескриптор табуляций



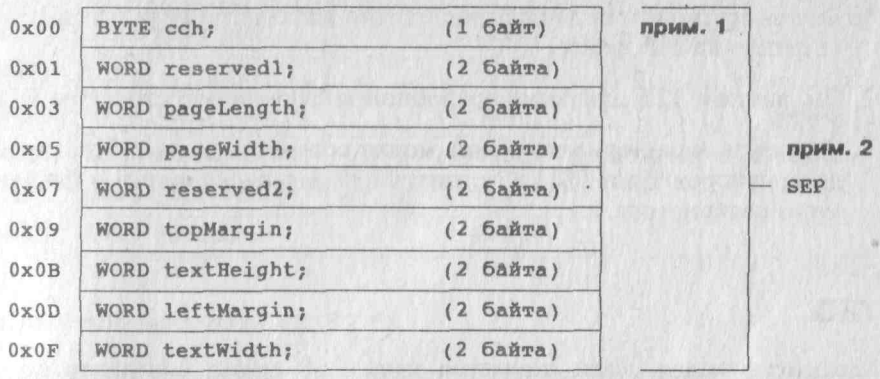
Примечания к рис. 12.10

1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.10 для более подробной информации о структуре TBD.

SEP

Характеристики раздела обычно относятся к более чем одному абзацу в документе. Однако файлы текстового процессора Write могут иметь лишь один раздел, и, следовательно, структура SEP (рис. 12.11), описывающая характеристики раздела, не так ценна. Текстовый процессор Microsoft Word *for DOS, после которого* Write вошел в моду, использует разделы с большей пользой.

Рис. 12.11. Характеристики раздела



Примечания к рис. 12.11

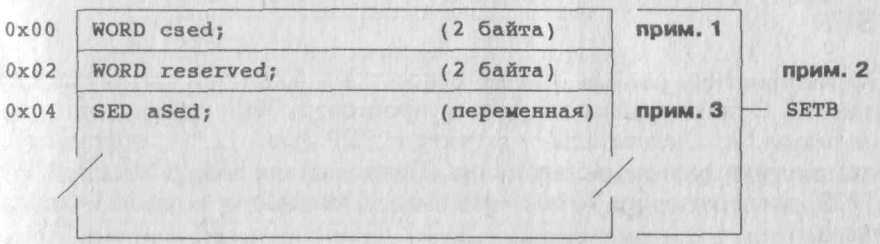
1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла. Используйте элемент pnSep структуры WRHEADER для определения местоположения в файле страницы характеристик раздела, если таковая вообще имеется в файле.
2. См. листинг 12.10 для более подробной информации о структуре SEP.

SETB

Таблица раздела (SETB) вместе с дескрипторами раздела (SED) образуют каталог записей типа SEP характеристик раздела. Поскольку WRI-файл имеет лишь один раздел, этот каталог не представляет практического интереса.

Формат таблицы раздела представлен на рис. 12.12. Эта информация может быть найдена в файле с помощью элемента pnSetb заголовка WRHEADER.

Рис. 12.12. Таблица раздела



Примечания к рис. 12.12

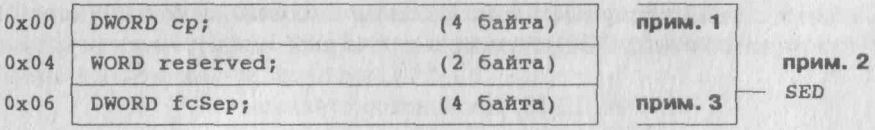
1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.12 для более подробной информации о структуре SETB.
3. Таблица раздела определяет расположение характеристик раздела в массиве дескрипторов раздела, имеющих тип SED. См. рис. 12.13 и листинг 12.13 для более подробной информации о структуре SED.

SED

Таблица раздела (SETB) хранит один или несколько дескрипторов раздела, имеющих тип SED (см. рис. 12.13). В этих структурах просто дано расположение в файле записей типа SEP, но т.к. WRI-файл может иметь только один раздел, структура SED не представляет практического интереса.

> **Совет.** Для доступа к структурам SEP в WRI-файле проще воспользоваться элементом pnSep заголовка WRHEADER, чем читать таблицу раздела (SETB) и дескрипторы (SED), которые ссылаются на ту же самую информацию. Если бы WRI-файлы могли иметь несколько разделов, структуры SETB и SED играли бы более заметную роль, чем ту, которая отведена им в настоящей версии текстового процессора Write (3.1 на момент написания этой книги).

Рис. 12.13. Дескриптор раздела



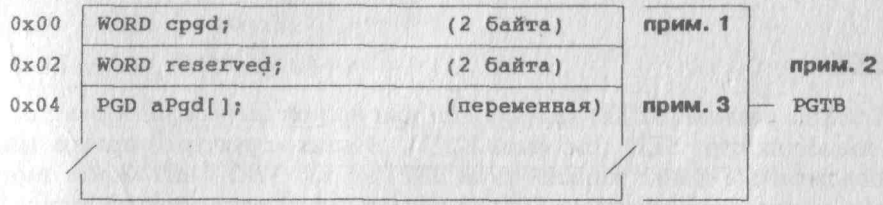
Примечания к рис. 12.13

1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.13 для более подробной информации о структуре SED.
3. В поле fcSep находится ссылка на запись типа SEP характеристик раздела, связанную с данным дескриптором SED. См. рис. 12.11 и листинг 12.11 для более подробной информации о структуре SEP.

PGTB

Таблица страниц документа делит WRI-файл на страницы. Однако не во всех файлах имеется эта информация. Формат таблицы страниц приведен на рис. 12.14. Доступ к этой таблице может быть осуществлен с помощью элемента pnPgtb заголовка файла.

Рис. 12.14. Таблица страниц



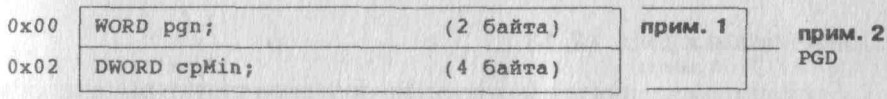
Примечания к рис. 12.14

1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.14 для более подробной информации о структуре PGTB.
3. Таблица страниц может иметь один или несколько дескрипторов типа PGD. См. рис. 12.15 и листинг 12.15 для более подробной информации о структуре PGD.

PGD

Таблица страниц документа состоит из одного или нескольких страничных дескрипторов типа PGD, показанного на рис. 12.15.

Рис. 12.15. Дескриптор страницы



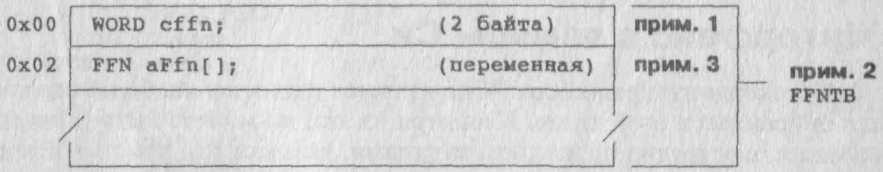
Примечания к рис. 12.15

1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.15 для более подробной информации о структуре PGD.

FFNTB

Наименования гарнитур шрифтов, использующихся в файле, хранятся в таблице наименований гарнитур, имеющей тип FFNTB (рис. 12.16). Для доступа к этой таблице внутри WRI-файла используйте элемент pnFntb заголовка WRHEADER.

Рис. 12.16. Таблица наименований гарнитур



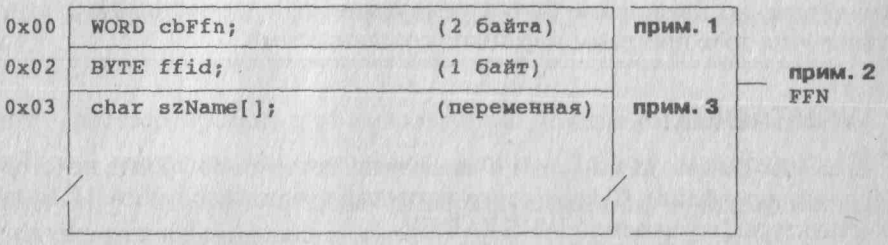
Примечания к рис. 12.16

1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.16 для более подробной информации о структуре FFNTB.
3. Таблица наименований гарнитур содержит одну или несколько записей типа FFN. См. рис. 12.17 и листинг 12.17 для более подробной информации о структуре FFN.

FFN

Наименования шрифтовых гарнитур хранятся в структурах FFN, формат которых показан на рис. 12.17. Эти структуры, в свою очередь, хранятся в таблице наименований гарнитур (FFNTB).

Рис. 12.17. Формат структуры FFN



Примечания к рис. 12.17

1. Байтовые смещения даны относительно начала структуры, а не относительно начала файла.
2. См. листинг 12.17 для более подробной информации о структуре FFN.
3. ASCIIZ-строка, содержащая наименование гарнитуры шрифта, выбранной с помощью команд Character | Fonts...

Интерфейс с языком Си

Файл текстового процессора Write является одним из наиболее сложных среди описанных в этой книге. Несмотря на это, вы можете быть удивлены, обнаружив, как трудно проследить за такими, казалось бы, простыми вещами, как набранный курсивом текст или выровненный абзац. Формат файлов Write — превосходный пример того, насколько сложные структуры данных могут потребоваться, чтобы реализовать сами собой разумеющиеся программные возможности.

Несмотря на сложность и большое число различных структур в WRI-файлах, все элементы данных состоят из обычных слов, двойных слов, целых чисел, символов и т.д. Имейте это в виду, когда будете изучать данную главу, распутывая непроходимые, как может показаться, хитросплетения элементов данных.

Замечание. Файлы Windows Write сделаны по образцу файлов Word for DOS — фактически эти программы могут читать файлы друг у друга. Однако при этом шрифтовая информация не всегда сохраняется. Так, Write игнорирует таблицы сносок, созданные в Word for DOS. Но более всего обескураживает тот факт, что документы, созданные в Word for Windows, полностью отличаются от документов, подготовленных текстовыми процессорами Word for DOS и Write. Несмотря на то, что Word for Windows может читать и записывать WRI-файлы, формат его собственных DOC-файлов совершенно не совместим с файлами, созданными Word for DOS или Write. Быть может, кто-нибудь и найдет наконец выход из этого лабиринта, но пока лучше быть внимательнее при совместном использовании этих трех программ и файлов, созданных ими.

WRHEADER

Все WRI-файлы начинаются с заголовка, который содержит некоторую информацию о файле. Формат этого заголовка показан в листинге 12.1 в виде Си-структуры, называемой WRHEADER.

Замечание. Структура WRHEADER, как и другие Си-структуры, рассматриваемые в этой главе, не объявлены в windows.h.

Листинг 12.1. WRHEADER

```
typedef struct tagWRHEADER {
    WORD    wIdent;
    WORD    dty;
    WORD    wTool;
    WORD    reserved1[4];
    DWORD   fcMax;
    WORD    pnPara;
    WORD    pnFntb;
    WORD    pnSep;
    WORD    pnSetb;
    WORD    pnPgtb;
    WORD    pnFfntb;
    WORD    reserved2[33];
    WORD    pnMax;
    WORD    reserved3[15];
} WRHEADER;
```

- wIdent — идентифицирует WRI-файл. Значение этого поля должно быть равным 0xBE31. Если же оно равно 0xBE32, то файл содержит объекты OLE. Файлы Word for DOS также имеют идентификатор 0xBE31.

Совет. Для того чтобы отличить файл Word for DOS от файла Windows Write, прочитайте его первое 16-разрядное слово. Если это число 0xBE32, то файл содержит OLE-объекты, и, следовательно, может быть создан только в текстовом процессоре Write. (Word for DOS не распознает протокол OLE.) Если же первое слово равно 0xBE31, то файл может быть создан как в Word for DOS, так и в Windows Write. В этом случае проверьте слово по смещению 0x60. Если оно равно нулю, файл был создан в Word for DOS; в противном случае — в Windows Write. Если первое слово файла не равно ни 0xBE31, ни 0xBE32, тогда он может быть создан в Word for Windows или быть просто текстовым файлом. Многие производители программного обеспечения распространяют тексты в DOC-файлах, хотя в Windows правильным расширением имени файла, содержащего ASCII-текст, является TXT.

- dty — зарезервировано и не используется. Должно равняться нулю.
- wTool — зарезервировано и не используется. Должно равняться 0xAB00.
- reserved1 — эти четыре слова зарезервированы и не используются. Они должны равняться нулю.

- fcMax — байтовый размер текстовой информации, содержащейся в файле, плюс размер одной страницы, т.е. 128 байтов. Чтобы узнать, сколько байтов занимает текст, следует вычесть 128 из fcMax.

Замечание. Страница — это один 128-байтовый сектор. Не путайте с текстовой страницей документа — они не имеют отношения друг к другу. Чтобы различать эти два понятия, я буду использовать словосочетание "страница документа". Первая страница в файле имеет номер 0, вторая — номер 1 и т.д.

- pnPara — номер страницы, с которой начинаются данные по форматированию абзацев. Все файлы должны иметь не менее одного абзаца (даже если текст отсутствует) и, следовательно, не менее одной страницы с информацией о формате.
- pnFntb — номер страницы, содержащей таблицу сносок, которая имеется в файлах Word for DOS. Если таблица сносок отсутствует, значение этого поля равно значению pnSep. Windows Write не использует таблицу сносок, так что в WRI-файлах это поле дублирует поле pnSep.
- pnSep — номер страницы, содержащий данные характеристик раздела (SEP). Если эти данные отсутствуют в файле, то значение этого поля равно значению поля pnSetb.
- pnSetb — номер страницы, содержащей таблицу разделов. Если таковая отсутствует в файле, в это поле помещается значение поля pnPgtb.
- pnPgtb — номер страницы, содержащей таблицу страниц документа. Если таковая отсутствует в файле, в это поле помещается значение поля pnFfntb.
- pnFfntb — номер страницы, содержащей таблицу наименований шрифтовых гарнитур. Если таковая отсутствует в файле, в это поле помещается значение поля pnMax.
- reserved2 — зарезервировано для текстового процессора Word for DOS.
- pnMax — число 128-байтовых страниц (секторов диска) в файле. Поскольку первая страница имеет номер 0, номер последней страницы равен pnMax-1.
- reserved3 — не документировано и не используется. Содержит нули. Это 30-байтовое поле просто дополняет заголовок файла до 128-байтовой страницы. Вам не следует читать эти байты.

Замечание. В структуре WRHEADER используется необычный прием индикации наличия или отсутствия в WRI-файле разного рода информации. Возьмем последовательность ABCD, где буквами представлены элементы структуры. Если A равно B, то объект, на который ссылается A, отсутствует в файле. Если B равно C, то объект, относящийся к B, не существует и т.д. Если же B не равен C, подразумевается, что B присутствует в файле. Например в WRHEADER, если элемент pnSetb равен pnPgtb, то объект, на который ссылается pnSetb (таблица разделов), отсутствует. Если же pnSetb не равен pnPgtb, то поле pnSetb адресует в файле таблицу разделов. Очевидно, что столь удачный метод присвоения одному элементу значения следующего исключает необходимость хранения флажков типа "эта информация отсутствует в файле". (В данном случае нуль мог бы сработать в качестве такого флажка, т.к. в нулевой странице может быть только заголовок файла.)

WRTEXT

Вслед за заголовком файла расположены одна или несколько страниц ASCII-текста, где все символы являются значащими и принадлежат к набору Windows ANSI. Абзац заканчивается парой управляющих кодов возврата каретки (0x0D) и перевода строки (0x0A). Принудительному обрыву строки соответствует управляющий код прогона страницы (0x0C). Символы табуляции имеют код 0x09. Никакие прочие управляющие коды не используются.

Замечание. Write хранит текст по абзацам, а не по строкам. Фактическая длина строки зависит от формата абзаца. Когда Write отображает документ на экране, большие абзацы разбиваются по строкам так, чтобы целиком уместиться в заданных пределах. Специальные символы конца строки и переноса слова отсутствуют.

Ради полноты описания, в листинге 12.2 показан формат текста WRI-файла в виде структуры языка Си, названной WRTEXT. Разумеется, вам не потребуется обрабатывать текст как массив переменной длины с единственным элементом типа char. В большинстве случаев, возможно, лучше будет прочитать абзац из файла в буфер и работать с текстом с помощью указателя символов.

Листинг 12.2 WRTEXT

```
typedef struct tagWRTEXT {
    char  wrText[1];
} WRTEXT;
```

Совет. Следующая формула официально рекомендована для вычисления номера начальной страницы WRI-файла, содержащей текстовые данные:

pn = (fcMax + 127) / 128

Например, если поле fcMax в структуре WRHEADER равно 207, то pn будет равняться 2. Однако поскольку текстовые данные в WRI-файлах всегда начинаются со второй страницы, нет необходимости прибегать к вычислением по приведенной выше формуле.

WRPICT

Документы, создаваемые в текстовом процессоре Write могут также содержать графические рисунки в расширенном формате метафайла или в виде растрового изображения. Картинки хранятся в WRI-файлах в виде структуры переменной длины WPICT, показанной в листинге 12.3.

Листинг 12.3. WPICT

```
typedef struct tagWRPICT {
    METAFILEPICT  mfp;
    WORD  dxaOffset;
    WORD  dxaSize;
    WORD  dyaSize;
    WORD  cbOldSize;
    BYTE  bmReserved[14];
    WORD  cbHeader;
    DWORD cbSize;
    WORD  mx;
    WORD  my;
    BYTE  pictData[1];
} WRPICT;
```

- mfp — структура METAFILEPICT, используемая системным буфером Windows. Это структура стандартная, за исключением одной особенности, используемой только редактором Write: если элемент mm (режим отображения) равен 0xE3, массив picdData содержит растровое изображение, в противном случае — метафайл. Элемент hMF этой структуры не используется. (См. главы 8 и 11 для информации по метафайлам и структуре METAFILEPICT.)
- dxaOffset — левое поле картинки в твипах (twips).

Замечание. Слово *твип* (twip) образовано от "twentieth of point" (одна двадцатая доля пункта). Так как 1 пункт равен 1/72 дюйма, один твип равен 1/1440 дюйма.

- dxaSize — ширина картинки в твипах.
- dyaSize — высота картинки в твипах.
- cbOldSize — не используется. Очевидно, когда-то предназначалось для представления размера картинки в байтах, но теперь всегда равно нулю.

Замечание. Префикс cb в именах элементов структур образован от count of bytes (счетчик байтов).

- bmReserved — зарезервировано для растровых изображений.
- cbHeader — общее число байтов в данной структуре без учета массива pictData.
- cbSize — число байтов в массиве pictData. Это поле заменяет собой cbOldSize, более не используемое.
- mx — горизонтальный коэффициент масштабирования.
- my — вертикальный коэффициент масштабирования.

Замечание. Число 1000 в элементах mx и my означает 100 процентов (нет масштабирования). Любое другое значение показывает, что размеры картинки были изменены с помощью команды Edit | Size текстового процессора Write.

- pictData — картинка в форме метафайла. Длина этого массива определяется в поле cbSize.

WROLE

Текстовый процессор Write может запоминать OLE-объекты, представляющие собой любые данные, созданные в программах, которые поддерживают протокол OLE. Структура WROLE, показанная в листинге 12.4, похожа на структуру WPICT из предыдущего раздела.

Замечание. Некоторые зарезервированные элементы структуры WROLE предусмотрены, очевидно, для выравнивания других элементов на те же самые позиции, что имеют аналогичные элементы структуры WPICT. Одноименные элементы в обоих структурах предназначены для одних и тех же целей.

Листинг 12.4. WROLE

```
typedef struct tagWROLE {
    WORD    mm;
    DWORD   reserved1;
    WORD    objectType;
    WORD    dxaOffset;
    WORD    dxaSize;
    WORD    dyaSize;
    WORD    reserved2;
    DWORD   dwDataSize;
    DWORD   reserved3;
    DWORD   dwObjNum;
    WORD    reserved4;
    WORD    cbHeader;
    DWORD   reserved5;
    WORD    mx;
    WORD    my;
    BYTE    oleData[1];
} WROLE;
```

- mm — режим отображения (но как таковой не используется). Должен быть равным 0xE4.
- reserved1 — не документировано и не используется. Очевидно, предусмотрено для выравнивания на соответствующие смещения других элементов структуры.
- objectType — тип объекта: статический (1), встроенный (2) или связанный (3).
- dxaOffset — левое поле объекта в твипах (1/20 пункта).
- dxaSize — ширина объекта в твипах (1/20 пункта).
- dyaSize — высота объекта в твипах (1/20 пункта).
- reserved2 — не используется и не документировано.
- dwDataSize — длина в байтах массива oleData.
- reserved3 — не используется и не документировано.
- dwObjNum — имя объекта, закодированное в виде 8-символьной строки, но запоминаемое и используемое в виде длинного целого.

- reserved4 — не используется и не документировано.
- cbHeader — общее число байтов в данной структуре без учета массива oleData.
- reserved5 — не используется и не документировано.
- mx — горизонтальный коэффициент масштабирования.
- my — вертикальный коэффициент масштабирования.
- oleData — OLE-объект. Формат этих данных зависит от программы, создавшей этот объект.

WRFORMAT

Кроме страниц, содержащих текст и картинки, Write запоминает информацию о формате символов и абзацев. Форматирование символов относится к одному или нескольким символам — например, к фрагменту текста, выделенного курсивом. Форматирование абзацев касается всего абзаца, завершающегося управляющими кодами возврата каретки и перевода строки. Форматирование символов и абзацев может применяться совместно. Так, например, абзац может содержать текстовые фрагменты с различными шрифтами и стилями (жирный, курсивный, подчеркнутый и т.д.).

Структура WRFORMAT в листинге 12.5 показывает, как организована страница с информацией о форматировании. Номер первой форматной страницы в файле определяется элементом pnPara структуры WRHEADER. Далее по мере необходимости следуют другие форматные страницы, описывающие характеристики всех абзацев и символов документа.

Предупреждение. Элементы afod и afprop являются объектами переменной длины. Следовательно, приведенная здесь структура WRFORMAT носит исключительно информационный характер и не должна компилироваться. Для использования информации о форматировании прочитайте поле afod в некоторую переменную. Это значение покажет относительную позицию символов документа, к которым относится данная форматная страница. Далее, прочитайте байт cfod, который содержит число записей типа FOD в массиве afod. Каждая такая запись обеспечивает доступ к структурам FPROP в массиве afprop, которые можно читать индивидуально по мере необходимости. С другой стороны, вы можете ввести всю страницу с информацией о форматах в 128-байтовый буфер и затем обращаться к ней посредством индексирования буфера некоторым целым i (например, buffer[i]).

Листинг 12.5. WRFORMAT

```
typedef ctruct tagWRFORMAT {
  DWORD fcFirst;
  FOD afod[1];
  FPROP afprop[1];
  BYTE cfod;
} WRFORMAT;
```

- fcFirst — байтовое смещение относительно начала файла первого символа из тех, к которым относится данная форматная страница.
- afod — массив дескрипторов форматов символов или абзацев. Каждый дескриптор обеспечивает доступ к информации о форматировании для одного абзаца или для нескольких символов. В структурах FOD этого массива даны расположения и, следовательно, номера записей типа FPROP массива afprop на этой же самой форматной странице (см. также структуру FOD в листинге 12.6).
- afprop — массив форматных характеристик, если таковые вообще имеются, которые относятся к символам или абзацу, определяемым соответствующей структурой FOD массива afod (см. также структуру FPROP в листинге 12.7).
- cfod — число структур FOD в массиве afod. Используйте это значение для определения размеров массива afod и косвенно (поскольку записи в afod указывают на записи в afprop) массива afprop.

Замечание. Массив afod растет от своего начала вниз, т.е. его последующие элементы имеют большее смещение в файле. Массив afprop растет в обратном направлении, начиная с байта, предшествующего элементу cfod, т.е. его последующие элементы имеют меньшее смещение в файле.

FOD

Форматный дескриптор FOD, показанный в листинге 12.6, определяет расположение в файле текста, подлежащего форматированию в соответствии с некоторыми характеристиками формата. Здесь также определяется расположение в файле соответствующих структур FPROP, содержащих упомянутые характеристики, закодированные в форме битовых полей и чисел. У каждого абзаца документа должна быть ровно одна структура FOD.

Листинг 12.6. FOD

```
typedef struct tagFOD {
    DWORD   fcLim;
    WORD    bfProp;
} FOD;
```

- fcLim — смещение в байтах относительно начала файла символа, следующего за тем, к которому относится этот форматный дескриптор. Используйте это значение для определения конца текстового фрагмента, отформатированного в соответствии с определяемыми данной структурой характеристиками.
- bfProp — смещение в байтах внутри массива afod структуры WRFORMAT до структуры FPROP, относящейся к данному дескриптору FOD. Для того чтобы найти структуру FPROP, связанную с заданным дескриптором FOD, найдите начало массива afod, а затем с помощью команды seek переместите внутренний указатель файла на bfProp байтов вперед. Теперь вы можете прочитать структуру FPROP, которая содержит информацию о форматировании символов и абзацев, рассматриваемую в следующем разделе.

Замечание. Эксперименты показали, что смещение bfProp равно 0xFFFF, если у данного дескриптора FOD отсутствует структура FPROP. Этот факт не упоминается в документации фирмы Microsoft, но, по-видимому, имеет место для WRI-файлов.

FPROP

Характеристики абзацев и символов запоминаются в структурах FPROP (листинг 12.7). В типичном WRI-файле одна структура FPROP может содержать информацию для нескольких дескрипторов FOD. Дескрипторы и связанные с ними структуры FPROP должны находиться на одной и той же 128-байтовой странице, и по этой причине WRI-файл может иметь несколько идентичных элементов типа FPROP. Говоря иначе, форматные характеристики одного и того же текстового фрагмента, выделенного курсивом и повторяющегося в разных местах, могут храниться в единственной структуре FPROP, если только все дескрипторы FOD, относящиеся к этому текстовому фрагменту, находятся на той же самой 128-байтовой странице.

Замечание. Структура FPROP имеет переменную длину. Ее размер в байтах дан в элементе cbSize.

Листинг 12.7. FPROP

```
typedef struct tagFPROP {
    BYTE  cbSize;
    BYTE  rgchProp[1]
} FPROP;
```

- cbSize — размер в байтах структуры FPROP.
- rgchProp — один или более байтов информации о форматировании символов (CHP) или абзаца (PAP). Здесь хранится минимальное количество информации. Все явно не указанные характеристики заменяются на подразумеваемые по умолчанию. В следующих двух разделах рассматриваются структуры CHP и PAP.

CHP

Элемент rgchProp в записях типа FPROP содержит один или несколько байтов либо структуры CHP, либо структуры PAP. Структура CHP, показанная в листинге 12.8, описывает форматные характеристики одного или нескольких символов.

Листинг 12.8. CHP

```
typedef struct tagCHP {
    BYTE  reserved1;
    BYTE  style1;
    BYTE  fontSize;
    BYTE  style2;
    BYTE  reserved2;
    BYTE  position;
} CHP;
```

- reserved1 — не используется. По умолчанию равняется 1.
- style1 — жирное начертание, если бит 0 установлен; курсивное начертание, если бит 1 установлен. Остальные биты (2—7) представляют собой индекс в таблице гарнитур, показывающий, какой шрифт следует использовать.
- fontSize — высота шрифта в полупунктах. Значение по умолчанию 24 определяет 12-пунктовую высоту.
- style2 — дополнительная информация о стиле. Если бит 0 установлен, то текст подчеркнутый. Остальные биты не используются.
- reserved2 — не используется.

- position — относительная позиция символа. 0 — нормальная позиция; 1—127 подстрочный текст; 128—255 — надстрочный текст.

Замечание. Элемент reserved1 по умолчанию устанавливается равным 1. Элемент fontSize по умолчанию содержит число 24. Остальные элементы структуры обнуляются. Поле переменной длины rgchProp структуры FPROP перекрывает эти установки по умолчанию байт за байтом. Если массив rgchProp имеет два байта, он перекрывает первые два байта списка форматных характеристик CHP, а остальные байты сохраняют значения по умолчанию. Этот метод экономит пространство в WRI-файле, т.к. не требуется запоминать всю структуру CHP, где большинство элементов принимают свои значения по умолчанию.

PAP

Запись типа FPROP может включать в себя дескриптор типа PAP (листинг 12.9) форматных характеристик абзаца. Эти характеристики относятся ко всему абзацу. Форматные характеристики типа CHP могут принадлежать текстовым фрагментам внутри абзаца. Как правило, абзац описывается одной структурой PAP и несколькими структурами CHP.

Листинг 12.9. PAP

```
typedef struct tagPAP {
    BYTE  reserved1;
    BYTE  justification;
    WORD  reserved2;
    int   rightIndent;
    int   leftIndent;
    int   firstLeftIndent;
    WORD  lineSpacing;
    WORD  reserved3;
    WORD  reserved4;
    BYTE  miscInfo;
    BYTE  reserved3[5];
    TBD   tabs[14];
} PAP;
```

- reserved1 — не используется. Должно равняться нулю.
- justification — определяет позицию абзаца относительно левого и правого пределов (выключка): 0 — выключка влево, 1 — выключка по центру, 2 — выключка вправо и 3 — выключка по формату.
- reserved2 — не используется. Должно равняться нулю.
- rightIndent — втяжка справа в твипах (1/20 пункта).

- leftIndent — втяжка слева в твипах (1/20 пункта).
- firstLeftIndent — величина абзацного отступа в твипах (1/20 пункта), т.е. дополнительная втяжка слева первой строки абзаца.
- lineSpacing — расстояние между строками в твипах (1/20 пункта).
- reserved3 — не используется. Должно равняться нулю.
- reserved4 — не используется. Должно равняться нулю.
- miscInfo — определяет прочую форматную информацию. Если бит 0 установлен, этот абзац является сноской; в противном случае это обычный абзац. Если бит 3 установлен, то абзац печатается на первой странице документа; в противном случае печать начинается на последующих страницах документа. Например, печать колонтитула на титульном листе может быть отключена. Если бит 4 установлен, то этот абзац является графической иллюстрацией; в противном случае он содержит текст. Другие биты не используются и должны быть сброшены.
- reserved3 — не используется. Все байты должны равняться нулю.
- tabs — содержит до 14 дескрипторов табуляций (структур TBD).

TBD

Дескрипторы табуляций хранят позиции и типы табуляций, установленных командой Document | Tabs... Структуры TBD (листинг 12.10) в массиве tabs структуры PAP (см. листинг 12.9).

Листинг 12.10. TBD

```
typedef struct tagTBD {
    WORD  dxa;
    BYTE  tabType;
    BYTE  reserved;
} TBD;
```

- dxa — расстояние в твипах (1/20 пункта) от левого края до заданной позиции табуляции.
- tabType — тип табуляции: 0x00 для текста (выключка влево относительно позиции табуляции), 0x03 для десятичных чисел (выключка вправо относительно позиции табуляции).
- reserved — не используется и не документировано.

SEP

Пользователи Word for Windows и Word for DOS могут использовать разделы для установки общих характеристик как для одного раздела, так и для всего текста документа. Все документы имеют не менее одного раздела, и если вы не определили таковой, установки раздела по умолчанию действуют на весь текст документа.

В Windows Write можно определить лишь один раздел, который не играет столь значительной роли, как в других двух более интеллектуальных текстовых процессорах фирмы Microsoft. Информация раздела хранится в структурах SEP (листинг 12.11). Расположение этих структур в файле определяется полем pnSep структуры WRHEADER (см. листинг 12.1). Структура SEP занимает 17 байтов.

Листинг 12.11. SEP

```
typedef struct tagSEP {
    BYTE  cch;
    WORD  reserved1;
    WORD  pageLength;
    WORD  pageWidth;
    WORD  reserved2;
    WORD  topMargin;
    WORD  textHeight;
    WORD  leftMargin;
    WORD  textWidth;
} SEP;
```

- cch — размер данной структуры в байтах без учета поля cch. Если значение этого поля меньше, чем sizeof(SEP) - 1, элементы структуры SEP принимают значения по умолчанию, перечисленные в табл. 12.1.
- reserved1 — не используется. Должно равняться нулю.
- pageLength — высота страницы документа в твипах (1/20 пункта).
- pageWidth — ширина страницы документа в твипах (1/20 пункта).
- reserved2 — не используется. Должно равняться 0xFFFF.
- topMargin — верхнее поле страницы документа в твипах (1/20 пункта).
- textHeight — высота области текста в твипах (1/20 пункта).
- leftMargin — левое поле страницы документа в твипах (1/20 пункта).
- textWidth — ширина области текста в твипах (1/20 пункта).

Табл. 12.1.

Значения полей структуры SEP по умолчанию

Элемент структуры SEP	Значение по умолчанию в твипах	Значение по умолчанию в дюймах
yaMax	15840	11
xaMax	12240	8.5
yaTop	1440	1
dyaText	12960	9
xaLeft	1800	1.25
dxaText	8640	6

Замечание. Как упоминалось ранее в этой главе, один твип равен 1/20 пункта. Поскольку пункт равен 1/72 дюйма, один твип равен 1/1440 дюйма — прекрасная мера, удобная для позиционирования абзацев, выравнивания и прочих структурных единиц документа. Все элементы структуры SEP, кроме первого, представляются в твипах. Например, число 1440 представляет 1 дюйм, число 15840 — 11 дюймов и т.д. Твипы предусмотрены также для представления дробных чисел в виде целых значений. Так, например, число 12240 представляет 8.5 дюймов.

SETB

Таблица разделов SETB, показанная в листинге 12.12, хранит один или несколько дескрипторов раздела типа SED. Эта таблица используется для файлов Word for DOS как каталог множества структур типа SEP, содержащих характеристики раздела. Поскольку Windows Write создает только один раздел, эта таблица не слишком сложна. WRI-файлы могут иметь или не иметь таблицу разделов в зависимости от значения поля pnSetb заголовка WRHEADER (см. листинг 12.1).

Совет. Вероятно, нет необходимости использовать таблицу разделов WRI-файла. Поскольку документы текстового процессора Write имеют лишь один раздел, для определения местоположения информации о разделе можно использовать поле pnSep структуры WRHEADER. Таблица разделов просто указывает на те же самые данные и, таким образом, является избыточной.

Листинг 12.12. SETB

```
typedef struct tagSETB {
    WORD  csed;
    WORD  reserved;
    SED   aSed[1];
} SETB;
```

- csed — счетчик дескрипторов разделов в таблице. Если таблица разделов присутствует в WRI-файле, значение этого поля всегда равно 0x02.
- reserved — не используется и не документировано.
- aSed — массив дескрипторов типа SED.

Замечание. Первый дескриптор раздела настоящий, а второй — фиктивный и служит для обозначения конца массива aSed. Это объясняет то, что поле csed равно 0x02, а не 0x01, несмотря на тот факт, что документ Write может иметь только один раздел.

SED

Дескриптор раздела типа SED (см. листинг 12.13) похож на элемент дискового каталога. Он адресует данные, хранящиеся где-то в другом месте — в данном случае структуры типа SEP (см. листинг 12.11), содержащие характеристики раздела.

Листинг 12.13. SED

```
typedef struct tagSED {
    DWORD  cp;
    WORD   reserved;
    DWORD  fcSep;
} SED;
```

- cp — смещение в байтах относительно начала файла до символа, *следующего* за тем, который принадлежит данному разделу.
- reserved — не используется и не документировано.
- fcSep — смещение в байтах относительно начала файла до структуры SEP, связанной с данным элементом типа SED.

Замечание. *Если документ текстового процессора Write имеет таблицу разделов, он содержит ровно две структуры типа SED. Поскольку WRI-файлы могут иметь только один раздел, первый элемент этой структуры (cp) содержит байтовое смещение после последнего символа в файле. Вторая, фиктивная структура типа SED в массиве aSed (см. листинг 12.12) имеет в поле cp число 0xFFFFFFFF. Если будущие версии Write позволят использовать много разделов, программа может применять это специальное значение для определения конца списка дескрипторов разделов типа SED.*

PGTB

Таблица страниц документа PGTB разбивает WRI-файл по страницам для вывода на печать и присваивает им номера. Таблица страниц документа показана в листинге 12.14.

Листинг 12.14. PGTB

```
typedef struct tagPGTB {
    WORD  cpgd;
    WORD  reserved;
    PGD   aPgd[1];
} PGTB;
```

- cpgd — счетчик структур PGD в массиве aPgd.
- reserved — не используется и не документировано.
- aPgd — массив дескрипторов страниц документа типа PGD.

PGD

Каждый дескриптор страницы документа в таблице страниц aPgd хранится в виде структуры PGD, показанной в листинге 12.15.

Листинг 12.15. PGD

```
typedef struct tagPGD {
    WORD   pgn;
    DWORD  cpMin;
} PGD;
```

- pgn — номер страницы документа.
- cpMin — смещение в байтах относительно начала файла до первого символа на заданной странице документа.

FFNTB

Если в файле имеется таблица наименований шрифтовых гарнитур, она запоминается в виде структуры FFNTB, показанной в листинге 12.16.

Листинг 12.16. FFNTB

```
typedef struct tagFFNTB {
    WORD  cffn;
    FFN   aFfn[1];
} FFNTB;
```

- cffn — счетчик структур типа FFNTB в массиве aFfn.
- aFfn — массив структур FFNTB, определяющих наименования шрифтовых гарнитур.

FFN

Каждый шрифт, используемый в WRI-файле, идентифицируется структурой FFN, показанной в листинге 12.17. Одна или несколько таких структур хранится в массиве aFfn.

Листинг 12.17. FFN

```
typedef struct tagFFN {
    WORD  cbFfn;
    BYTE  ffid;
    char  szName[1];
} FFN;
```

- cbFfn — смещение до следующего элемента FFN или нуль, если такового нет. Содержит 0xFFFF, если на следующей 128-байтовой странице находятся дополнительные наименования гарнитур.
- ffid — номер семейства шрифтов, например FF_ROMAN или FF_SWISS (см. табл. 7.2 в главе 7).
- szName — ASCIIZ-строка, содержащая имя гарнитуры. Эта строка имеет переменную длину и, следовательно, определяет размер структуры FFN в байтах.

Глава 13

ФАЙЛЫ ГРУПП (.GRP)

Менеджер программ (Windows Program Manager) запоминает свои дочерние окна в файлах групп, имена которых имеют расширение .GRP. В этих файлах он хранит имена программ и документов, пиктограммы и другие данные. Вы можете использовать информацию о формате файлов групп из этой главы для их чтения и записи, например в специальной установочной программе, создающей новое групповое окно.

Формат файла

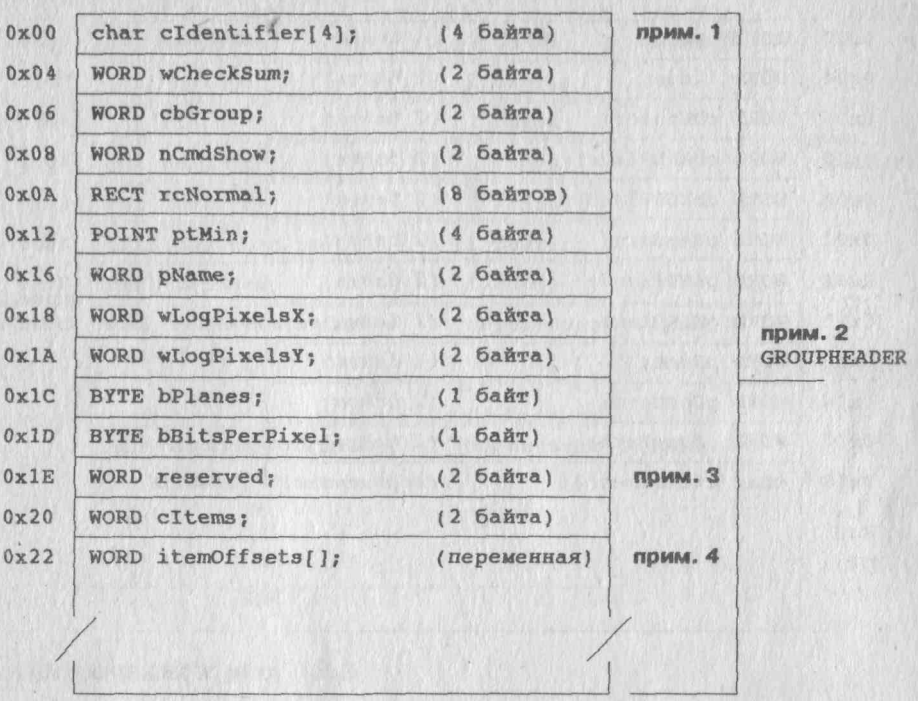
В следующей структурной диаграмме показан состав файла группы.

> **Совет.** Информация данной главы в общем случае касается любого приложения Windows, использующего многодокументный интерфейс (MDI). Даже если вам не потребуется писать программы для непосредственной обработки GRP-файлов, вы можете использовать структуры, подобные приведенным здесь, для хранения конфигураций MDI-окон.

GROUPHEADER

Файлы групп начинаются с универсального заголовка, показанного на рис. 13.1. Используйте его для проверки правильности формата GRP-файла, для проверки его целостности и для поиска другой информации в файле.

Рис. 13.1. GROUPHEADER



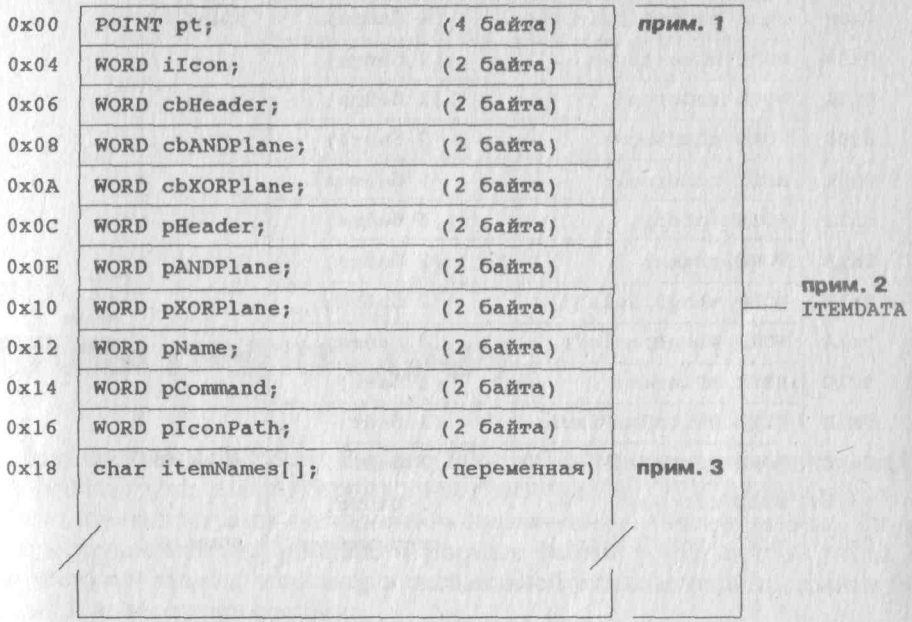
Примечания к рис. 13.1

1. Записывайте в это поле символьную строку PMCC, идентифицируя таким образом законный файл группы. Данная строка не имеет оконечного нулевого байта.
2. См. листинг 13.1 для более подробной информации об этой структуре.
3. Это поле не используется и должно содержать 0x0000.
4. Элементы этого массива содержат байтовые смещения структур типа ITEMDATA, рассматриваемых в следующем разделе, относительно начала файла

ITEMDATA

Описания каждой пиктограммы из группового окна хранятся в структурах типа ITEMDATA, формат которых показан на рис. 13.2.

Рис. 13.2. ITEMDATA



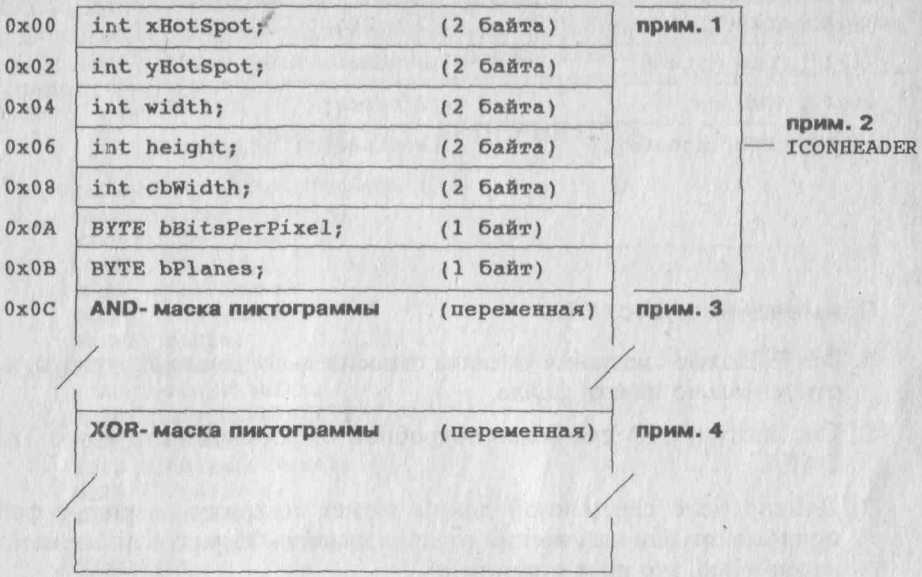
Примечания к рис. 13.2

1. Все байтовые смещения указаны относительно начала структуры, а не относительно начала файла.
2. См. листинг 13.2 для более подробной информации структуре ITEMDATA.
3. Имя пиктограммы, команда и ASCIIZ-строка, содержащая маршрут, хранятся в этом массиве последовательно друг за другом. Элементы pName, pCommand и pIconPath содержат байтовые смещения от начала файла до соответствующей символьной строки в этом массиве.

Растровые изображения пиктограмм

Менеджер программ копирует пиктограммы из EXE-файлов в GRP-файл каждого группового окна. Формат растровых изображений пиктограмм в файле группы показан на рис. 13.3.

Рис. 13.3. Растровые изображения пиктограмм в файле группы



Примечания к рис. 13.3

1. Все байтовые смещения указаны относительно начала структуры, а не относительно начала файла.

2. См. листинг 13.3 для более подробной информации о структуре ICONHEADER.

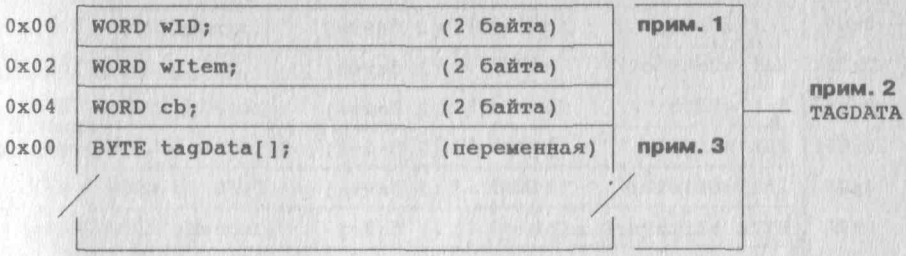
3. AND-маска пиктограммы является монохромным изображением, как и в файле пиктограмм (см. гл. 5).

4. XOR-маска пиктограммы, известная также как изображение переднего плана, обычно хранится в виде 16-цветного растрового изображения по 4 бита на пиксел.

TAGDATA

Некоторые GRP-файлы содержат дополнительную информацию, запоминаемую в конце файла в виде структур TAGDATA (рис. 13.4).

Рис. 13.4. TAGDATA



Примечания к рис. 13.4

1. Все байтовые смещения указаны относительно начала структуры, а не относительно начала файла.
2. См. листинг 13.4 для более подробной информации структуре TAGDATA.
3. Данное поле переменной длины может содержать маршрут файла приложения или назначения горячих клавиш. Однако если элемент cb равен нулю, это поле отсутствует.

Интерфейс с языком Си

В этом разделе рассматриваются Си-структуры, описывающие формат файла групп.

GROUPHEADER

Структура заголовка файла групп GROUPHEADER, показанная в листинге 13.1, содержит различные элементы, описывающие групповое окно и имеющиеся в нем пиктограммы. Кроме того, с помощью этой структуры проверяется целостность GRP-файла.

Замечание. Этот раздел буквально начинен предупреждениями о многочисленных расхождениях между официальной документацией фирмы Microsoft и рассматриваемыми данными GRP-файлов. Основываясь на собственных экспериментах с реальными файлами групп, я полагаю, что информация, следующая далее, верна. Однако имейте в виду, что мои структуры отличаются от структур Microsoft.

Предупреждение. Структура GROUPHEADER имеет переменную длину. Чтобы использовать эту структуру, необходимо выделить достаточное количество памяти для ее хранения (см. гл. 2). Не определяйте в программе глобальных или автоматических переменных этого типа.

Листинг 13.1. GROUPHEADER

```
typedef struct tagGROUPHEADER {
    char    cIdentifier[4];
    WORD    wCheckSum;
    WORD    cbGroup;
    WORD    nCmdShow;
    RECT    rcNormal;
    POINT   ptMin;
    WORD    pName;
    WORD    wLogPixelsX;
    WORD    wLogPixelsY;
    BYTE    bPlanes;
    BYTE    bBitsPerPixel;
    WORD    reserved;
    WORD    cItems;
    WORD    itemOffsets[1];
} GROUPHEADER;
```

- cIdentifier — четырехбайтовый массив, содержащий символы PMCC без завершающего нулевого байта.

Совет. Вместо того чтобы читать первые четыре байта файла индивидуально, можно гораздо проще проверить файл на соответствие требуемому формату, введя эти байты в переменную типа DWORD. Если ее значение будет равно 0x43434D50, то файл, скорее всего, является файлом группы.

- wCheckSum — инвертированная сумма всех 16-разрядных слов файла, включая значение поля wCheckSum.

Замечание. Придерживайтесь следующего алгоритма при вычислении нового значения поля wCheckSum для файла групп:
1. Присвойте wCheckSum начальное значение 0xFFFF.
2. Присвойте переменной result начальное значение 0x0000.
3. Просуммируйте в переменной result все 16-разрядные слова файла, включая wCheckSum. При этом рассматривайте их как целое без знака и игнорируйте переполнение.
4. Логически инвертируйте полученную сумму (result = ~result).
5. Поместите значение переменной result в поле wCheckSum.

Эти пять шагов вычисляют контрольную сумму, которая запоминается в файле, упрощая проверку его целостности. Например, для тестирования GRP-файла просто повторите шаги 2 и 3, т.е. сложите все 16-битовые слова, включая wCheckSum и игнорируя переполнение. Если в результате получится 0x0000, то контрольная сумма совпала и файл, вероятнее всего, не испорчен. Любой другой результат показывает, что один или несколько байтов где-то внутри файла были изменены.

- cbGroup — размер GRP-файла в байтах без учета структур TAGDATA, присоединенных к концу файла. Используйте значение этого поля для доступа к элементам типа TAGDATA, содержащим дополнительную информацию о групповом окне, например маршрут по умолчанию или назначения горячих клавиш.

Предупреждение. В некоторых публикациях ошибочно утверждается, что в поле cbGroup запоминается число байтов в файле. Это справедливо для Windows 3.0, но не для Windows 3.1.

- nCmdShow — определяет, каким образом Менеджер программ отображает групповое окно на экране. Это поле может содержать три константы, объявленные в windows.h: SW_SHOWNORMAL (отображать окно нормального размера и в нормальной позиции), SW_SHOWMINIMIZED (отображать окно, свернутым в пиктограмму) или SW_SHOWMAXIMIZED.

Предупреждение. В некоторых публикациях говорится, что любая SW_-константа, объявленная в windows.h, может быть присвоена полю nCmdShow. Однако, проверив несколько GRP-файлов, я нашел лишь три константы, перечисленные в предыдущем абзаце. Во избежание проблем с Менеджером программ, используйте только их, но не SW_HIDE или SW_SHOWA.

- rcNormal — размер и место расположения группового окна внутри главного окна (окна Менеджера программ). Элементы left, top, right и bottom данной структуры используются только в том случае, когда nCmdShow равно SW_SHOWNORMAL.
- ptMin — координаты левого верхнего угла свернутого группового окна, отображаемого в виде пиктограммы. Целочисленные элементы x и y этой структуры типа POINT используются только в том случае, если nCmdShow равно SW_SHOWMINIMIZED. Значение поля ptMin касается лишь свернутого (минимизированного) группового окна в окне Менеджера программ, а не пиктограмм внутри окна.

Замечание. В официальной документации фирмы Microsoft говорится, что в ptMin определяются координаты левого верхнего угла группового окна. Видимо, это неправильно, т.к. элементы x, y данной структуры, судя по результатам тестирования нескольких GRP-файлов, все время равны -1, если групповое окно не свернуто.

- pName — байтовое смещение относительно начала файла первого символа ASCIIZ-строки, содержащей имя группового окна.

Замечание. Имя группового окна обычно следует за структурой GROUP-HEADER. Однако в целях большей безопасности используйте pName для доступа к этой символьной строке. Не полагайте, что она всегда идет сразу же вслед за заголовком, даже если может показаться, что это действительно так в GRP-файлах.

- wLogPixelsX — ширина пиктограммы группового окна в пикселах.
- wLogPixelsY — высота пиктограммы группового окна в пикселах.

Предупреждение. Очевидно, в официальной документации фирмы Microsoft допущена ошибка, когда утверждается, что wLogPixelsX и wLogPixelsY определяют разрешающую способность дисплея, используемого при создании пиктограмм Менеджера программ. Проверка нескольких GRP-файлов показала, что в этих полях фактически даны размеры пиктограмм, а не разрешающая способность дисплея.

- bPlanes — число цветовых плоскостей, используемых в изображении пиктограмм. Обычно равняется 0x01.
- bBitsPerPixel — число битов на пиксел в XOR-маске пиктограммы, содержащей изображение переднего плана пиктограммы. Обычно равно 0x04, определяя 16 цветов.

Предупреждение. В некоторых публикациях фирмы Microsoft только что рассмотренные два байта ошибочно объединяются в единое слово, называемое wBitsPerPixel. В других документах эти два значения переставляются местами. Проверка нескольких GRP-файлов показала, что bPlanes обычно равняется 0x01, а bBitsPerPixel — 0x04, из чего можно предположить правильность указанного выше порядка следования этих байтов.

- reserved — не используется и не документировано. Должно содержать 0x0000.

- cItems — счетчик элементов типа WORD массива itemOffsets — последнего поля структуры GROUPHEADER.

Замечание. Групповое окно может быть пустым, и тогда cItems равняется нулю. В этом случае массив itemOffsets отсутствует в заголовке.

- itemOffsets — массив 16-разрядных байтовых смещений от начала файла до структур типа ITEMDATA, рассматриваемых в следующем разделе. Выражение itemOffsets[0] определяет местоположение первой структуры, itemOffsets[1] — второй и т.д. Расположение последней структуры определяется выражением itemOffsets[cItems - 1].

Замечание. Обычно вслед за структурой GROUPHEADER располагается ASCIIZ-строка переменной длины, содержащая имя группового окна. Менеджер программ отображает ее в качестве заголовка группового окна или как подпись под свернутым в пиктограмму окном. Элемент pName заголовка содержит байтовое смещение первого символа этой строки относительно начала файла.

ITEMDATA

После заголовка файла группы следуют ноль, один или более объектов типа ITEMDATA — по одному на каждую пиктограмму в групповом окне. Структура ITEMDATA показана в листинге 13.2. Используйте массив itemOffsets структуры GROUPHEADER, чтобы получить байтовые смещения от начала файла до каждого элемента типа ITEMDATA.

Листинг 13.2. ITEMDATA

```
typedef struct tagITEMDATA {
    POINT  pt;
    WORD   iIcon;
    WORD   cbHeader;
    WORD   cbANDPlane;
    WORD   cbXORPlane;
    WORD   pHeader;
    WORD   pANDPlane;
    WORD   pXORPlane;
    WORD   pName;
    WORD   pCommand;
    WORD   pIconPath;
    char   itemNames[1];
} ITEMDATA;
```

- pt — содержит координаты левого нижнего угла пиктограммы из данного группового окна. (Структура POINT, объявленная в windows.h, имеет два целочисленных элемента — x и y.)
- iIcon — индекс исходного ресурса пиктограммы, хранящегося в EXE-файле, который связан с данным элементом группового окна.
- cbHeader — размер в байтах заголовка пиктограммы, хранящегося в файле группы вместе с ее AND- и XOR-масками.

Предупреждение. Элемент cbHeader называется cbResource в официальной документации фирмы Microsoft, где также говорится, что здесь запоминается размер ресурса пиктограммы, хранимого в соответствующем EXE-файле. Проверка GRP-файлов показала, что эта информация не верна и cbHeader (обычно равный 0x000C) больше похож на размер 12-байтового заголовка, который предшествует маскам пиктограммы в файле группы.

- cbANDPlane — размер в байтах AND-маски пиктограммы, хранящейся в файле группы.
- cbXORPlane — размер в байтах XOR-маски пиктограммы, хранящейся в файле группы.
- pHeader — байтовое смещение от начала файла до заголовка пиктограммы, хранящегося в файле группы. Маски AND и XOR следуют за этим заголовком именно в том порядке, как они перечислены в этом предложении.
- pANDPlane — байтовое смещение от начала файла до AND-маски пиктограммы, хранящейся в файле группы.
- pXORPlane — байтовое смещение от начала файла до XOR-маски пиктограммы, хранящейся в файле группы.
- pName — байтовое смещение от начала файла до ASCIIZ-строки, содержащей подпись, отображаемую под пиктограммой в групповом окне. Эта строка имеет переменную длину.
- pCommand — байтовое смещение от начала файла до ASCIIZ-строки, содержащей команду, которая выполняется при выборе пиктограммы. Обычно эта строка, имеющая переменную длину, содержит имя программного файла. Например, WINFILE.EXE для запуска Менеджера файлов.
- pIconPath — байтовое смещение от начала файла до ASCIIZ-строки, содержащей полный маршрут программного файла. Эта строка имеет переменную длину.

- itemNames — ASCIIZ-строки, содержащие имя пиктограммы, команду и маршрут файла программы. Размер этого массива зависит от длины этих символьных строк. Следующая структура ITEMDATA идет сразу вслед за предыдущей. Растровые изображения пиктограмм и, возможно, структуры TAGDATA (рассматриваются далее в этой главе) следуют за последней структурой ITEMDATA.

> **Замечание.** Официальная структура ITEMDATA не включает в себя элемент itemNames. С технической точки зрения, символьные строки с наименованием пиктограммы, командой и маршрутом не обязательно должны входить в структуру, т.к. они адресуются смещениями pName, pCommand и pIconPath и, следовательно, могут быть расположены где-нибудь в другом месте внутри файла. Однако проверка нескольких GRP-файлов показала, что эти строки всегда располагаются вслед за соответствующими элементами типа ITEMDATA.

ICONHEADER

Менеджер программ копирует изображения из соответствующего EXE-файла в файл группы с целью наиболее быстрого доступа. Время от времени Менеджер программ снова обращается к EXE-файлу. Например, при запуске задачи или при смене дисплейного драйвера, что требует повторного создания файлов групп с пиктограммами другого цвета или с другим разрешением. Однако наиболее часто вы видите в групповых окнах копии оригинальных пиктограмм, хранящиеся в соответствующих GRP-файлах.

Каждое изображение пиктограммы в файле группы состоит из трех частей: заголовка, AND-маски и XOR-маски. Назначение масок объясняется в гл. 4, 5 и 6. Заголовок представлен в формате структуры ICONHEADER, показанной в листинге 13.3.

Листинг 13.3. ICONHEADER

```
typedef struct tagICONHEADER {
    int   xHotSpot;
    int   yHotSpot;
    int   width;
    int   height;
    int   cbWidth;
    BYTE  bBitsPerPixel;
    BYTE  bPlanes;
} ICONHEADER;
```

- xHotSpot — не используется. Это поле должно содержать 0x0000.
- yHotSpot — не используется. Это поле должно содержать 0x0000.

Замечание. Два предыдущих элемента, которые должны были бы содержать координаты горячей точки, по-видимому, были зарезервированы для представления курсоров в файлах групп. Не совсем понятно, зачем в файле группы могут понадобиться пиктограммы курсоров, однако очевидно, что заголовок пиктограммы готов запомнить эту информацию, если таковая появится в будущих версиях Windows.

Предупреждение. Microsoft полагает, что поля xHotSpot и yHotSpot заголовка пиктограммы должны содержать 0x0000. Однако в фактических GRP-файлы оба этих поля равны 0x0010 по непонятным причинам. Поскольку горячие точки имеют смысл только для курсоров, а не для пиктограмм, эти расхождения, по-видимому, не столь важны.

- width — ширина пиктограммы в пикселах.
- height — высота пиктограммы в пикселах.
- cbWidth — число байтов в строке пикселов изображения. Строки выровнены на границу слова, поэтому значение этого поля всегда четное.
- bBitsPerPixel — число битов на пиксел в XOR-маске, которая содержит изображение переднего плана пиктограммы. Обычно равняется 0x04, определяя 16 цветов.
- bPlanes — число цветовых плоскостей, используемых в изображении пиктограммы. Обычно равняется 0x01.

Предупреждение. В некоторых публикациях изменяется порядок следования этих двух полей. Однако проверка нескольких реальных GRP-файлов показала правильность указанного здесь порядка следования этих элементов.

Маски пиктограммы AND (фоновое изображение) и XOR (изображение переднего плана) следуют сразу же за структурой ICONHEADER. Использование и формат этих растровых изображений описаны в гл. 4.

TAGDATA

Вслед за изображениями пиктограмм в файле группы могут располагаться дополнительные структуры переменной длины типа TAGDATA (листинг 13.4). Заметьте, что не все файлы групп имеют эти структуры.

Совет. GRP-файлы содержат структуры TAGDATA, если элемент cbGroup структуры GROUPHEADER не равен длине файла в байтах. В этом случае выражение cbGroup + 1 определяет байтовое смещение в файле до массива структур TAGDATA.

Листинг 13.4. TAGDATA

```
typedef struct tagTAGDATA {
    WORD  wID;
    WORD  wItem;
    WORD  cb;
    BYTE  tagData[1];
} TAGDATA;
```

- wID — тип информации, содержащейся в массиве tagData данной структуры. Содержит одну из четырех констант из табл. 13.1.

Табл. 13.1.

Идентификаторы структуры TAGDATA

Константа	Назначение элемента tagData
0x8101	tagData содержит маршрут файла приложения
0x8102	tagData содержит назначение горячей клавиши
0x8103	tagData отсутствует
0xFFFF	Маркер конца массива структур TAGDATA

- wItem — индекс структуры ITEMDATA, к которой относится данный элемент типа TAGDATA.
- cb — число байтов, которое занимает данная структура TAGDATA. Если это поле содержит 0x0000, tagData не существует.
- tagData — содержит информацию в одном из форматов, перечисленных в табл. 13.1. Этот элемент не существует, если wItem равен 0x8103 или 0xFFFF.

Предупреждение. В некоторых публикациях tagData по непонятным причинам именуется rgb. Этот элемент также неверно упоминается как указатель.

Первой структуре TAGDATA в файле группы предшествует недокументированный 9-байтовый заголовок, обычно имеющий следующий вид:

```
80 FF FF 0A 00 50 4D 43 43
```

Последние четыре байта этой последовательности представляют собой ASCII-символы PMCC, хранящиеся также в поле cIdentifier структуры GROUPHEADER в начале всех GRP-файлов. Непонятно, почему данная строка повторяется в этом месте — вероятно, для дополнительной проверки правильности формата файла.

Первые пять байтов приведенного выше заголовка расшифровываются не так просто. Первый байт (0x80), кажется, является флагом, хотя его назначение не ясно. Далее идущее слово (0xFFFF) может быть зарезервированным или неиспользуемым. Следующее слово (0x000A), вероятнее всего, представляет собой байтовое смещение от начала заголовка до первой структуры TAGDATA.

Замечание. Фиктивная структура служит маркером конца массива структур TAGDATA. В этой фиктивной структуре элементы wID и wItem содержат 0xFFFF, элемент cb равен 0x0000, а массив tagData отсутствует.

Предупреждение. Элемент cb маркера конца должен быть равным 0x0006 (размер фиктивной структуры в байтах), а не 0x0000. Эта ошибка может вызвать проблемы в программах, которые всегда предполагают ненулевое значение поля cb.

Глава 14

ИНФОРМАЦИОННЫЕ ФАЙЛЫ ПРОГРАММ (.PIF)

Windows использует PIF-файлы для того, чтобы подготовить память и выбрать различные режимы для запуска программ DOS. Имеются два стандартных PIF-файла: DOSPRMT.PIF для открытия окна DOS и _DEFAULT.PIF для запуска DOS-программ, не имеющих специального PIF-файла. Для создания и редактирования PIF-файлов применяется Редактор PIF-файлов (PIF Editor).

Формат PIF-файлов в Windows основан на похожем формате PIF-файлов в программной среде TopView, являющейся более ранней, но не совсем удачной попыткой снабдить PC DOS графическим пользовательским интерфейсом. Ранние версии Windows и TopView были конкурирующими, хотя в то время мало кто из пользователей PC уделял внимание подобным программным продуктам.

> **Совет.** Для того чтобы запустить DOS-программу, щелкните мышью два раза подряд по ее PIF-файлу в окне Менеджера файлов или перетащите этот PIF-файл в окно Менеджера программ и щелкните два раза по пиктограмме. Если вы запускаете EXE- или COM-программы DOS, выбрав соответствующие файлы в окне Менеджера файлов, то Windows использует установки из _DEFAULT.PIF. По этой причине всегда лучше запускать DOS-программы через их PIF-файлы.

К сожалению, фирма Microsoft выдала весьма скудную информацию о формате PIF-файлов и к тому же только для старой версии Windows 2.x и стандартного режима Windows 3.x. Официальная информация о формате новых PIF-файлов Windows 3.1, описываемого в данной главе по результатам экспериментов, фактически недоступна. Фирма Microsoft сообщила, что формат PIF-файлов подвержен изменениям, так что приведенная здесь информация, возможно, не будет касаться будущих версий Windows.

Замечание. Ввиду расхождения с официальной документацией по PIF-файлам, разрабатывая приведенные в этой главе структуры данных, я был свободен в выборе наименований полей, которые очень сильно напоминают текстовые метки в диалоговых окнах Редактора PIF-файлов. Это должно помочь вам установить соответствие между программными опциями и описываемыми здесь полями. В прочих публикациях по PIF-файлам используются другие имена.

Формат файла

PIF-файл начинается с заголовка, состоящего из различных полей, большинство из которых относятся к опциям, задаваемым в окне Редактора PIF-файлов. Формат этого заголовка показан на рис 14.1.

Совет. Поскольку любой PIF-файл занимает ровно 545 байтов дискового пространства, его будет проще всего обрабатывать в буфере такой же длины, чем читать показанные здесь структуры по отдельности. Если вы решите последовать этому предложению, используйте приведенные ниже диаграммы и соответствующие им Си-структуры в качестве руководства по полям вашего буферного массива.

PIFHEADER

Рис. 14.1. Заголовок PIF-файла

0x00	BYTE reserved1;	(1 байт)	прим. 1
0x01	BYTE reserved2;	(1 байт)	
0x02	char windowTitle[30];	(30 байтов)	прим. 2
0x20	WORD maxMem;	(2 байта)	
0x22	WORD minMem;	(2 байта)	
0x24	char programFilename[63];	(63 байта)	прим. 4
0x63	WORD msFlags;	(2 байта)	
0x65	char startupDirectory[64];	(64 байта)	прим. 5
0xA5	WORD reserved3[138];	(276 байтов)	прим. 6

прим. 3 — PIFHEADER

Примечания к рис. 14.1

1. Этот и следующий байты зарезервированы и не используются. Однако Редактор PIF-файлов записывает в следующий байт число 0xF0, когда выбирается опция Close Window on Exit (закрыть окно после выхода). В поле msFlags также имеется соответствующий бит для этой опции.
2. Эта и другие строки в PIF-файле содержат ASCII-символы, а в неиспользуемых позициях записываются пробелы (0x20). Эти строки фиксированной длины и не содержат оконечный нулевой байт.
3. См. листинг 14.1 для более подробной информации о структуре PIFHEADER.
4. Эта ASCII-строка фиксированной длины заполняется пробелами и не имеет оконечного нулевого байта.
5. Эта ASCII-строка фиксированной длины заполняется пробелами и не имеет оконечного нулевого байта.
6. Эта большая зарезервированная область содержит поля, оставшиеся от более старых PIF-файлов версии 2.x и стандартного режима. Пропускайте их при чтении новых PIF-файлов в 386 расширенном режиме Windows 3.1.

PIFDATA

Вслед за заголовком PIF-файла по байтовому смещению 0x1B9 расположены дополнительные установки, показанные на рис. 14.2. Эти значения включают в себя установки памяти, опции мультизадачности и различные переключатели, хранящиеся в байтах флажков.

Примечания к рис. 14.2

1. Все байтовые смещения в этой структуре отсчитываются от начала файла.
2. Эта структура имеет всего шесть флаговых байтов, которые содержат битовые поля и такие переключатели режимов, как фоновый/исключительный, горячие клавиши и др. Флаги с номерами 4 и 6 в настоящее время не используются и должны содержать 0x00. См. комментарии к листингу 14.2 о битовых флажках, располагающихся в этих полях.
3. См. листинг 14.2 для более подробной информации о структуре PIFDATA.
4. Скэн-код горячей клавиши приложения запоминается здесь, если только он задан в поле Application Shortcut Key (горячая клавиша

приложения) диалогового окна Advanced Option (расширенные опции) Редактора PIF-файлов.

5. Эти флажки определяют комбинацию клавиш Alt, Ctrl и Shift, являющуюся префиксом для скэн-кода горячей клавиши приложения. См. рис. 14.7 далее в этой главе для информации о каждом флаге поля appKeyFlags.

6. Эта строка фиксированной длины, содержащая необязательные параметры, является единственной заканчивающейся нулевым байтом строкой в PIF-файле. Когда DOS-программа запускается под Windows, эта строка присоединяется к командной строке, хранящейся в поле programFilename заголовка файла (см. рис. 14.1).

Рис. 14.2. PIFDATA

0x1B9	WORD memKBDesired;	(2 байта)	прим. 1
0x1BB	WORD memKBRequired;	(2 байта)	
0x1BD	WORD foregroundPriority;	(2 байта)	
0x1BF	WORD backgroundPriority;	(2 байта)	
0x1C1	WORD emsKBLimit;	(2 байта)	
0x1C3	WORD emsKBRequired;	(2 байта)	
0x1C5	WORD xmsKBLimit;	(2 байта)	
0x1C7	WORD xmsKBRequired;	(2 байта)	
0x1C9	BYTE fFlags1;	(1 байт)	прим. 2
0x1CA	BYTE fFlags2;	(1 байт)	
0x1CB	BYTE fFlags3;	(1 байт)	
0x1CC	BYTE fFlags4;	(1 байт)	
0x1CD	BYTE fFlags5;	(1 байт)	
0x1CE	BYTE fFlags6;	(1 байт)	
0x1CF	WORD reserved1;	(2 байта)	
0x1D1	WORD keyScanCode;	(2 байта)	прим. 4
0x1D3	WORD appKeyFlags;	(2 байта)	прим. 5
0x1D5	BYTE reserved2[12];	(2 байта)	
0x1E1	char optionalParameters[64];	(64 байта)	прим. 6

прим. 3 — PIFDATA

Интерфейс с языком Си

Простоты ради я разбил формат PIF-файла на две структуры — PIFHEADER и PIFDATA. Вы можете объединить их в одну большую структуру или читать/записывать их поля индивидуально по смещениям, показанным на рис. 14.1 и 14.2. Можно также прочитать PIF-файл полностью в 545-байтовый буфер и использовать эти структуры как руководство по доступу к требуемым полям. В следующих далее разделах описываются обе структуры, а также приводятся диаграммы шести флажковых байтов и одного флажкового слова в структуре PIFDATA.

PIFHEADER

В листинге 14.1 показана структура PIFHEADER, с которой начинается любой PIF-файл.

Листинг 14.1. PIFHEADER

```
typedef struct tagPIFHEADER {
    BYTE  reserved1;
    BYTE  reserved2;
    char  windowTitle[30];
    WORD  maxMem;
    WORD  minMem;
    char  programFilename[63];
    WORD  msFlags;
    char  startupDirectory[64];
    WORD  reserved3[138];
} PIFHEADER;
```

- reserved1 — не используется и не документировано. Должно содержать 0x00.
- reserved2 — не используется и не документировано. Должно содержать 0xF0, если задана опция Close Window on Exit (закрыть окно после выхода).

> **Замечание.** См. поле msFlags, в котором также имеется флажок для опции Close Window on Exit (закрыть окно после выхода).

- windowTitle — строка, отображаемая в качестве заголовка, если DOS-программа запущена в режиме окна. Не используется в полноэкранном режиме. Эта строка фиксированной длины заполняется пробелами (ASCII-код 0x20) в незанятых другими символами позициях. Оконечный нулевой байт отсутствует.

- maxMem — это поле, очевидно, не используется в 386 расширенном режиме PIF-файлов, однако всегда содержит константу 128.
- minMem — это поле, очевидно, также не используется в 386 расширенном режиме PIF-файлов, однако всегда содержит константу 128.

Замечание. Поля maxMem и minMem используются в старых 2.x версиях PIF-файлов, а также в стандартном режиме Windows 3.1.

- programFilename — имя файла программы, включая маршрут, если последний необходим для запуска программы. Эта строка фиксированной длины заполняется пробелами (ASCII-код 0x20) в незанятых другими символами позициях. Оконечный нулевой байт отсутствует.
- msFlags — прочие флажки. Если задана опция Close Window on Exit (закрыть окно после выхода), значение этого поля равно 0x0010, в противном случае — 0x0000.

Замечание. Поле msFlags содержит несколько битовых флажков, используемых в PIF-файлах старых версий Windows 2.x и PIF-файлах стандартного режима. В 386 расширенном режиме используется только бит 4 (подразумевается, что самый старший бит имеет номер 7).

- startupDirectory — каталог, который при запуске программы становится текущим. Обычно включает в себя идентификатор дисковода и маршрут. Эта строка фиксированной длины заполняется пробелами (ASCII-код 0x20) в незанятых другими символами позициях. Оконечный нулевой байт отсутствует.
- reserved3 — не используется и не документировано.

Совет. Поле reserved3 включается в структуру PIFHEADER только для справки. Вы можете исключить его без каких-либо последствий. Оно использовалось в PIF-файлах старых версий Windows 2.x и PIF-файлах стандартного режима.

PIFDATA

Дополнительные данные, идущие в PIF-файле вслед за заголовком, показаны в листинге 14.2 как элементы структуры PIFDATA.

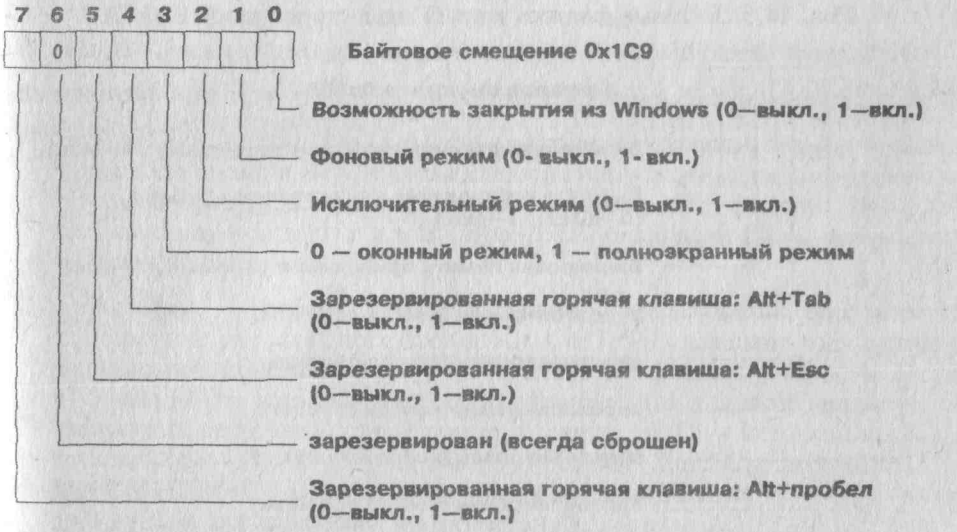
Листинг 14.2. PIFDATA

```
typedef struct tagPIFDATA {
    WORD  memKBDesired;
    WORD  memKBRequired;
    WORD  foregroundPriority;
    WORD  backgroundPriority;
    WORD  emsKBLimit;
    WORD  emsKBRequired;
    WORD  xmsKBLimit;
    WORD  xmsKBRequired;
    BYTE  fFlags1;
    BYTE  fFlags2;
    BYTE  fFlags3;
    BYTE  fFlags4;
    BYTE  fFlags5;
    BYTE  fFlags6;
    WORD  reserved1;
    WORD  keyScanCode;
    WORD  appKeyFlags;
    BYTE  reserved2[12];
    char  optionalParameters[64];
} PIFDATA;
```

- memKBDesired — количество памяти в килобайтах (1024 байта), желательное для программы. По возможности программе будет распределено не менее указанного здесь количества памяти. В окне Редактора PIF-файлов эта установка задается в строке с меткой Memory Requirements (требования по памяти). Может принимать значения из диапазона 0—640 или -1 для выделения всей доступной памяти.
- memKBRequired — количество памяти в килобайтах (1024 байта), необходимое для программы. Программа не будет выполняться, если указанное здесь количество памяти недоступно. В окне Редактора PIF-файлов эта установка задается в строке с меткой Memory Requirements (требования по памяти). Может принимать значения из диапазона 0—640 или -1 для выделения всей доступной памяти.
- foregroundPriority — приоритет, задаваемый в поле с меткой Foreground Priority (приоритет переднего плана) раздела Multitasking Options (опции мультизадачности) диалогового окна Advanced Options (расширенные опции) Редактора PIF-файлов. Принимает значения из диапазона 1—10000.
- backgroundPriority — приоритет, задаваемый в поле с меткой Background Priority (фоновый приоритет) раздела Multitasking Options (опции мультизадачности) диалогового окна Advanced Options (расширенные опции) Редактора PIF-файлов. Принимает значения из диапазона 1—10000.

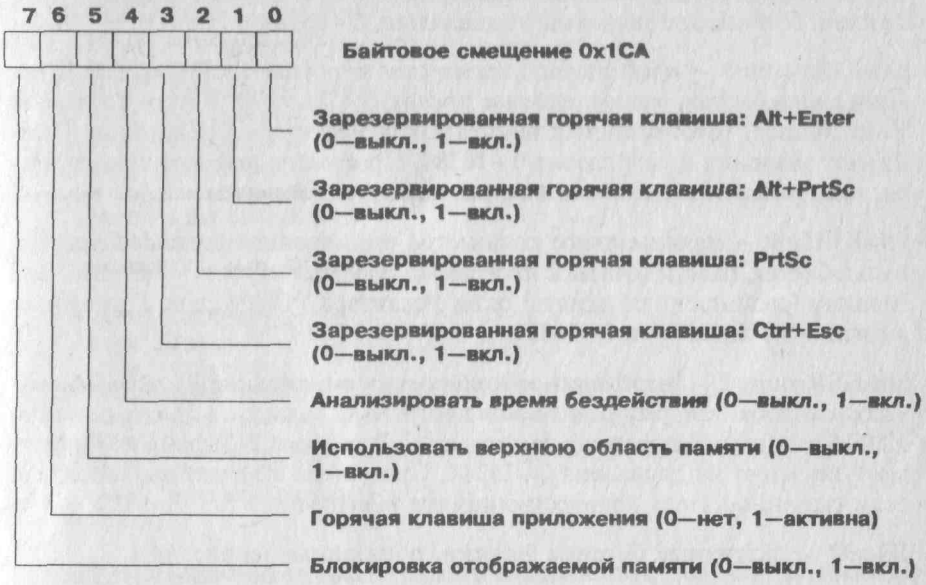
- emsKBLimit — максимальное количество отображаемой (expanded) памяти в килобайтах, распределяемое программе. Задается в строке с меткой EMS Memory (отображаемая память) диалогового окна Редактора PIF-файлов. Принимает значения из диапазона 0—16384.
- emsKBRequired — необходимое количество отображаемой (expanded) памяти в килобайтах, распределяемое программе. Задается в строке с меткой EMS Memory (отображаемая память) окна Редактора PIF-файлов. Принимает значения из диапазона 0—16384. Программа не будет выполняться, если указанное здесь количество памяти недоступно.
- xmsKBLimit — максимальное количество расширенной (extended) памяти в килобайтах, распределяемое программе. Задается в строке с меткой XMS Memory (расширенная память) окна Редактора PIF-файлов. Принимает значения из диапазона 0—16384.
- xmsKBRequired — необходимое количество расширенной (extended) памяти в килобайтах, распределяемое программе. Задается в строке с меткой XMS Memory (расширенная память) окна Редактора PIF-файлов. Принимает значения из диапазона 0—16384. Программа не будет выполняться, если указанное здесь количество памяти недоступно.
- fFlags1 — различные битовые флажки, показанные на рис. 14.3.

Рис. 14.3. Битовые флажки поля fFlags1 структуры PIFDATA



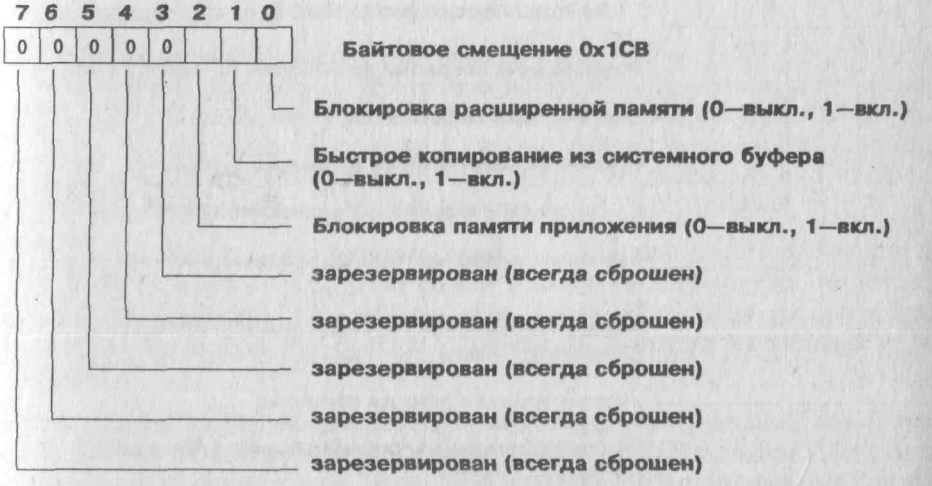
- fFlags2 — различные битовые флажки, показанные на рис. 14.4.

Рис. 14.4. Битовые флажки поля fFlags2 структуры PIFDATA



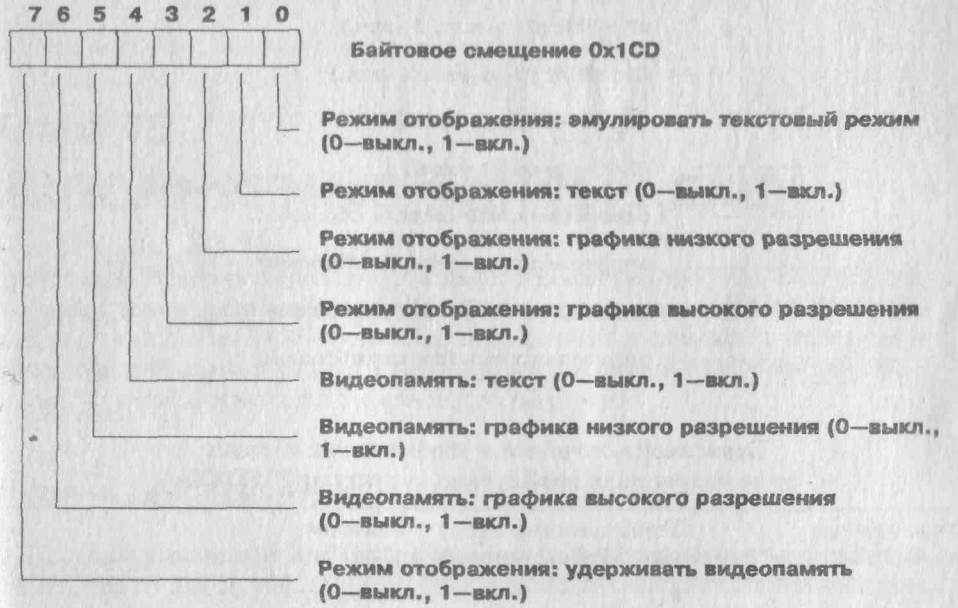
- fFlags3 — различные битовые флажки, показанные на рис. 14.5.

Рис. 14.5. Битовые флажки поля fFlags3 структуры PIFDATA



- fFlags4 — в настоящее время не используется. Это поле должно содержать 0x00.
- fFlags5 — различные битовые флажки, показанные на рис. 14.6.

Рис. 14.6. Битовые флажки поля fFlags5 структуры PIFDATA



- fFlags6 — в настоящее время не используется. Это поле должно содержать 0x00.
- reserved1 — не используется и не документировано. Должно содержать 0x00.
- keyScanCode — клавиатурный скэн-код горячей клавиши, заданной в поле Application Shortcut Key (горячая клавиша приложения) диалогового окна Advanced Option (расширенные опции) Редактора PIF-файлов. Этот флажок используется только в том случае, если бит 6 поля Flags2 установлен (горячая клавиша активна).
- appKeyFlags — набор битовых флажков, определяющих префиксную комбинацию управляющих клавиш Alt, Ctrl и Shift для скэн-кода горячей клавиши приложения. На рис. 14.7 показаны все флажки этого поля. Биты 0—3 могут быть установлены в любой комбинации для выбора одного из следующих сочетаний управляющих клавиш, где C — скэн-код из поля keyScanCode: Alt+C, Ctrl+C, Alt+Ctrl+C, Alt+Shift+C, Ctrl+Shift+C и Alt+Ctrl+Shift+C. В табл. 14.1 показаны комбинации флажков поля appKeyFlags для каждого и этих сочетаний.

Рис. 14.7. Битовые флажки поля appKeyFlags структуры PIFDATA

Табл. 14.1.

Зависимость сочетания управляющих клавиш от значения поля appKeyFlags структуры PIFDATA

Управляющая комбинация	Установленные биты поля appKeyFlags	Значение поля appKeyFlags
Alt	3	0x08
Ctrl	2	0x04
Alt+Ctrl	3, 2	0x0C
Alt+Shift	3, 0	0x09
Ctrl+Shift	2, 1	0x06
Alt+Ctrl+Shift	3, 2, 1, 0	0x0F

- optionalParameters — ASCIIZ-строка фиксированной длины, содержащая параметры, которые должны быть добавлены к командной строке, определенной в поле programFilename структуры PIFHEADER. Заполняется пробелами (ASCII-код 0x20) в незанятых другими символами позициях.

Замечание. Строка optionalParameters имеет оконечный нулевой байт. Другие символьные строки в PIF-файле не завершаются нулями. Имейте в виду это отличие, когда будете читать или записывать строковые поля.

Глава 15
ИСПОЛНЯЕМЫЕ ФАЙЛЫ (.EXE)

Программы Windows используют модифицированный формат EXE-файлов, который с точки зрения оболочки MS-DOS (как правило, COMMAND.COM) выглядит, как любая DOS-программа, но фактически скрывает в себе приложение Windows и его ресурсы. В данной главе рассматриваются многие фрагменты "нового" формата EXE-файлов для Windows.

Формат файла

Поскольку исполняемый файл содержит большое количество различных элементов, то для лучшего понимания рекомендуется сначала просмотреть приведенные здесь Си-структуры, возвращаясь время от времени к соответствующим диаграммам во время чтения описаний структур в разделе "Интерфейс с языком Си".

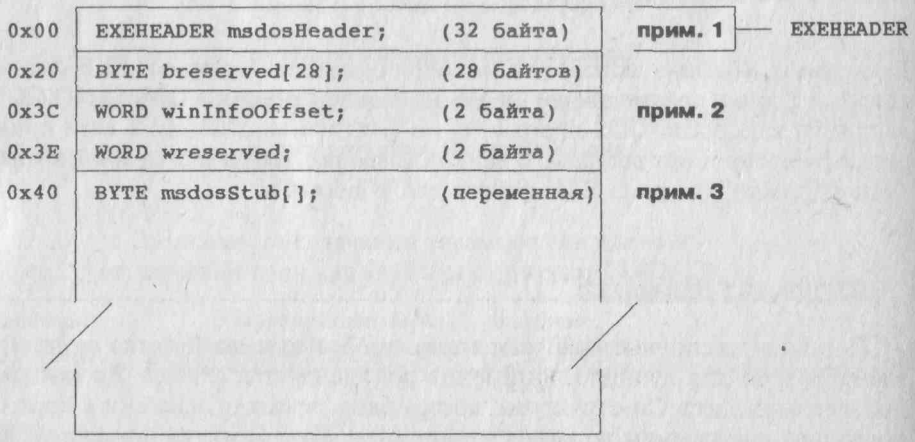
Замечание. Обычный прием загрузки файла в отладчик (например, MS-DOS DEBUG) не работает для EXE-файлов, т.к., помимо загрузки файла в память, отладчик также подготавливает его для выполнения. Например, если вы дали DOS-команду DEBUG NOTEPAD.EXE и затем нажали D+Enter для дампирования первого сектора файла, то увидите байты, отличные от тех, что хранятся на диске. Чтобы получить истинную картину, скопируйте NOTEPAD.EXE в NOTEPAD.DAT, например, а затем загрузите копию в отладчик. Тогда можно будет проверить фактически хранящиеся в файле значения.

Замечание. Различные таблицы могут находиться в исполняемом файле по любому байтовому смещению. Структуры не выравниваются на границы секторов.

OLDHEADER

По соображениям совместимости EXE-файлы Windows начинаются с заголовка в стиле DOS, который включает программную заглушку, печатающее сообщение "This application requires Microsoft Windows" (Эта программа требует Microsoft Windows), если пользователь попытается запустить приложение Windows из DOS. Заголовок также определяет расположение в файле данных, относящихся к Windows.

Рис. 15.1. OLDHEADER



Примечания к рис. 15.1

1. См. структуру EXEHEADER в листинге 15.2 для более подробной информации об этих полях.

2. Если слово по смещению 0x18 внутри структуры EXEHEADER больше или равно 0x40, то файл содержит приложение Windows и winInfoOffset адресует в файле информационный заголовок WININFO.

3. Программа-заглушка обычно отображает на экране сообщение "This application requires Microsoft Windows" (Эта программа требует Microsoft Windows), если пользователь попытается запустить приложение Windows из DOS. Однако она может быть полноценной DOS-программой, т.е. один EXE-файл может объединять DOS- и Windows-версии некоторой программы.

EXEHEADER

Рис. 15.2. EXEHEADER

0x00	WORD exSignature;	(2 байта)	прим. 1
0x02	WORD exExtraBytes;	(2 байта)	
0x04	WORD exPages;	(2 байта)	
0x06	WORD exRelocItems;	(2 байта)	
0x08	WORD exHeaderSize;	(2 байта)	
0x0A	WORD exMinAlloc;	(2 байта)	
0x0C	WORD exMaxAlloc;	(2 байта)	
0x0E	WORD exInitSS;	(2 байта)	
0x10	WORD exInitSP;	(2 байта)	
0x12	WORD exCheckSum;	(2 байта)	
0x14	WORD exInitIP;	(2 байта)	
0x16	WORD exInitCS;	(2 байта)	
0x18	WORD exRelocTable;	(2 байта)	
0x1A	WORD exOverlay;	(2 байта)	
0x1C	DWORD reserved;	(4 байта)	прим. 3

прим. 2 — EXEHEADER

Примечания к рис. 15.2

1. Байтовые смещения в этой структуре даны относительно начала файла. Слово exSignature занимает первые два байта всякого EXE-файла как в DOS, так и в Windows.
2. См. структуру EXEHEADER в листинге 15.2 для более подробной информации об этих полях.
3. Это двойное слово не включается в официальную структуру EXEHEADER.

WINHEADER

Если EXE-файл содержит *приложение* Windows, то соответствующие данные запоминаются в структуре WINHEADER, формат которой показан на рис. 15.3. Этот рисунок иллюстрирует общую организацию EXE-файлов Windows.

Рис. 15.3. WINHEADER

0x00	WININFO infoHeader;	(64 байта)	прим. 1
	Таблица сегментов	(переменная)	прим. 2
	Таблица ресурсов	(переменная)	прим. 3
	Таблица резидентных имен	(переменная)	прим. 4
	Таблица ссылок на модули	(переменная)	прим. 5
	Таблица импортированных имен	(переменная)	прим. 6
	Таблица входов	(переменная)	прим. 7
	Таблица нерезидентных имен	(переменная)	прим. 8
	Сегменты кода и данных	(переменная)	прим. 9

Примечания к рис. 15.3

1. Используйте элемент WinInfoOffset структуры OLDHEADER для определения местоположения структуры WINHEADER, в которой поле infoHeader содержит указатели на другие таблицы файла.
2. См. раздел TBSEGMENT в этой главе.
3. См. раздел TBRESOURCE в этой главе.
4. См. раздел TBRESNAME этой главе.
5. См. раздел TBMODULE в этой главе.
6. См. раздел TBIMPNAME в этой главе.
7. См. раздел TBENTRY в этой главе.
8. См. раздел TBNONRESNAME в этой главе.
9. Содержит также настроечную информацию. См. раздел "Сегменты кода и данных" в этой главе.

WININFO

Большая структура, показанная на рис. 15.4, описывает различные характеристики приложения, а также содержит несколько указателей на сегменты кода, ресурсы и другие таблицы.

Рис. 15.4. WININFO

0x00	WORD signature;	(2 байта)	прим. 1
0x02	BYTE linkerVersion;	(1 байт)	
0x03	BYTE linkerRevision;	(1 байт)	
0x04	WORD entryTabOffset;	(2 байта)	
0x06	WORD entryTabLen;	(2 байта)	
0x08	DWORD reserved1;	(4 байта)	
0x0C	WORD exeFlags;	(2 байта)	
0x0E	WORD dataSegNum;	(2 байта)	
0x10	WORD localHeapSize;	(2 байта)	
0x12	WORD stackSize;	(2 байта)	
0x14	DWORD cs_ip;	(4 байта)	
0x18	DWORD ss_sp;	(4 байта)	
0x1C	WORD segTabEntries;	(2 байта)	
0x1E	WORD modTabEntries;	(2 байта)	прим. 2 *WININFO*
0x20	WORD nonResTabSize;	(2 байта)	
0x22	WORD segTabOffset;	(2 байта)	
0x24	WORD resTabOffset;	(2 байта)	
0x26	WORD resNameTabOffset;	(2 байта)	
0x28	WORD modTabOffset;	(2 байта)	
0x2A	WORD impTabOffset;	(2 байта)	
0x2C	WORD nonResTabOffset;	(2 байта)	
0x2E	WORD reserved2;	(2 байта)	
0x30	WORD numEntryPoints;	(2 байта)	
0x32	WORD shiftCount;	(2 байта)	
0x34	WORD numResourceSegs;	(2 байта)	
0x36	BYTE targetOS;	(1 байт)	
0x37	BYTE miscFlags;	(1 байт)	
0x38	WORD fastLoadOffset;	(2 байта)	
0x3A	WORD fastLoadSize;	(2 байта)	
0x3C	WORD reserved3;	(2 байта)	
0x3E	BYTE winRevision;	(1 байт)	
0x3F	BYTE winVersion;	(1 байт)	

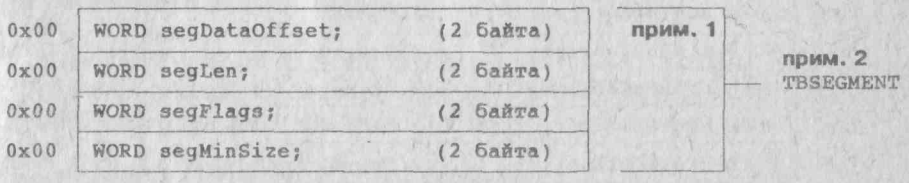
Примечания к рис. 15.4

1. Байтовые смещения в этой структуре даны относительно начала структуры, а не относительно начала файла.
2. См. структуру WININFO для более подробной информации об этих полях.

TBSEGMENT

Таблица сегментов содержит характеристики сегментов кода и данных программы. Она состоит из одного или нескольких элементов, формат которых показан на рис. 15.5.

Рис. 15.5. TBSEGMENT



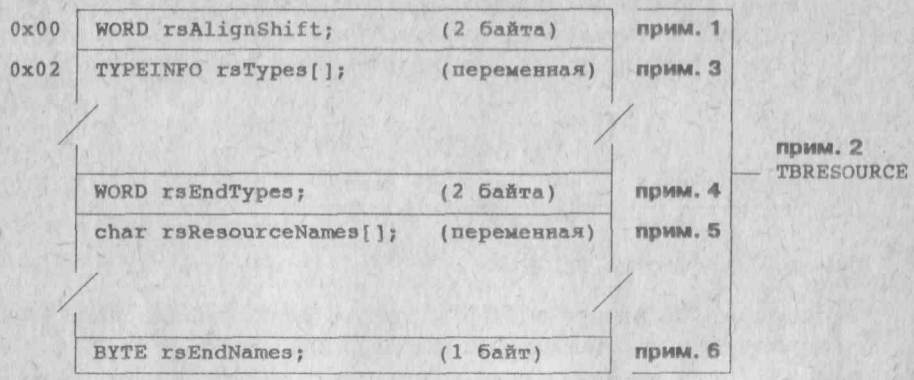
Примечания к рис. 15.5

1. Байтовые смещения в этой структуре даны относительно начала структуры, а не относительно начала файла.
2. См. структуру TBSEGMENT для более подробной информации об этих полях.

TBRESOURCE

Программы для Windows опираются на ресурсы — меню, графические управляющие кнопки, диалоговые окна и прочие элементы. Как правило, ресурсы создаются с помощью редактора ресурсов в текстовой форме, а затем компилируются компилятором ресурсов Microsoft RC в некоторое внутренне представление, сохраняемое в файле с расширением RES. Далее, двоичные образы ресурсов из этих файлов копируются компоновщиком в результирующий EXE-код наряду с каталогом, изображенным на рис. 15.6, где показаны типы ресурсов и их расположение.

Рис. 15.6. TBRESOURCE



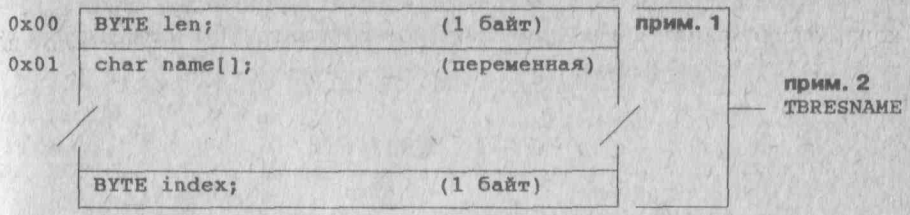
Примечания к рис. 15.6

1. Байтовые смещения в этой структуре даны относительно начала структуры, а не относительно начала файла.
2. См. структуру TBRESOURCE для более подробной информации об этих полях.
3. Это массив переменной длины структур TYPEINFO, описывающих каждый ресурс.
4. Число 0x00 в этом месте отмечает конец предшествующего массива переменной длины.
5. Содержит ноль или более имен, связанных с данным ресурсом. Отсутствует, если таких имен нет.
6. Число 0x00 в этом месте отмечает конец предшествующего массива переменной длины. Отсутствует, если предыдущий массив не существует.

TBRESNAME

В таблице резидентных имен перечислены все экспортируемые функции файла. Слово "резидентный" означает, что они никогда не удаляются из памяти. Элементы этой таблицы имеют формат, показанный на рис. 15.7.

Рис. 15.7. TBRESNAME



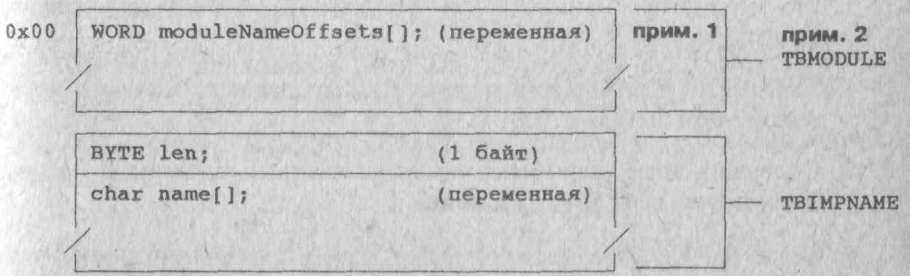
Примечания к рис. 15.7

1. Байтовые смещения в этой структуре даны относительно начала структуры, а не относительно начала файла.
2. См. структуру TBRESNAME для более подробной информации об этих полях.

TBMODULE

Таблица ссылок на модули, показанная на рис. 15.8, начинается с массива слов, содержащих смещения структур TBIMPNAME, также показанных в таблице.

Рис. 15.8. TBMODULE



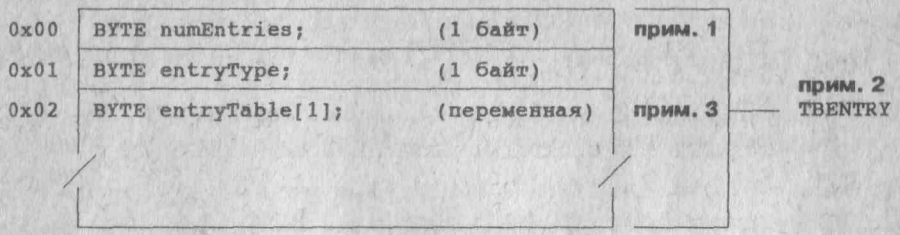
Примечания к рис. 15.8

1. Поскольку этот массив имеет переменную длину, байтовые смещения следующих элементов указать нельзя. Этот массив хранит ссылки на структуры типа TBIMPNAME, следующие за ним.
2. См. структуры TBMODULE и TBIMPNAME для более подробной информации об этих полях.

TBENTRY

Группы программных точек входа (которые также могут относится и к константам) хранятся в таблице точек входа. Эти группы называются связками. Каждая связка имеет формат, показанный на рис. 15.9.

Рис. 15.9. TBENTRY



Примечания к рис. 15.9

1. Байтовые смещения в этой структуре даны относительно начала структуры, а не относительно начала файла.
2. См. структуру TBENTRY для более подробной информации об этих полях.
3. Здесь хранятся одна или несколько структур типа MOVEABLEENTRY или типа FIXEDENTRY.

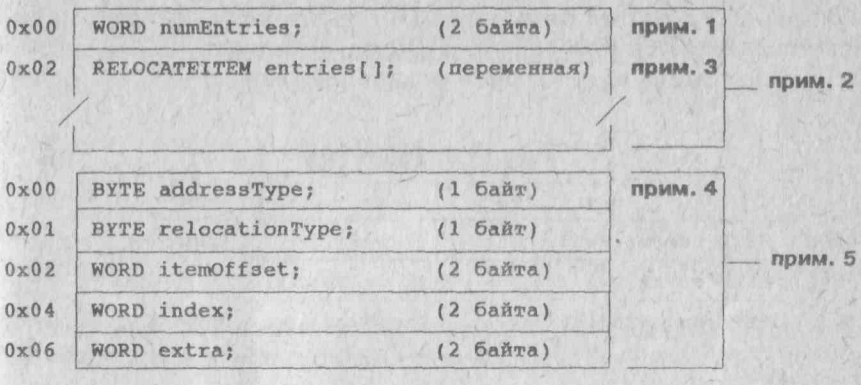
TBNONRESNAME

Таблица нерезидентных имен имеет тот же самый формат, что и таблица резидентных имен, но может быть сброшена на диск. См. рис. 15.7, где показан формат этой таблицы.

TBRELOCATIONTABLE

Сегменты кода и данных в EXE-файле могут содержать ссылки на другие сегменты, требующие модификации адресов при загрузке программы в память. На рис. 15.10 показан формат соответствующей настроечной таблицы.

Рис. 15.10. TBRELOCATIONTABLE



Примечания к рис. 15.10

1. Байтовые смещения в этой структуре даны относительно начала структуры, а не относительно начала файла.
2. См. структуру TBRELOCATIONTABLE для более подробной информации об этих полях.
3. Содержит один или несколько элементов типа RELOCATEITEM, показанных ниже.
4. Байтовые смещения в этой структуре даны относительно начала структуры, а не относительно начала файла.
5. См. структуру TBRELOCATEITEM для более подробной информации об этих полях.

Интерфейс с языком Си

Исполняемые файлы Windows имеют весьма сложную структуру и содержат множество данных различных типов, код, ресурсы и другую информацию. В этом разделе приводятся Си-структуры, описывающие формат EXE-файлов Windows.

Замечание. Некоторые листинги в этом разделе нумеруются иначе, чем другие. Например, 15.10a, 15.10b, 15.10c. Пронумерованные таким способом листинги связаны между собой специальным образом, что поясняется в комментариях под ними.

OLDHEADER

Если вы попытаетесь запустить Windows-программу в среде DOS, то получите сообщение вида:

```
"This program requires Microsoft Windows"
```

Вывод этого сообщения является результатом работы программы-заглушки, совместимой с DOS, с которой начинается всякий EXE-файл Windows. Программа-заглушка превращает EXE-файл Windows в обычную программу для MS-DOS, которая выводит указанное сообщение и завершается. Но для Windows этот же файл представляет собой полноценную прикладную программу.

В листинге 15.1 показан формат программы-заглушки, включая заголовок и указатель на начало кода и данных Windows-программы, хранящейся в EXE-файле.

Замечание. Ни одна из приводимых ниже структур не санкционирована официально фирмой Microsoft. Все идентификаторы, встречающиеся в этих структура, за исключением элементов EXEHEADER и некоторых других, выбраны произвольно. Прочие публикации по формату EXE-файлов Windows могут ссылаться на те же самые структурные единицы по другим идентификаторам.

Листинг 15.1. OLDHEADER

```
typedef struct tagOLDHEADER {
    EXEHEADER  msdosHeader;
    BYTE       breserved[28];
    WORD       winInfoOffset;
    WORD       wreserved;
    BYTE       msdosStub[1];
} OLDHEADER;
```

- msdosHeader — описывает различные характеристики программы-заглушки. Данная информация предназначена только для этой программы, но не для приложения Windows, содержащегося в файле. См. далее структуру EXEHEADER для более подробной информации об этом элементе.
- breserved — не используется и не документировано.
- winInfoOffset — байтовое смещение от начала файла до информационного заголовка Windows, показанного в виде структуры WINHEADER далее в этой главе.
- wreserved — не используется и не документировано.

- msdosStub — программа-заглушка для MS-DOS, выводящая сообщение "This program requires Microsoft Windows", если данный EXE-файл запускается в среде MS-DOS.

EXEHEADER

Структура EXEHEADER, показанная в листинге 15.2, начинает все EXE-файлы, предназначенные для работы в среде MS-DOS или в среде Windows. Все поля в этой структуре относятся к программе-заглушке, а не к приложению Windows.

Листинг 15.2. EXEHEADER

```
typedef struct tagEXEHEADER {
    WORD   exSignature;
    WORD   exExtraBytes;
    WORD   exPages;
    WORD   exRelocItems;
    WORD   exHeaderSize;
    WORD   exMinAlloc;
    WORD   exMaxAlloc;
    WORD   exInitSS;
    WORD   exInitSP;
    WORD   exCheckSum;
    WORD   exInitIP;
    WORD   exInitCS;
    WORD   exRelocTable;
    WORD   exOverlay;
    DWORD  reserved;   /* Нет в официальной структуре EXEHEADER */
} EXEHEADER;
```

- exSignature — если это поле содержит число 0x5A4D, данный файл является корректным EXE-файлом. Любое другое число в этом месте (первые два байта файла) говорит о том, что файл испорчен или имеет неверный формат.

> **Замечание.** Число 0x5A4D представляет собой два ASCII-символа — M и Z — инициалы разработчика (Mark Zbikowski).

- exExtraBytes — число байтов на последней странице, равное остатку от деления длины файла на 512 (размер одной страницы).
- exPages — число полных и частичных страниц в файле. Равно байтовому размеру файла, поделенному на 512, плюс 1, если был остаток от деления.
- exRelocItems — число указателей в настроечной таблице.

- exHeaderSize — число 16-байтовых параграфов в заголовке файла.
- exMinAlloc — необходимое количество памяти в 16-байтовых параграфах, требуемое дополнительно после загрузки программы.
- exMaxAlloc — максимальное количество памяти, которое может быть выделено программе.
- exInitSS — начальное значение перемещаемого адреса в регистре сегмента стека (SS). Изменяется в момент загрузки.
- exInitSP — начальное значение в регистре указателя стека.
- exCheckSum — поразрядная дополненная контрольная сумма всех 16-битовых слов файле, за исключением данного поля.
- exInitIP — начальное значение в регистре указателя команды (IP).
- exInitCS — начальное значение перемещаемого адреса в регистре сегмента кода (CS). Изменяется в момент загрузки.
- exOverlay — число оверлеев. Нуль для главного модуля (как это всегда бывает в случае программного файла для Windows).
- reserved — не используется и не документировано. Отсутствует в официальной структуре EXEHEADER.

WINHEADER

Если EXE-файл содержит приложение Windows, то поле winInfoOffset в структуре OLDHEADER указывает на структуру WINHEADER, показанную в листинге 15.3. Эта структура полностью описывает приложение Windows, включая код, данные и ресурсы.

Предупреждение. По причине своего большого размера и сложности структура WINHEADER, приведенная здесь, не может быть использована в программе непосредственно, а предназначена в качестве руководства по именам структурных единиц и порядке их следования в исполнительном заголовке.

Листинг 15.3. WINHEADER

```
typedef struct tagWINHEADER {
  WININFO       infoHeader;
  TBSEGMENT     segmentTable;
  TBRESOURCE    recourceTable;
  TBRESNAME     residentNameTable;
  TBMODULE      moduleRefTable;
  TBIMPNAME     importedNameTable;
  TBENTRY       entryTable;
  TBNONRESNAME nonResidentNameTable;
  BYTE          segments[1];
} WINHEADER;
```

- infoHeader — этот элемент по своему назначению сходен со структурой EXEHEADER для программы-заглушки. Содержит характеристики приложения Windows, хранящейся в EXE-файле.
- segmentTable — идущая вслед за информационным заголовком таблица, описывающая сегменты программы.
- recourceTable — таблица ресурсов, служащая каталогом ресурсов — меню, диалоговых панелей, курсоров и т.д., — хранимых в EXE-файле.
- residentNameTable — имена экспортируемых функций, запоминаемые как резидентные (постоянно находящиеся памяти) символьные строки.
- moduleRefTable — смещения имен модулей, хранящихся в следующей далее таблице импортируемых имен.
- importedNameTable — таблица символьных строк, содержащих имена программных модулей. Используйте смещения в таблице moduleRefTable адресации начала каждой строки.
- entryTable — коллекция программных точек входа.
- nonResidentNameTable — имена экспортируемых функций, запоминаемые как нерезидентные (могут сбрасываться на диск) символьные строки.
- segments — сегменты кода и данных приложения, а также настроечные данные.

WININFO

Часть EXE-файла, касающаяся Windows, начинается информационным заголовком, показанным в листинге 15.4 в виде структуры WININFO. Поля этой структуры хранят характеристики программы, а также смещения прочих таблиц в файле.

Замечание. Различные таблицы в EXE-файле Windows могут начинаться с любой байтовой позиции. Они не обязательно выровнены на границу сектора.

Листинг 15.4. WININFO

```
typedef struct tagWININFO {
    WORD    signature;
    BYTE    linkerVersion;
    BYTE    linkerRevision;
    WORD    entryTabOffset;
    WORD    entryTabLen;
    DWORD   reserved1;
    WORD    exeFlags;
    WORD    dataSegNum;
    WORD    localHeapSize;
    WORD    stackSize;
    DWORD   cs_ip;
    DWORD   ss_sp;
    WORD    segTabEntries;
    WORD    modTabEntries;
    WORD    nonResTabSize;
    WORD    segTabOffset;
    WORD    resTabOffset;
    WORD    resNameTabOffset;
    WORD    modTabOffset;
    WORD    impTabOffset;
    WORD    nonResTabOffset;
    WORD    reserved2;
    WORD    numEntryPoints;
    WORD    shiftCount;
    WORD    numResourceSegs;
    BYTE    targetOS;
    BYTE    miscFlags;
    WORD    fastLoadOffset;
    WORD    fastLoadSize;
    WORD    reserved3;
    BYTE    winRevision;
    BYTE    winVersion;
} WININFO;
```

- signature — содержит число 0x454E, т.е. два байта: 0x4E и 0x45 (именно в таком порядке). Это ASCII-коды букв N и E — аббревиатуры для "New Executable Code File" (Новый исполняемый программный файл).
- linkerVersion — номер версии компоновщика, создавшего данный EXE-файл.
- linkerRevision — модификация версии компоновщика, создавшего данный EXE-файл.

- entryTabOffset — байтовое смещение от начала заголовка WININFO до начала таблицы точек входа, показанной ниже в виде структуры TBENTRY.
- entryTabLen — байтовая длина таблицы точек входа. (см. TBENTRY).
- reserved1 — не используется и не документировано.
- exeFlags — флажки исполняемого кода, описанные в табл. 15.1.

Табл. 15.1.

Флажки исполняемого кода в поле exeFlags структуры WININFO

Бит	Маска	Описание
0	0x0001	Устанавливается компоновщиком для исполняемых файлов типа SINGLEDATA, имеющих только один сегмент данных. Всегда установлен для библиотек динамической связи (DLL), которые могут иметь лишь один сегмент данных.
1	0x0002	Устанавливается компоновщиком для исполняемых файлов типа MULTIPLEDATA, имеющих несколько сегментов данных. Как правило, устанавливается для большинства приложений Windows. Если биты 0 и 1 сброшены, подразумевается, что приложение имеет тип NOAUTODATA, где сегменты автоматических данных отсутствуют.
2	0x0004	Не документируется и не используется.
3	0x0008	Устанавливается для программы, которая будет выполняться в защищенном режиме (т.е. не в реальном и не в стандартном режимах).
4–7	0x00F0	Не документируется и не используется.
8	0x0100	Устанавливается для кода, несовместимого с библиотеками Windows для OS/2.
9	0x0200	Устанавливается для кода, совместимого с библиотеками Windows для OS/2.
10	0x0400	Не документируется и не используется.
11	0x0800	Устанавливается для самозагружающихся приложений, в которых начальный программный сегмент загружает остальную часть приложения.
12	0x1000	Не документируется и не используется.
13	0x2000	Устанавливается при обнаружении нефатальной ошибки во время компоновки.
14	0x4000	Устанавливается для библиотек, загружаемых выше кадра EMS.
15	0x8000	Устанавливается для библиотечных модулей. Если этот флажок установлен, Windows вызывает процедуру инициализации, адресуемую CS:SP. Если также установлен бит 0, регистр DS адресует библиотечный сегмент данных, в противном случае он адресует данные главной программы (программы, использующей эту библиотеку).

Замечание. Табл. 15.1 и другие ей подобные поясняют различные биты таких флажков, как exeFlags в структуре WININFO. В самой левой колонке даны номера битов, начиная с нуля для младшего бита в слове или байте. Во второй колонке показаны AND-маски, использующиеся для выделение данного бита. В правой колонке даются описания назначения каждого бита. Бит, помеченный как "не документируется и не используется", должен быть сброшен, если не указано иное.

- dataSegNum — число сегментов автоматических данных. Нуль, если биты 0 и 1 сброшены (программный файл типа NOAUTODATA).
- localHeapSize — начальный байтовый размер локальной динамической памяти. Нуль, если таковой не имеется.
- stackSize — начальный байтовый размер стека. Нуль для библиотечных модулей, в которых SS не равен DS.
- cs_ip — начальные значения для пары регистров CS:IP (IP в младшем слове).
- ss_sp — начальные значения для пары регистров SS:SP (SP в младшем слове).

Замечание. Значение регистра SS в поле ss_sp есть индекс в таблице сегментов, хранящейся в файле. Нулевое значение SP заставляет Windows инициализировать указатель стека адресом, равным сумме длин стека и сегмента автоматических данных.

- segTabEntries — число элементов в таблице сегментов (см. TBSEGMENT).
- modTabEntries — число элементов в таблице модулей (см. TBMODULE).
- nonResTabSize — размер в байтах таблицы нерезидентных имен.
- segTabOffset — байтовое смещение от начала структуры WININFO до таблицы сегментов (см. TBSEGMENT).
- resTabOffset — байтовое смещение от начала структуры WININFO до таблицы ресурсов (см. TBRESOURCE).
- resNameTabOffset — байтовое смещение от начала структуры WININFO до таблицы резидентных имен (см. TBRESNAME).
- modTabOffset — байтовое смещение от начала структуры WININFO до таблицы ссылок на модули (см. TBMODULE).
- impTabOffset — байтовое смещение от начала структуры WININFO до таблицы импортируемых имен (см. TBIMPNAME).

- nonResTabOffset — байтовое смещение от начала структуры WININFO до таблицы нерезидентных имен (см. TBNONRESNAME).
- reserved2 — не документируется и не используется.
- numEntryPoints — число перемещаемых точек входа.
- shiftCount — используется для выравнивания логического сектора. По умолчанию равняется 9, но обычно содержит число 4. Значение этого поля представляет собой $\log_2$ размера сегментного сектора. Устанавливается командой /Align:n компоновщика фирмы Microsoft или программой TLINK /A=n компании Borland, где n — сдвигаемое значение. Например, если n == 16, то shiftCount == 4 (2^4). Если n == 512, shiftCount == 9 (2^9) и т.д.
- numResourceSegs — число сегментов ресурсов.
- targetOS — флажки операционной системы, перечисленные в табл. 15.2.

Табл. 15.2.

Флажки операционной системы поля targetOS структуры WININFO

Бит	Маска	Описание
0	0x01	Неизвестная операционная система.
1	0x02	Microsoft OS/2 (помечен в некоторых публикациях как зарезервированный).
2	0x04	Microsoft Windows
3–7	0xF8	Не документируется и не используется.

- miscFlags — прочие флажки исполнимого кода, перечисленные в табл. 15.3.

Табл. 15.3.

Флажки поля MiscFlags структуры WININFO

Бит	Маска	Описание
0	0x01	Не документируется и не используется.
1	0x02	Файл приложения Windows 2.x, который может быть выполнен в защищенном режиме под Windows 3.x.
2	0x04	Файл приложения Windows 2.x, поддерживающего пропорциональные шрифты.
3	0x08	Файл имеет область быстрой загрузки (см. поля fastLoadOffset и fastLoadSize структуры WININFO).

- fastLoadOffset — смещение в 128-байтовых секторах области быстрой загрузки.
- fastLoadSize — размер в 128-байтовых секторах области быстрой загрузки.
- reserved3 — не документируется и не используется.
- winRevision — модификация версии Windows (1 для 3.1).
- winVersion — Версия Windows (3 для 3.1).

Замечание. Элементы winRevision и winVersion определяют минимальную версию Windows, ожидаемую приложением. Как правило, эти значения показывают самую раннюю версию Windows, с которой может работать данное приложение — обычно 3.0 для программ, совместимых с версиями 3.0 и 3.1.

Совет. Можно также рассматривать поля winRevision и winVersion как единое слово, в котором младший байт есть номер модификации, а старший — номер версии.

TBSEGMENT

Таблица сегментов в программном файле содержит информацию о каждом сегменте приложения. В листинге 15.5 эта таблица представлена в виде структуры TBSEGMENT языка Си.

Замечание. Поле segTabOffset структуры WININFO адресует первый элемент типа TBSEGMENT в файле. Поле segTabEntries задает число структур TBSEGMENT.

Листинг 15.5. TBSEGMENT

```
typedef struct tagTBSEGMENT {
    WORD  segDataOffset;
    WORD  segLen;
    WORD  segFlags;
    WORD  segMinSize;
} TBSEGMENT;
```

- segDataOffset — смещение в секторах от начала файла до сегмента данных. Нуль, если для этого элемента нет сегмента данных.
- segLen — длина сегмента в байтах. Нулевое значение представляет полный 64K сегмент.
- segFlags — сегментные флажки, перечисленные в табл. 15.4.

Табл. 15.4.

Сегментные флажки в поле segFlags структуры TBSEGMENT

Бит	Маска	Описание
0	0x0001	Устанавливается, если это сегмент данных. Сбрасывается для сегментов кода.
1	0x0002	Устанавливается, если загрузчик выделяет память для сегмента.
2	0x0004	Устанавливается, если сегмент загружен.
3	0x0008	Не документируется и не используется.
4	0x0010	Устанавливается, если сегмент имеет тип MOVEABLE. Сбрасывается, если сегмент имеет тип FIXED.
5	0x0020	Устанавливается, если сегмент имеет тип PURE или SHAREABLE. Сбрасывается, если сегмент имеет тип IMPURE или NONSHAREABLE.
6	0x0040	Устанавливается, если сегмент имеет тип PRELOAD. Сбрасывается, если сегмент имеет тип LOADONCALL.
7	0x0080	Если установлен для сегмента кода, сегмент имеет тип EXECUTEONLY. Если установлен для сегмента данных, сегмент имеет тип READONLY.
8	0x0100	Устанавливается, если сегмент имеет настроечные данные.
9	0x0200	Устанавливается, если сегмент согласующийся. Используется только OS/2. В соответствии с документацией по процессорам 80386 и 80486 фирмы Intel, согласующийся сегмент выполняется на том же уровне привилегий, что и вызывающий модуль. Несогласующийся сегмент может быть доступен только из второго кольца защиты.
10–11	0x0C00	Не документируется и не используется.
12	0x1000	Устанавливается, если сегмент может быть сброшен на диск.
13–15	0xE000	Не документируется и не используется.

- segMinSize — минимальное количество памяти, распределяемое сегменту. Число 0x0000 представляет полный 64K сегмент.

TBRESOURCE

Таблица ресурсов исполняемого файла служит каталогом ресурсов приложения. В листингах 15.6a, 15.6b и 15.6c показаны взаимосвязанные структуры TBRESOURCE, TYPEINFO и NAMEINFO в таблице ресурсов.

Замечание. Элемент resTabOffset структуры WININFO адресует первый элемент типа TBRESOURCE в файле.

Предупреждение. Структура TBRESOURCE содержит два вложенных элемента переменной длины и, следовательно, не может непосредственно применяться в программе, а показана здесь как справочник.

Листинг 15.6a. TBRESOURCE

```
typedef struct tagTBRESOURCE {
  WORD rsAlignShift;
  TYPEINFO rsTypes[1];
  WORD rsEndTypes;
  char rsResourceName[1];
  BYTE rsEndNames;
} TBRESOURCE;
```

- rsAlignShift — счетчик выравнивающих сдвигов. См. поле shiftCount структуры WININFO.
- rsTypes — массив структур типа TYPEINFO, по одной на каждый тип ресурса в файле. Структура TYPEINFO показана в листинге 15.6b.
- rsEndTypes — если нуль, то это поле отмечает конец таблицы ресурсов.

Предупреждение. Если поле rsEndTypes содержит нуль, следующие два элемента отсутствуют.

- rsResourceName — необязательный массив символьных строк, описывающих ресурсы в таблице. Первый байт каждой символьной строки содержит ее длину.
- rsEndNames — содержит нуль и тем самым отмечает конец массива rsResourceName, а следовательно, и конец таблицы ресурсов.

Листинг 15.6b. TYPEINFO

```
typedef struct tagTYPEINFO {
    WORD       rtTypeID;
    WORD       rtResourceCount;
    DWORD      rtReserved;
    NAMEINFO   rtNameInfo[1];
} TYPEINFO;
```

- rtTypeID — тип ресурса. Если старший бит слова установлен (другими словами, если rtTypeID AND 0x8000 равно 0x8000), то это поле за исключением старшего бита содержит константу из табл. 15.5. Если старший бит сброшен, то число в поле rtTypeID представляет собой смещение имени ресурса в массиве rsResourceName структуры TBRESOURCE.

Табл. 15.5.

Идентификаторы типов ресурсов, определенные в windows.h

Константа	Значение	Описание
RT_CURSOR	1	Курсор
RT_BITMAP	2	Растровое изображение
RT_ICON	3	Пиктограмма
RT_MENU	4	Меню
RT_DIALOG	5	Диалоговая панель, управляющая кнопка и т.п.
RT_STRING	6	Таблица символьных строк
RT_FONTDIR	7	Каталог шрифтов
RT_FONT	8	Шрифт
RT_ACCELERATOR	9	Горячая клавиша
RT_RCDATA	10	Пользовательский ресурс
RT_GROUP_CURSOR	12	Каталог курсоров
RT_GROUP_ICON	14	Каталог пиктограмм

Замечание. В гл. 16 описываются форматы ресурсов, хранящихся в EXE-файлах Windows, которые отличаются от прочих описываемых в этой книге файловых форматов. Например, файл растрового изображения похож на ресурс растрового изображения, хранящийся в EXE-файле, но не является его точной копией.

- rtResourceCount — число ресурсов определенного в rtTypeID типа.
- rtReserved — не используется и не документируется.
- rtNameInfo — массив структур типа NAMEINFO, которые содержат дополнительные характеристики ресурсов. Структура этого типа показана в листинге 15.6c.

Замечание. Значение поля rtResourceCount равно числу структур типа NAMEINFO, хранящихся в массиве rtNameInfo.

Листинг 15.6с. NAMEINFO

```
typedef struct tagNAMEINFO {
    WORD  rnOffset;
    WORD  rnLength;
    WORD  rnFlags;
    WORD  rnID;
    WORD  reserved[2];
} NAMEINFO;
```

- rnOffset — смещение в единицах выравнивания от начала файла до места расположения ресурсов. *Единица выравнивания* равна числу байтов, заданному в элементе rsAlignShift структуры TBRESOURCE.
- rnLength — байтовый размер ресурса.
- rnFlags — флажки ресурса, перечисленные в табл. 15.6.

Табл. 15.6.

Флажки ресурса в поле rnFlags структуры NAMEINFO

Бит	Маска	Описание
0-3	0x000F	Не документируется и не используется.
4	0x0010	Устанавливается, если ресурс может быть перемещен (MOVEABLE). Сбрасывается, если ресурс неперемещаемый (FIXED).
5	0x0020	Устанавливается для разделяемых (PURE) ресурсов.
6	0x0040	Устанавливается, если ресурс должен быть предварительно загруженным (PRELOAD). Сбрасывается, если ресурс загружается только по запросу.
7-15	0xFF80	Не документируется и не используется.

- rnID — целочисленный идентификатор ресурса, если старший бит этого поля установлен. Если же он сброшен, то это поле содержит байтовое смещение от начала таблицы ресурсов до символьной строки с именем ресурса.

> **Совет.** Все программирующие в среде Windows знают, что ресурсы могут идентифицироваться как целым числом, так и символьной строкой. Используйте поле rnID для выбора способа идентификации ресурса.

- reserved — эти два слова не документируются и не используются.

Замечание. Microsoft именует предыдущие два слова как rnHadle и rnUsage соответственно, но не сообщает об их назначении.

TBRESNAME

Таблица резидентных имен в EXE-файле содержит имена функций, экспортируемых файлом, т.е. тех функций, которые могут быть вызваны из другой программы. Таблица резидентных имен постоянно находится в памяти и никогда не сбрасывается на диск. Однако в 386 расширенном режиме Windows может переместить любую область памяти в виртуальное дисковое пространство. Таким образом, совсем необязательно, что на резидентную таблицу все время будет расточительно тратиться память.

В листинге 15.7 показана псевдоструктура TBRESNAME, описывающая каждую строку таблицы резидентных имен.

Замечание. Элемент resNameTabOffset структуры WININFO адресует таблицу резидентных имен.

Предупреждение. Структура TBRESNAME содержит массив переменной длины типа char и, следовательно, не может быть использована непосредственно для определения программных переменных.

Листинг 15.7. TBRESNAME

```
typedef struct tagTBRESNAME {
  BYTE len;
  char name[1];
  BYTE index; } TBRESNAME;
```

- len — число байтов в символьной цепочке, хранящейся в поле name. Содержит нуль в конце таблицы.

Замечание. Если len равно нулю, то следующие два элемента — name и index — не существуют.

- name — ASCII-строка с именем экспортируемой функции, где строчное и прописное начертания одной и той же буквы считаются различными символами. Не имеет оконечного нулевого байта.
- index — индекс в таблице точек входа, адресующей идентифицируемую данным элементом типа TBRESNAME функцию. См. элементы entryTabOffset и entryTabLen структуры WININFO, а также структуру TBENTRY далее в этой главе.

Замечание. Первое имя в таблице резидентных имен есть имя модуля.

TBMODULE

Таблица ссылок на модули есть просто список 16-разрядных смещений, адресующих в файле имена импортируемых модулей, хранящихся в форме структур типа TBIMPNAME. Формат этой таблицы показан в листинге 15.8 в виде структуры TBMODULE.

Замечание. Поле modTabEntries структуры WININFO определяет число элементов в таблице ссылок на модули. Поле modTabOffset структуры WININFO адресует эту таблицу.

Листинг 15.8. TBMODULE

```
typedef struct tagTBMODULE {
    WORD  moduleNameOffsets[1];
} TBMODULE;
```

- moduleNameOffsets — массив переменной длины, содержащий смещения структур типа TBIMPNAME, рассматриваемых в следующем разделе. Размер этого массива равен 2*modTabEntries.

TBIMPNAME

Таблица импортируемых модулей хранит имена модулей, используемых EXE-файлом. В листинге 15.9 показан формат элемента этой таблицы в виде Си-структуры TBIMPNAME.

Замечание. Поле impTabOffset структуры WININFO адресует первый элемент типа TBIMPNAME в файле.

Листинг 15.9. TBIMPNAME

```
typedef struct tagTBIMPNAME {
    BYTE  len;
    char  name[1];
} TBIMPNAME;
```

- len — число байтов в символьной цепочке, хранящейся в поле name. Содержит нуль в конце таблицы.

- name — ASCII-строка с именем импортируемой функции. Не имеет оконечного нулевого байта.

TBENTRY

Во время создания EXE-файла компоновщик строит таблицу точек входа и нумерует каждый ее элемент, начиная с 1. Эти номера весьма важны, т.к. дают возможность другим модулям ссылаться на точки входа просто с помощью целочисленных значений. Но это также означает, что компоновщик не может переупорядочивать точки входа. Тем не менее компоновщик пытается по возможности сжать таблицу.

Таблица точек входа состоит из элементов, называемых также связками, имеющих вид структуры TBENTRY, показанной в листинге 15.10a. В зависимости от типа точки входа массив entryTable этой структуры может содержат одну или более структур типа MOVEABLEENTRY (листинг 15.10b) или типа FIXEDENTRY (листинг 15.10c).

Замечание. Элемент entryTabOffset структуры WININFO адресует таблицу TBENTRY, а поле entryTabLen структуры WININFO содержит длину этой таблицы.

Листинг 15.10a. TBENTRY

```
typedef struct tagTBENTRY {
    BYTE  numEntries;
    BYTE  entryType;
    BYTE  entryTable[1];
} TBENTRY;
```

- numEntries — число элементов в данной связке. Нулевое значение сигнализирует о конце таблицы точек входа.
- entryType — тип элемента. Равен 0xFF для перемещаемых сегментов (MOVEABLEENTRY) или 0xFE, если элемент относится к константе, а не к программной точке входа. Любое другое значение представляет собой индекс сегмента.
- entryTable — одна или несколько структур типа MOVEABLEENTRY (листинг 15.10b) или типа FIXEDENTRY (листинг 15.10c).

Листинг 15.10b. MOVEABLEENTRY

```
typedef struct tagMOVEABLEENTRY {
    BYTE  entryFlags;
    WORD  int3f;
    BYTE  segmentNumber;
    WORD  segmentOffset;
} MOVEABLEENTRY;
```

- entryFlags — флажки точки входа, перечисленные в табл. 15.7.

Табл. 15.7.

Флажки точки входа в элементе entryFlags структуры MOVEABLEENTRY

Бит	Маска	Описание
0	0x0001	Устанавливается, если элемент экспортируется этим модулем.
1	0x0002	Устанавливается, если сегмент, соответствующий этой точке входа, использует сегмент разделяемых данных.
2	0x0004	Не документируется и не используется.
3–7	0x0F80	Если программа содержит межуровневые переходы в защищенном режиме, в битах 3—7 запоминается размер в 16-разрядных словах требуемого стека переходов.

Замечание. *Межуровневый переход* происходит, когда задача, работающая в одном из уровней защищенного режима, или кольце, вызывает задачу, работающую в другом кольце, через прерывание. Например, обращения к функциям виртуального таймерного устройства GetTickCount() и TimerCount() влекут за собой межуровневый переход, т.к. виртуальный таймер работает в кольце 0. Как правило, лучше избегать межуровневых переходов, которые, по данным фирмы Intel, длятся примерно в 100 раз дольше, чем дальние вызовы функций. К счастью, межуровневого перехода не происходит, когда вызывается подпрограмма из DLL. Однако это означает, что процессор не в состоянии предотвратить несанкционированный доступ к разделяемым библиотекам динамической связи — одна из причин, по которой защита памяти в Windows не так надежна, как должна быть в идеальной операционной системе.

- int3f — команда прерывания 0x3F.
- segmentNumber — порядковый номер сегмента.
- segmentOffset — смещение сегмента.

Листинг 15.10c. FIXEDENTRY

```
typedef struct tagFIXEDENTRY {
    BYTE  flags;
    WORD  segmentOffset;
} FIXEDENTRY;
```

flags — флажки точки входа, перечисленные в табл. 15.7.
segmentOffset — смещение сегмента.

TBNONRESNAME

Исполняемый файл может иметь таблицу нерезидентных имен экспортируемых функций, идентичную по форме и назначению таблице резидентных имен. Единственное различие состоит в том, что нерезидентная таблица может быть сброшена на диск, а затем возвращена обратно в память по запросу. Поскольку Windows может использовать механизм виртуальной памяти, нерезидентные таблицы не так полезны в смысле экономии памяти, как это было в ранних версиях Windows.

Замечание. Структура TBNONRESNAME идентична по формату структуре TBRESNAME в листинге 15.7. См. описания элементов этой структуры.

Листинг 15.11. TBNONRESNAME

```
typedef struct tagTBNONRESNAME {
    BYTE  len;
    char  name[1];
    BYTE  index;
} TBNONRESNAME;
```

Сегменты кода и данных

Последние элементы в EXE-файле Windows — сегменты кода и данных приложения. Поскольку фактические адреса сегментов неизвестны до момента выполнения программы, для дальних вызовов функций, которые выходят за пределы сегмента, требуется загрузчик, модифицирующий адреса в программе. Эту работу загрузчик выполняет с помощью информации из *настроечной таблицы*, хранящейся после сегментов кода или данных.

В листинге 15.12 показан формат настроечной таблицы в виде структуры RELOCATIONTABLE. Эта структура имеет 16-разрядное слово, содержащее число элементов типа RELOCATEITEM (см. листинг 15.13), хранящихся в таблице.

Листинг 15.12. RELOCATIONTABLE

```
typedef struct tagRELOCATIONTABLE {
    WORD numEntries;
    RELOCATEITEM entries[1];
} RELOCATIONTABLE;
```

- numEntries — элементов типа RELOCATEITEM, хранящихся в массиве entries.
- entries — массив переменной длины структур типа RELOCATEITEM.

Листинг 15.13. RELOCATEITEM

```
typedef struct tagRELOCATEITEM {
    BYTE  addressType;
    BYTE  relocationType;
    WORD  itemOffset;
    WORD  index;
    WORD  extra;
} RELOCATEITEM;
```

- addressType — тип адресного значения, на который влияет данный настроечный элемент. Имеет одно из значений, перечисленных в табл. 15.8.

Табл. 15.8.

Значения поля addressType структуры RELOCATEITEM

Значение	Описание
0x00	Смещение относится только к младшему байту
0x02	16-разрядный селектор сегмента
0x03	32-разрядный указатель вида сегмент:смещение
0x05	16-разрядное смещение
0x0B	48-разрядный указатель
0x0D	32-разрядное смещение

- relocationType — тип настроечного элемента. Имеет одно из значений, перечисленных в табл. 15.9.

Табл. 15.9.

Значения поля relocationType структуры RELOCATEITEM

Значение	Описание
0x00	Внутренняя ссылка
0x01	Ссылка на импортируемый порядковый номер
0x02	Ссылка на импортируемое имя
0x03	Ссылка на операционную систему

- itemOffset — смещение в сегменте до элемента, требующего настройки.
- index — равно индексу в таблице ссылок в случае импортируемых порядковых номеров или имен. Для внутренних ссылок на фиксированные сегменты младший байт этого слова содержит номер сегмента, а старший байт равен нулю. Для внутренних ссылок на перемещаемые сегменты это поле принимает значение 0x00FF.
- extra — для импортируемых порядковых номеров это слово равно порядковому номеру функции. Для импортируемых имен оно содержит смещение в таблице импортируемых имен. Для внутренних ссылок на фиксированные сегменты это слово равно смещению сегмента. Для внутренних ссылок на перемещаемые сегменты это слово равно порядковому номеру в таблице точек входа сегмента.

Глава 16

РЕСУРСЫ ИСПОЛНЯЕМОГО ФАЙЛА (.EXE)

Наряду с кодом, данными и другими элементами, EXE-файл в Windows содержит также таблицу ресурсов, рассматриваемую в этой главе. Эта таблица образует каталог ресурсов приложения в двоичной форме. В данной главе исследуются форматы двоичных ресурсов, которые не являются точной копией этих же данных, хранящихся отдельно. Например, данные в BMP-файле отличаются от соответствующего двоичного ресурса растрового изображения в EXE-файле.

Формат файла

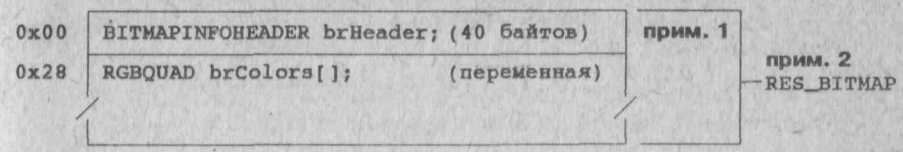
Как и в предыдущей главе, для описания организации двоичных ресурсов потребовалось множество структурных диаграмм. Сначала просмотрите иллюстрации, а затем все время возвращайтесь к ним, читая описания соответствующих Си-структур в разделе "Интерфейс с языком Си".

Замечание. Смещения во всех диаграммах этой главы даны относительно начала соответствующих структур, а не относительно начала файла. Все структуры адресуются элементами таблицы ресурсов программного файла, как показано в предыдущей главе.

Ресурсы растровых изображений

Ресурсы растровых изображений (рис. 16.1) похожи на BMP-файлы, но не имеют структуры BITMAPFILEHEADER. Кроме того, ресурсы растровых изображений лучше иметь маленькими, чтобы не раздувать EXE-файл. Как правило, большое растровое изображение следует хранить в BMP-файле.

Рис. 16.1. Ресурсы растровых изображений



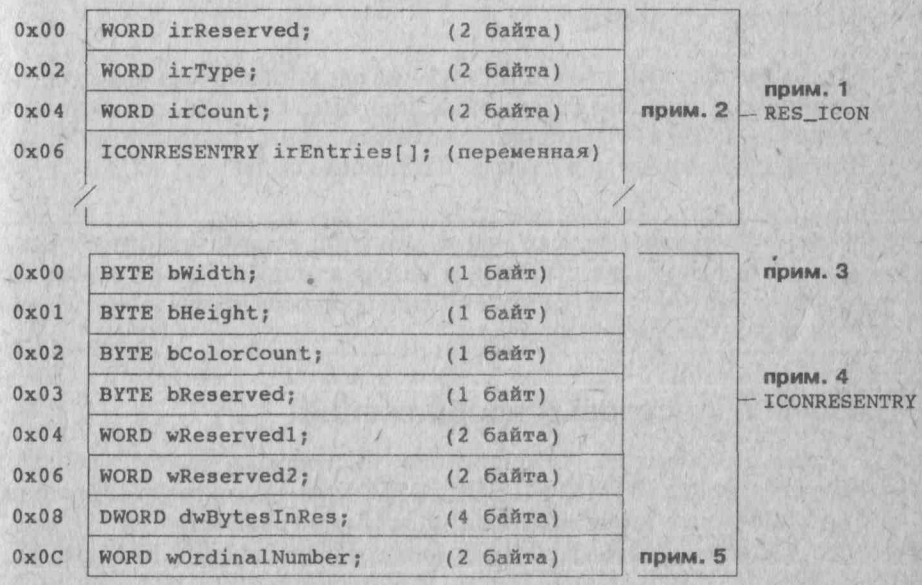
Примечания к рис. 16.1

1. См. заголовок BITMAPINFOHEADER и структуру RGBQUAD в гл. 4 для более подробной информации об этих полях.
2. См. RES_BITMAP далее в этой главе.

Ресурсы пиктограмм

Ресурсы пиктограмм состоят из каталога (RES_ICON), содержащего одну или несколько структур ICONRESENTRY, описывающих изображение каждой пиктограммы. Эти изображения идентичны хранящимся в ICO-файлах, рассматриваемых в гл. 5.

Рис. 16.2. Ресурсы пиктограмм



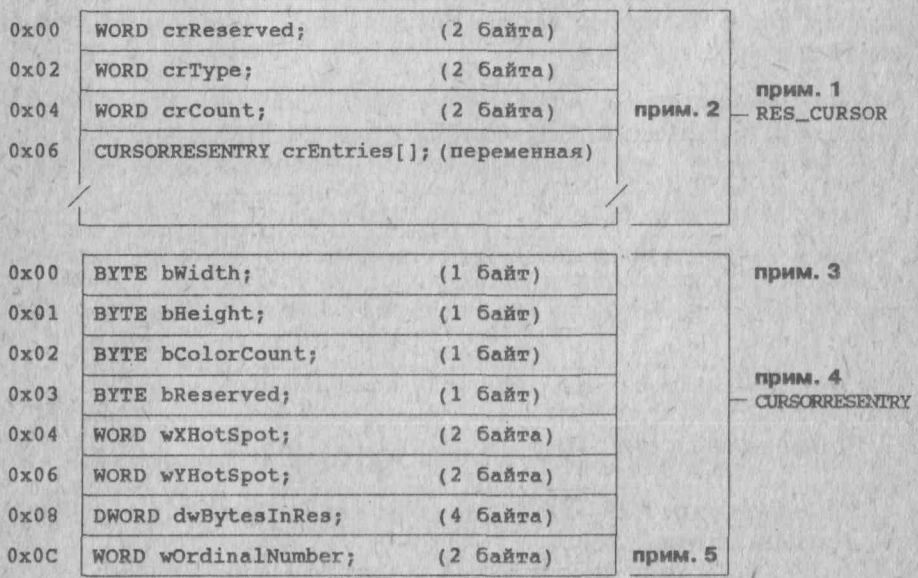
Примечания к рис. 16.2

1. См. структуру RES_ICON в данной главе для более подробной информации об этих полях.
2. Это поле определяет число структур ICONRESENTRY в идущем следом массиве irEntries.
3. Одна или несколько структур этого типа запоминается в массиве irEntries каталога пиктограмм.
4. См. структуру ICONRESENTRY в данной главе для более подробной информации об этих полях.
5. За исключением этого идентификатора, структура ICONRESENTRY идентична структуре ICONDIRENTRY, рассматриваемой в гл. 5.

Ресурсы курсоров

В качестве ресурсов курсоры и пиктограммы имеют похожие форматы. На рис. 16.3 показано, как ресурсы курсоров организованы в EXE-файле.

Рис. 16.3. Ресурсы курсоров



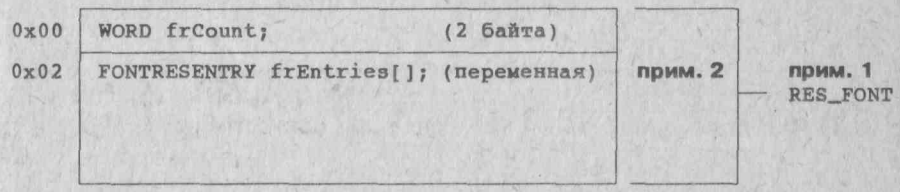
Примечания к рис. 16.3

1. См. структуру RES_CURSOR в данной главе для более подробной информации об этих полях.
2. Это поле определяет число структур CURSORRESENTRY в идущем следом массиве crEntries.
3. Одна или несколько структур этого типа запоминается в массиве crEntries каталога пиктограмм.
4. См. структуру CURSORRESENTRY в этой главе для более подробной информации об этих полях.
5. За исключением этого идентификатора структура CURSORRESENTRY идентична структуре CURSORDIRENTRY, рассматриваемой в гл. 6.

Шрифтовые ресурсы

Шрифтовые ресурсы состоят из каталога шрифтов, содержащего одну или несколько информационных структур. Поля этих структур похожи на соответствующие поля в шрифтовых файлах, рассматриваемых в гл. 7. Формат каталога шрифтов показан на рис. 16.4. На рис. 16.5 приводится информационная шрифтовая структура.

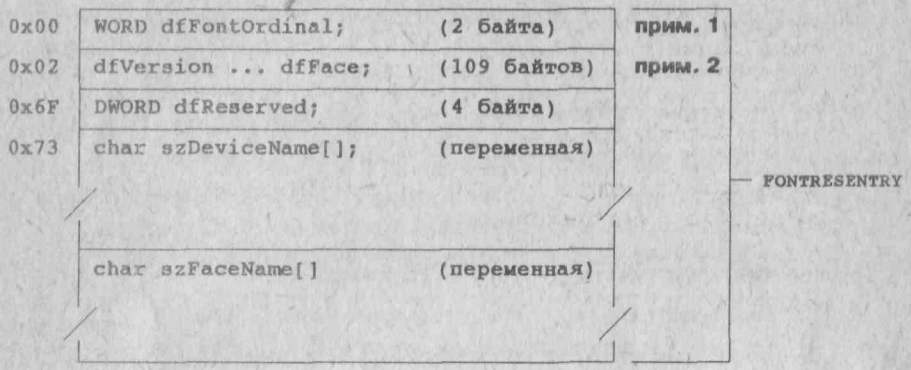
Рис. 16.4. Каталог шрифтовых ресурсов



Примечания к рис. 16.4

1. См. структуру RES_FONT в данной главе для более подробной информации об этих полях.
2. См. рис. 16.5.

Рис. 16.5. Информационная шрифтовая структура



Примечания к рис. 16.5

1. За исключением данного элемента, все поля этой структуры идентичны одноименным полям шрифтового файла, рассматриваемого в гл. 7.
2. См. гл. 7, а также структуру FONTRESENTRY в данной главе для более подробной информации об этих полях.

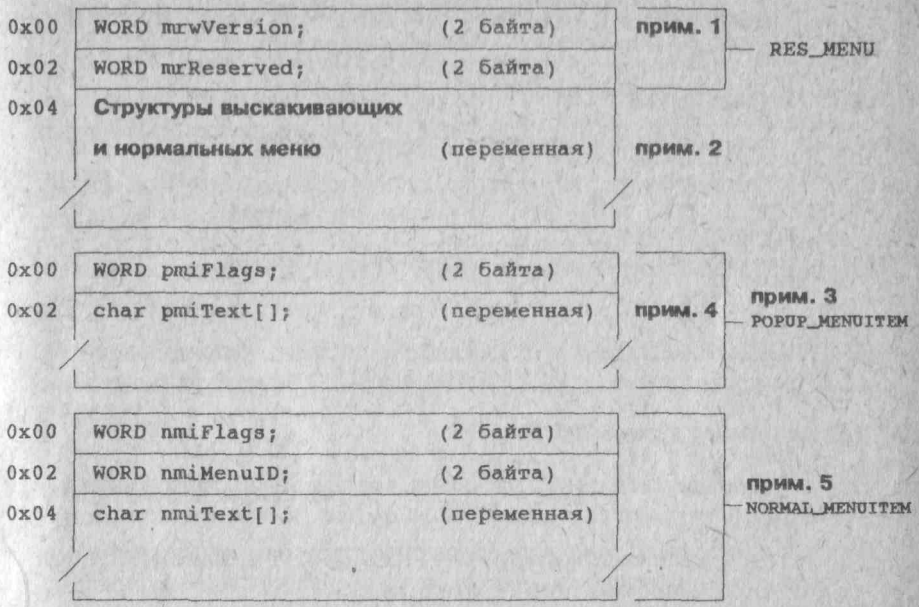
Ресурсы меню

Ресурс меню начинается с небольшого заголовка, за которым следуют один или несколько выскакивающих или нормальных элементов меню. На рис. 16.6 показаны три компонента ресурса меню.

Примечания к рис. 16.6

1. Это и следующее поля не используются и должны содержать нули. Очевидно, эти два слова включены для совместимости сверху вниз с ранними версиями Windows.
2. Один или несколько выскакивающих или нормальных элементов меню.
3. См. структуру POPUP_MENUITEM в данной главе для более подробной информации об этих полях и о том, как отличить эту структуру от NORMAL_MENUITEM.
4. См. структуру NORMAL_MENUITEM в данной главе для более подробной информации об этих полях.

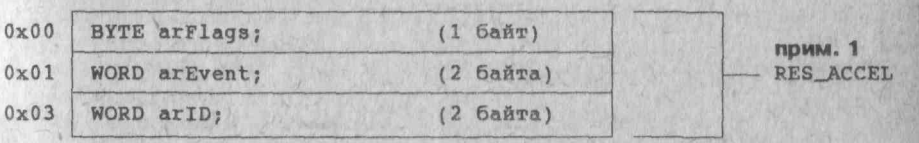
Рис. 16.6. Ресурс меню



Ресурсы ускорителей

Ресурс горячих клавишных комбинаций, или ускорителей (accelerator resource), состоит из списка структур, формат каждой из которых показан на рис. 16.7. При нажатии горячей комбинации клавиш Windows выдает сообщение WM_COMMAND, содержащее код из поля arEvent. Как правило, соответствующий элемент меню определяет тот же код события, связывая пункт меню и горячую клавишную комбинацию с одним и тем же программным действием.

Рис. 16.7. Ресурс ускорителя



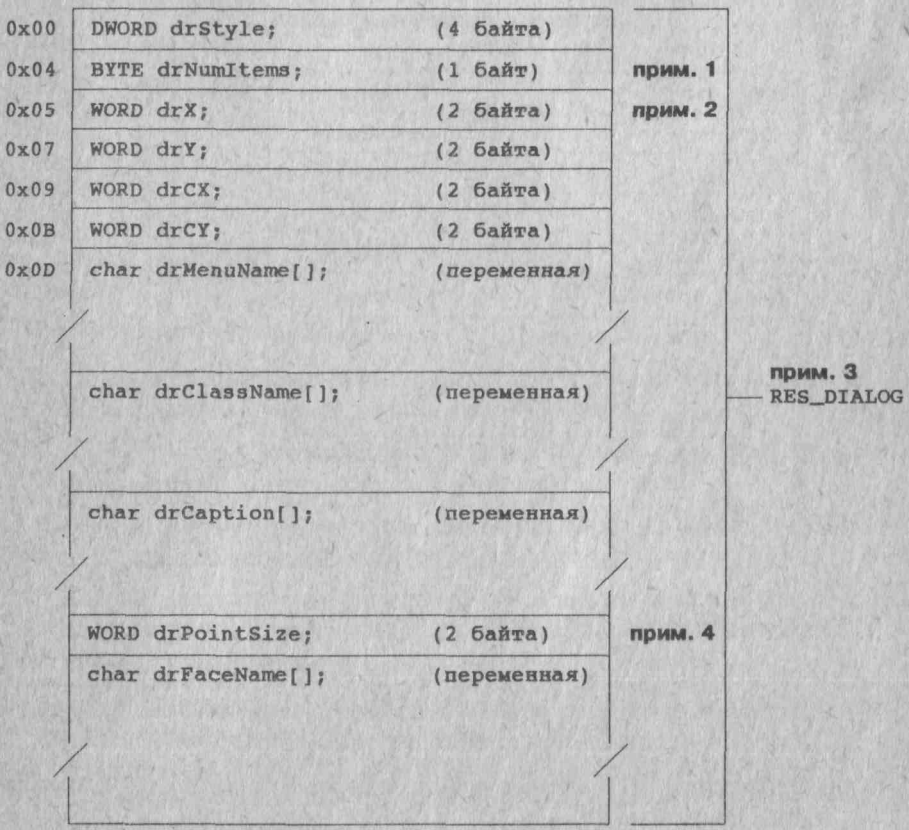
Примечания к рис. 16.7

1. Ресурс ускорителей состоит из одной или нескольких структур этого типа. См. структуру RES_ACCEL в данной главе для более подробной информации об этих полях.

Ресурсы диалоговых панелей

Каждый ресурс диалоговой панели в EXE-файле состоит из заголовка, показанного на рис. 16.8, и идущих следом структур, которые содержат описания графических элементов управления (controls), показанные на рис. 16.9. Типичное приложение имеет много диалоговых ресурсов, хранимых в этом формате.

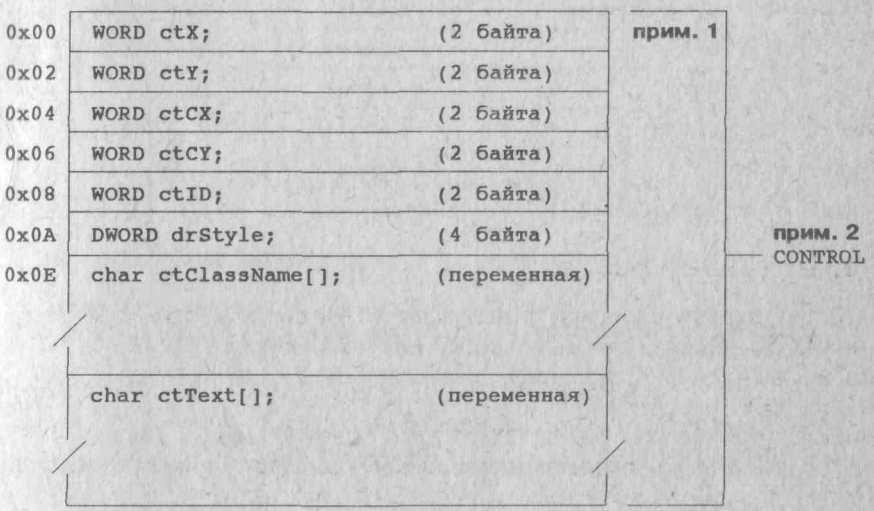
Рис. 16.8. Ресурс диалоговой панели



Примечания к рис. 16.8

1. Это поле показывает число структур, содержащих управляющие элементы (см. рис. 16.9), которые следуют за заголовком.
2. Координаты, ширина и высота выражаются в логических единицах, равных некоторой доле средней ширины или высоты символов шрифта, используемого в диалоговой панели. Если не указано иное, для этих вычислений используется системный шрифт.
3. См. структуру RES_DIALOG в данной главе для более подробной информации об этих полях.
4. Элементы drPointSize и drFaceName присутствуют только в том случае, если в поле drStyle установлен флажок DS_SETFONT.

Рис. 16.9. Структура для управляющих элементов в ресурсе диалоговой панели



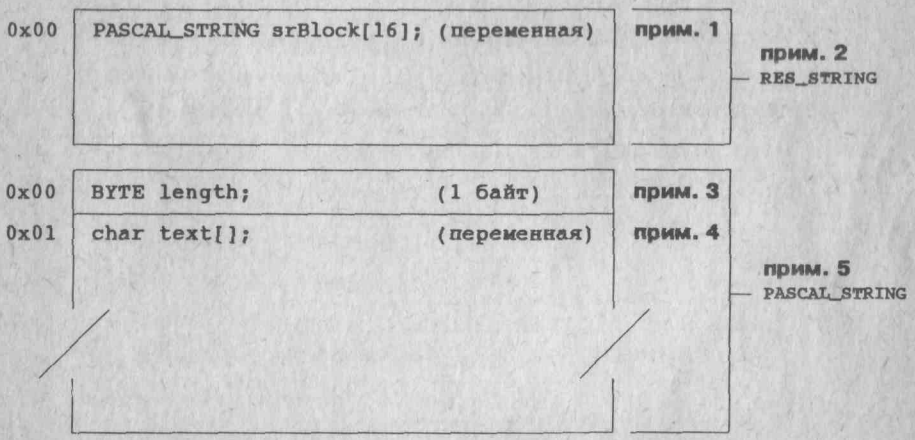
Примечания к рис. 16.9

1. Как и в заголовке, координаты, ширина и высота выражаются в логических единицах, равных некоторой доле средней ширины или высоты символов шрифта, используемого в диалоговой панели.
2. См. структуру CONTROL в данной главе для более подробной информации об этих полях.

Ресурсы таблиц символьных строк

Ресурсы таблиц символьных строк хранятся блоками по 16 строк, любая из которых может быть пустой. Все символьные строки имеют начальный байт, содержащий число символов в данной строке, и не имеют оконечного нуля.

Рис. 16.10. Ресурс таблицы символьных строк



Примечания к рис. 16.10

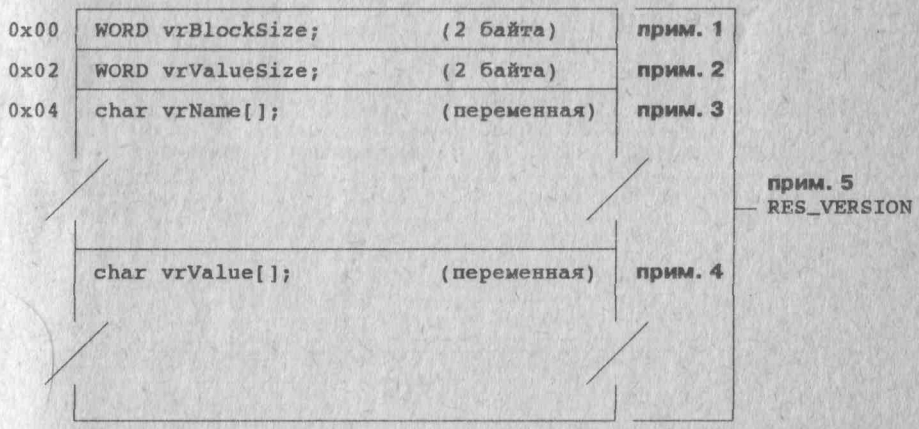
1. Минимальный размер блока равен 16 байтам, когда все строки пустые. Максимальный размер блока равен 4096 байтам.
2. См. структуру RES_STRING в данной главе для более подробной информации об этих полях и строковых ресурсах.
3. В этом байте хранится число символов, следующих за ним. Если это нуль, следующее далее текстовое поле отсутствует.
4. Поскольку длина символьной строки хранится в одном байте, максимальная длина текстового массива составляет 255 байтов. Оконечный нулевой байт отсутствует.
5. В некоторых версиях языка программирования Паскаль используются символьные строки, подобные показанным здесь. По этой причине символьные строки с префиксным байтом длины часто называются "паскалевскими". См. структуру PASCAL_STRING в данной главе для более подробной информации об этих полях.

Ресурсы версий

Ресурс версий, вероятно, наименее понятный из всех типов ресурсов Windows. Ресурс версий хранит дополнительную информацию о файле, авторские права, номер версии, код языка, номер набора символов и другие характеристики.

На рис. 16.11. Показан общий формат блоков ресурса версий, которые могут быть вложенными. Конкретные значения полей блока зависят от его типа.

Рис. 16.11. Блок ресурса версий

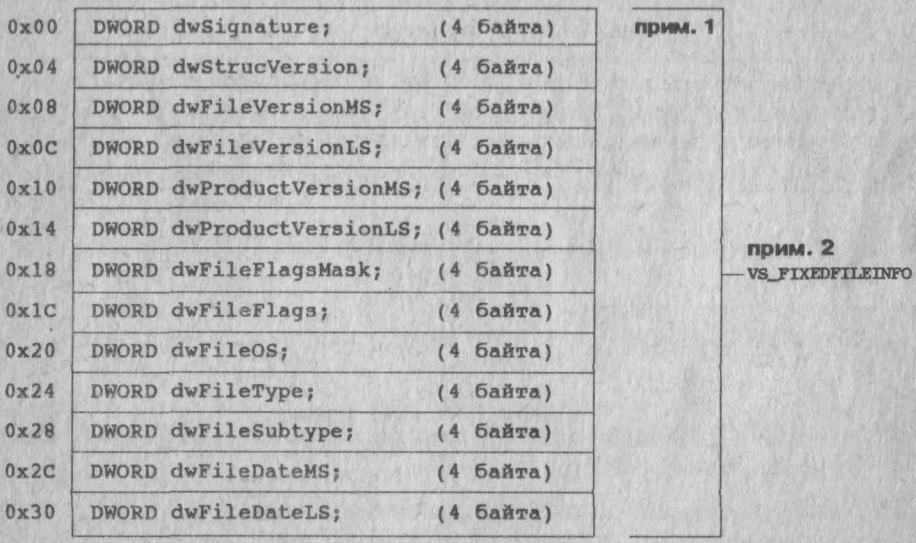


Примечания к рис. 16.11

1. В этом поле задается байтовый размер блока, включая все вложенные блоки.
2. В этом поле задается байтовый размер массива vrValue[].
3. Содержит имя блока, обычно идентифицирующее данный блок. Является символьной строкой переменной длины с нулевым байтом на конце.
4. Сущность содержимого блока зависит от его типа. Ресурс версий всегда начинается с корневого блока. См. рис. 16.12.
5. См. структуру RES_VERSION в данной главе для более подробной информации об этих полях и для примеров соответствующих текстовых описаний ресурса версий.

На рис 16.12 показан формат корневого блока ресурса версий. В этом блоке содержится общая информация о файле. Более подробная информация хранится в информационных блоках символьных строк, языка и в блоках переменной информации, рассматриваемые ближе к концу данной главы.

Рис. 16.12. Корневой блок ресурса версий



Примечания к рис. 16.12

1. Поля, заканчивающиеся буквами MS (наиболее значимая часть) и LS (наименее значимая часть), фактически являются 64-разрядными числами. Здесь они представлены двумя 32-разрядными значениями, т.к. в большинстве компиляторов Си отсутствуют 64-разрядный целый тип данных.

2. См. структуру VS_FIXEDFILEINFO в данной главе для более подробной информации об этих полях. Эта структура определена во включаемом файле ver.h, имеющемся во всех совместимых с Windows компиляторах языка Си и в большинстве других систем разработки программ.

Интерфейс с языком Си

Ресурсы растровых изображений (RES_BITMAP)

Ресурс растровых изображений почти идентичен соответствующему BMP-файлу, но не содержит структуру BITMAPFILEHEADER. В листинге 16.1 показан формат этого ресурса как структура RES_BITMAP на языке Си. Ресурс в этом формате требует экземпляра структуры типа TYPEINFO в таблице ресурсов с полем rtTypeID, равным RT_BITMAP.

Замечание. Имена всех типов структур, описывающих ресурсы, в данной главе имеют префикс RES_ из соображений единообразия. Однако они отличаются от имен, применяемых в других публикациях по форматам ресурсов.

Листинг 16.1. RES_BITMAP

```
typedef struct tagRES_BITMAP {
    BITMAPINFOHEADER  brHeader;
    RGBQUAD           brColors[1];
} RES_BITMAP;
```

- brHeader —информационный заголовок, описывающий формат растрового изображения, размер и т.д.
- brColors — массив цветов, используемых в изображении. Размер этого массива зависит от типа растрового изображения, хранимого в нем. Вслед за этим массивом идут AND- и XOR-маски (фон и передний план).

Замечание. См. гл. 4 для детальной информации о структурах BITMAPINFOHEADER и RGBQUAD.

Ресурсы пиктограмм (RES_ICON)

Ресурсы пиктограмм, описываемые структурами типа RES_ICON (листинг 16.2), похожи на каталоги пиктограмм типа ICONHEADER, рассматриваемые в гл. 5. Однако ресурс пиктограмм завершается массивом структур типа ICONRESENTRY (см. листинг 16.3), которые немного отличаются от структур типа ICONDIRENTRY, находящихся в ICO-файлах. Ресурс в этом формате требует экземпляра структуры типа TYPEINFO в таблице ресурсов с полем rtTypeID, равным RT_ICON.

Листинг 16.2. RES_ICON

```
typedef struct tagRES_ICON {
    WORD          irReserved;
    WORD          irType;
    WORD          irCount;
    ICONRESENTRY  irEntries[1];
} RES_ICON;
```

- irReserved — не используется и не документируется. Должно содержать 0x0000.
- irType — тип ресурса. Должен равняться 0x0001.
- irCount — число структур ICONRESENTRY в массиве irEntries.
- irEntries — массив структур ICONRESENTRY, по одной на каждую пиктограмму ресурса.

Структура ICONRESENTRY в листинге 16.3 весьма схожа со структурой ICONDIRENTRY из гл. 5. Отличие заключается в том, что вместо поля dwImageOffset структуры ICONDIRENTRY в ресурсе присутствует 16-разрядное поле wOrdinalNumber. Все прочие поля обеих структур совпадают.

Листинг 16.3. ICONRESENTRY

```
typedef struct tagICONRESENTRY {
    BYTE   bWidth;
    BYTE   bHeight;
    BYTE   bColorCount;
    BYTE   bReserved;
    WORD   wReserved1;  /* wplanes; */
    WORD   wReserved2;  /* wBitCount; */
    DWORD  dwBytesInRes;
    WORD   wOrdinalNumber;
} ICONRESENTRY;
```

- bWidth ... dwBytesInRes — те же поля, что и в структуре ICONDIRENTRY, рассматриваемой в гл. 5.
- wOrdinalNumber — уникальный порядковый номер данной пиктограммы, автоматически генерируемый редактором или компилятором ресурсов.

Замечание. См. также структуру ICONIMAGE в гл. 5 для более подробной информации о формате пиктограмм.

Ресурсы курсоров (RES_CURSOR)

За исключением двух моментов, ресурсы курсоров и пиктограмм являются идентичными. Во-первых, курсор бывает только монохромным, а пиктограмма может быть полноцветной. Во-вторых, курсор имеет горячую точку, а пиктограмма нет.

Ресурсы курсоров также похожи на курсоры, хранящиеся в CUR-файлах. Структура RES_CURSOR, показанная в листинге 16.4, сходна со структурой CURSORHEADER, рассматриваемой в гл. 6. Однако RES_CURSOR завершается массивом структур CURSORRES (листинг 16.5). Ресурс в этом формате требует экземпляра структуры типа TYPEINFO в таблице ресурсов с полем rtTypeID, равным RT_CURSOR.

Листинг 16.5. RES_CURSOR

```
typedef struct tagRES_CURSOR {
    WORD            crReserved;
    WORD            crType;
    WORD            crCount;
    CURSORRESENTRY  crEntries[1];
} RES_CURSOR;
```

- crReserved — не используется и не документируется. Должно содержать 0x0000.
- irType — тип ресурса. Должен равняться 0x0002.
- irCount — число структур CURSORRESENTRY в массиве irEntries.
- irEntries — массив структур CURSORRESENTRY, по одной на каждый курсор ресурса.

Структура CURSORRESENTRY в листинге 16.5 весьма схожа со структурой CURSORDIRENTRY из гл. 6. Отличие заключается в том, что вместо поля dwImageOffset структуры CURSORDIRENTRY в ресурсе присутствует 16-разрядное поле wOrdinalNumber. Все прочие поля обеих структур совпадают.

Листинг 16.3. CURSORRESENTRY

```
typedef struct tagCURSORRESENTRY {
    BYTE    bWidth;
    BYTE    bHeight;
    BYTE    bColorCount;
    BYTE    bReserved;
    WORD    wXHotSpot;
    WORD    wYHotSpot;
    DWORD   dwBytesInRes;
    WORD    wOrdinalNumber;
} CURSORRESENTRY;
```

- bWidth ... dwBytesInRes — те же поля, что и в структуре CURSORDIRENTRY, рассматриваемой в гл. 6.
- wOrdinalNumber — уникальный порядковый номер данного курсора, автоматически генерируемый редактором или компилятором ресурсов.

Замечание. См. также структуру CURSORIMAGE в гл. 5 для более подробной информации о формате курсоров.

Ресурсы шрифтов (RES_FONT)

Шрифтовые ресурсы хранятся в каталоге типа RES_FONT, показанного в листинге 16.6. Ресурс в этом формате требует экземпляра структуры типа TYPEINFO в таблице ресурсов с полем rtTypeID, равным RT_FONTDIR.

Листинг 16.6. RES_FONT

```
typedef struct tagRES_FONT {
    WORD          frCount;
    FONTRESENTRY  frEntries[1];
} RES_FONT;
```

- frCount — число структур типа FONTRESENTRY в массиве frEntries.
- frEntries — массив структур FONTRESENTRY, по одной на каждый шрифт в ресурсе.

Каждый шрифт описывается большой структурой FONTRESENTRY, показанной в листинге 16.7. Большинство ее полей идентичны описанным в гл. 7. Те элементы, которые имеет отличия, описаны следом за листингом. Ресурс в этом формате требует экземпляра структуры типа TYPEINFO в таблице ресурсов с полем rtTypeID, равным RT_FONT.

Листинг 16.7. FONTRESENTRY

```
typedef struct tagFONTRESENTRY {
    WORD   dfFontOrdinal;
    WORD   dfVersion;
    DWORD  dfSize;
    char   dfCopyright[60];
    WORD   dfType;
    WORD   dfPoints;
    WORD   dfVertRes;
```

```
    WORD    dfHorizRes;
    WORD    dfAscent;
    WORD    dfInternalLeading;
    WORD    dfExternalLeading;
    BYTE    dfItalic;
    BYTE    dfUnderline;
    BYTE    dfStrikeOut;
    WORD    dfWeight;
    BYTE    dfCharSet;
    WORD    dfPixWidth;
    WORD    dfPixHeight;
    BYTE    dfPitchAndFamily;
    WORD    dfAvgWidth;
    WORD    dfMaxWidth;
    BYTE    dfFirstChar;
    BYTE    dfLastChar;
    BYTE    dfDefaultChar;
    BYTE    dfBreakChar;
    WORD    dfWidthBytes;
    DWORD   dfDevice;
    DWORD   dfFace;
    DWORD   dfReserved;
/*    char    szDeviceName[1]; */
/*    char    szFaceName[1]; */
} FONTRESENTRY;
```

- dfFontOrdinal — уникальный порядковый номер, идентифицирующий данный шрифт. Генерируется редактором или компилятором ресурсов.
- dfVersion...dfFace — идентичны полям, описанным в гл. 7.
- dfReserved — не используется и не документируется.
- szDeviceName — ASCIIZ-строка, содержащая устройство, предназначенное для данного шрифта, если таковое имеется (например, шрифт, предназначенный только для принтера, может быть идентифицирован этим способом).
- szFaceName — ASCIIZ-строка, содержащая наименование гарнитуры шрифта. Например, Arial или Courier.

Замечание. Последние два поля структуры FONTRESENTRY показаны как комментарии, т.к. формально не принадлежат ей. Поскольку структуры языка Си не могут иметь поля переменной длины, элементы szDeviceName и szFaceName показаны здесь лишь для справки. Программы должны читать или записывать эти символьные строки индивидуально.

Ресурсы меню (RES_MENU)

Ресурс меню начинается с пустого заголовка типа RES_MENU, показанного в листинге 16.8. Очевидно, он включен для совместимости с ранними версиями Windows. Следом за заголовком идут дополнительные структуры, описывающие выскакивающие и нормальные компоненты меню. Ресурс в этом формате требует экземпляра структуры типа TYPEINFO в таблице ресурсов с полем rtTypeID, равным RT_MENU.

Листинг 16.8. RES_MENU

```
typedef struct tagRES_MENU {
    WORD  mrwVersion;
    WORD  mrReserved;
} RES_MENU;
```

- mrwVersion — номер версии меню. Не используется в Windows 3.0. Должен равняться 0x0000.
- mrReserved — не используется и не документируется. Должно содержать 0x0000.

За заголовком следуют одна или несколько структур типа POPUP_MENUITEM (*листинг* 16.9) или NORMAL_MENUITEM (листинг 16.10).

> **Совет.** Чтобы различить структуры POPUP_MENUITEM и NORMAL_MENUITEM, проверьте, установлен ли бит MF_POPUP в первом слове элемента меню. Если да, то это выскакивающий элемент меню, если нет — нормальный.

Листинг 16.9. POPUP_MENUITEM

```
typedef struct tagPOPUP_MENUITEM {
    WORD  pmiFlags;
    char  pmiText[1];
} POPUP_MENUITEM;
```

- pmiFlags — логическая ИЛИ-комбинация констант, определенных в windows.h и перечисленных здесь в табл. 16.1. За исключением MF_END, это те же самые константы, которые используются при создании ресурсов меню с помощью компилятора или редактора ресурсов.
- pmiText — ASCIIZ-строка, содержащая текст элемента меню в том виде, котором он появляется на экране.

Табл. 16.1.

Константы для поля pmiFlags структуры POPUP_MENUITEM

Константа	Значение	Назначение
MF_STRING	0x0000	Неотмеченный текст элемента меню. Является установкой по умолчанию.
MF_GRAYED	0x0001	Отображать элемент меню серым полутоном, показывая тем самым, что он в данный момент недоступен.
MF_DISABLED	0x0002	Отметить элемент меню как постоянно недоступный. Предназначается для использования во время разработки программ, когда некоторые команды, исполняемые по меню, еще не завершены.
MF_BITMAP	0x0004	Элемент меню является растровым изображением, а не текстом. В "серьезных" приложениях используется не так часто.
MF_CHECKED	0x0008	Отображать элемент меню помеченным "галочкой".
MF_POPUP	0x0010	Элемент является выскакивающим меню напротив своего имени, появляющемся на полоске меню в верхней части окна. Устанавливается для структур типа POPUP_MENUITEM и всегда сброшен в структурах типа NORMAL_MENUITEM.
MF_MENUBARBREAK	0x0020	Отображать данный элемент меню, начиная с новой колонки, вверх и вправо от выпадающего окна меню. Разделять команды меню вертикальной чертой.
MF_MENUBREAK	0x0040	То же самое, что и MF_MENUBARBREAK, но без вертикального разделителя команд меню.
MF_END	0x0080	Отмечает конец меню. Вставляется компилятором или редактором ресурсов. Никогда не используется непосредственно при создании меню приложения.
MF_OWNERDRAW	0x0100	Показывает, что окно меню должно появляться как реакция на сообщения WM_MEASUREITEM и WM_DRAWITEM и что программа ответственна за отображение этого элемента меню.
MF_SEPARATOR	0x0800	Отображать в окне меню как горизонтальную черту. Используется в качестве разделителя групп команд. Может использоваться только в выскакивающих меню.

Структура NORMAL_MENUITEM, показанная в листинге 16.10, почти идентична выскакивающему меню, но имеет дополнительное 16-разрядное поле nmiMenuID.

Листинг 16.10. NORMAL_MENUITEM

```
typedef struct tagNORMAL_MENUITEM {
    WORD  nmiFlags;
    WORD  nmiMenuID;
    char  nmiText[1];
} NORMAL_MENUITEM;
```

- *pmiFlags* — логическая ИЛИ-комбинация констант из табл. 16.1, *за исключением* MF_POPUP, разрешенной только для структур типа POPUP_MENUITEM.
- nmiMenuID — уникальный целочисленный идентификатор данного элемента меню. Его значение передается вместе с сообщением WM_COMMAND, когда элемент меню выбирается.
- nmiText — ASCIIZ-строка, содержащая текст элемента меню в том виде, котором он появляется на экране.

Замечание. Выскакивающие элементы меню (показанные на полоске меню в окне приложения) обычно содержат один или несколько нормальных элементов (показанные как команды внутри окна меню). Выскакивающие меню могут также содержать другие выскакивающие меню, которые, в свою очередь, могут содержать еще больше нормальных элементов. Такие вложенные структуры называются *каскадными меню*. Выбор каскадного меню вызывает появление другого выскакивающего меню, обычно с правой стороны. Каскадные меню могут иметь более глубоко вложенные меню, хотя два или три уровня вложенности наиболее часто употребимы на практике.

Ресурсы ускорителей (RES_ACCEL)

Ресурс горячих клавишных комбинаций, или ускорителей, состоит из одной или нескольких структур типа RES_ACCEL (см. листинг 16.11). Каждая структура этого типа определяет клавишную комбинацию, относящуюся к элементу меню. Нажатие этих клавиш имеет тот же эффект, что и выбор связанной с ней команды из выскакивающего меню. Ресурс в этом формате требует экземпляра структуры типа TYPEINFO в таблице ресурсов с полем rtTypeID, равным RT_ACCELERATOR.

Листинг 16.11. RES_ACCEL

```
typedef struct tagRES_ACCEL {
    BYTE  arFlags;
    WORD  arEvent;
    WORD  arID;
} RES_ACCEL;
```

- arFlags — логическая ИЛИ-комбинация флажков из табл. 16.2. Число 0x80 отмечает конец таблицы ускорителей.

Табл. 16.2.

Значения поля arFlags структуры RES_ACCEL

Значение	Описание
0x02	Не подсвечивать элемент меню, когда нажат ускоритель
0x04	Горячая клавиша сопровождается нажатием Shift
0x08	Горячая клавиша сопровождается нажатием Ctrl
0x10	Горячая клавиша сопровождается нажатием Alt
0x80	Метка конца таблицы ускорителей

- arEvent — код горячей клавиши, хранящийся как ASCII-символ или, в более общем случае, как код виртуальной клавиши.

Совет. Коды виртуальных клавиш определены в windows.h. Поищите там имена с префиксом VK_. Например, константа VK_ESCAPE представляет виртуальную клавишу для клавиши Esc.

- arID — уникальный целочисленный идентификатор для данного ускорителя. Передается вместе с сообщением WM_COMMAND, когда нажимается горячая клавишная комбинация.

Замечание. Горячие клавиши не связаны напрямую с командами меню. Они просто выдают некоторый идентификатор в сообщение WM_COMMAND и, следовательно, вызывают те же действия, что и команды меню, имеющие такой же идентификатор. Ускорители не требуют наличия соответствующих элементов меню. Они могут работать независимо.

Ресурсы диалоговых панелей (RES_DIALOG)

Ресурсы диалоговых панелей, или просто диалогов, являются одними из наиболее сложных в Windows. Каждая диалоговая панель состоит из заголовка, описывающего ее размер, расположение и т.д. Следом идет список графических элементов управления (controls). Формат заголовка диалога показан в листинге 16.12. Ресурс в этом формате требует экземпляра структуры типа TYPEINFO в таблице ресурсов с полем rtTypeID, равным RT_DIALOG.

Предупреждение. Структура RES_DIALOG содержит поля переменной длины и не может быт использована непосредственно для определения переменных. Используйте структуру, приведенную в следующем листинге как руководство для индивидуального чтения/записи элементов диалога.

Листинг 16.12. RES_DIALOG

```
typedef struct tagRES_DIALOG {
  DWORD drStyle;
  BYTE drNumItems;
  WORD drX;
  WORD drY;
  WORD drCX;
  WORD drCY;
  char drMenuName[1];
  char drClassName[1];
  char drCaption[1];
  WORD drPointSize;   // если drStyle включает константу DS_SETFONT
  char drFaceName[1]; // если drStyle включает константу DS_SETFONT
} RES_DIALOG;
```

- drStyle — логическая ИЛИ-комбинация одной или нескольких стилевых констант, таких, как DS_MODALFRAME или DS_ABSALIGN. Эти константы определены в windows.h и, как правило, выбираются в редакторе ресурсов.
- drNumItems — число графических управляющих элементов в данной диалоговой панели. Определяет количество структур CONTROL, следующих за заголовком RES_DIALOG.
- drX — горизонтальная координата диалогового окна, измеряемая в *горизонтальных диалоговых единицах*, равных четверти средней ширины символов используемого шрифта. Эта координата отсчитывается относительно левого края родительского окна, если drStyle не включает константу DS_ABSALIGN (тогда отсчет ведется относительно левого края экрана).

- drY — вертикальная координата диалогового окна, измеряемая в *вертикальных диалоговых единицах*, равных четверти средней высоты символов используемого шрифта. Подобно drX, эта координата отсчитывается относительно верхнего края родительского окна, если drStyle не включает константу DS_ABSALIGN.

Замечание. Горизонтальные и вертикальные диалоговые единицы вычисляются по системному шрифту, если drStyle не содержит флажок DS_SETFONT. Тогда единицы измерения будут вычисляться на основе шрифта, определяемого полями drPointSize и drFaceName. Поскольку все эти вычисления происходят в момент создания диалога, замена шрифта во время выполнения влечет за собой неправильное отображение диалоговой панели на экране.

- drCX — ширина диалогового окна в горизонтальных диалоговых единицах измерения, используемых в поле drX.
- drCY — ширина диалогового окна в вертикальных диалоговых единицах измерения, используемых в поле drY.
- drMenuName — содержит целочисленный или символьный идентификатор меню, если таковое имеется в диалоговой панели. В диалоге без меню это поле представляет собой единственный байт, содержащий 0x00. В диалоге с меню, идентифицированным целым числом, это поле состоит из трех байтов: 0xFF и двухбайтового идентификатора меню. В диалоге с меню, идентифицированным символьной строкой, это поле содержит ASCIIZ-строку.
- drClassName — обычно содержит единственный байт 0x00, показывающий, что для диалогового окна используется класс по умолчанию. В приложениях, использующих специальные классы диалоговых окон, это поле содержит ASCIIZ-строку с именем класса.
- drCaption — ASCIIZ-строка, содержащая заголовок диалогового окна. Может быть пустой, если таковой отсутствует.
- drPointSize — необязательное поле, содержащее размер шрифта, используемого в диалоге и определяемого следующим далее полем, в пунктах.
- drFaceName — необязательно присутствующая ASCIIZ-строка, содержащая наименование гарнитуры шрифта, используемого для всех текстов в графических элементах управления.

Предупреждение. Поля drPointSize и drFaceName присутствуют в структуре только в том случае, если в поле drStyle установлен флажок DS_SETFONT.

Следом за заголовком RES_DIALOG идут одна или несколько структур типа CONTROL (листинг 16.13), по одной на каждый графический элемент управления, принадлежащий данному диалогу.

Листинг 16.13. CONTROL

```
typedef struct tagCONTROL {
  WORD ctX;
  WORD ctY;
  WORD ctCX;
  WORD ctCY;
  WORD ctID;
  DWORD drStyle;
  char ctClassName[1];
  char ctText[1];
} CONTROL;
```

- ctX — горизонтальная координата левого верхнего угла элемента управления, измеряемая в горизонтальных диалоговых единицах, подобно используемым в поле drX.
- ctY — вертикальная координата левого верхнего угла элемента управления, измеряемая в вертикальных диалоговых единицах, подобно используемым в поле drY.
- ctCX — ширина графического элемента управления в горизонтальных диалоговых единицах измерения.
- ctCY — высота графического элемента управления в вертикальных диалоговых единицах измерения.
- ctID — уникальная константа, идентифицирующая графический элемент управления. Имеет значение 0xFFFF (-1) для неизменяемых текстов и других управляющих элементов, на которые программы никогда прямо не ссылаются.
- drStyle — логическая ИЛИ-комбинация таких стилевых констант, как WS_CHILD или WS_VISIBLE, определяемых в windows.h.
- ctClassName — обычно содержит единственный байт 0x00, показывающий, что для элемента управления используется класс по умолчанию. В приложениях, использующих специальные классы элементов управления, это поле содержит ASCIIZ-строку с именем класса.
- ctText — ASCIIZ-строка, содержащая текст элемента управления. Например, надпись на кнопке.

Ресурсы таблиц символьных строк (RES_STRING)

Хорошо написанная программа для Windows запоминает все сообщения об ошибках и прочую текстовую информацию в виде строковых ресурсов. Эти символьные строки могут редактироваться непосредственно в EXE-файле и таким образом переводиться на другой язык или модифицироваться без перекомпиляции самой программы.

Строковый ресурс состоит из блоков по 16 ASCII-строк, каждой из которых предшествует байт ее длины. Таким образом, эти строки не содержат оконечных нулей. Множество блоков строкового ресурса вместе составляют 65536 символьных строк, что намного больше любой разумной потребности приложения.

В листинге 16.14 показан формат блока строкового ресурса в виде Си-подобной псевдоструктуры RES_STRING. Каждый блок содержит 16 символьных строк, описываемых другой псевдоструктурой PASCAL_STRING, показанной в листинге 16.15. Ресурс в этом формате требует наличия в таблице ресурсов структуры TYPEINFO с полем rtTypeID, равным RT_STRING.

Предупреждение. Структуры RES_STRING и PASCAL_STRING показаны здесь только для справки. Поскольку некоторые символьные строки в таблице могут быть пустыми (в этом случае поле text структуры PASCAL_STRING отсутствует), эти структуры нельзя использовать для определения программных переменных. Вместо этого используйте приведенные ниже описания как справочник по индивидуальному чтению/записи строковых ресурсов.

Листинг 16.14. RES_STRING

```
typedef struct tagRES_STRING {
    PASCAL_STRING srBlock[16];
} RES_STRING;
```

- srBlock — ровно 16 объектов типа PASCAL_STRING, каждый из которых содержит длину строки и ASCII-текст.

Замечание. Во многих версиях компиляторов языка Паскаль для представления символьных строк используется формат, подобный описываемому структурой PASCAL_STRING. Этот формат отличается от традиционно используемых в Си символьных строк с оконечными нулевыми байтами.

Листинг 16.15. PASCAL_STRING

```
typedef struct tagPASCAL_STRING {
    BYTE  length;
    char  text[1];  /* отсутствует, если length == 0x00 */
} PASCAL_STRING;
```

- length — число из диапазона 0—255, представляющее количество символов в строке. Поскольку длина строки хранится в одном байте, ее максимальное значение составляет 255 символов.
- text — ASCII-текст символьной строки. Если length содержит 0x00, это поле не существует.

Совет. Пустые строки запоминаются в таблице строковых ресурсов в виде нулей. При создании ресурса вы можете сэкономить пространство в EXE-файле, не оставляя пустых промежутков в упорядоченной последовательности символьных строк.

Ресурсы версий (RES_VERSION)

Ресурс версий содержит символьные строки и другие элементы, идентифицирующие файл, включая отметки об авторских правах или, например, замечания о том, является ли данный файл дистрибутивной реализацией программы или же только отладочной версией для внутреннего использования. Информация о версии доступна с помощью динамической библиотеки VER.DLL, поставляемой вместе с Windows.

Информация в ресурсе версий хранится в виде вложенных блоков, имеющих один и тот же общий формат, показанный в листинге 16.16 как Си-подобная псевдоструктура RES_VERSION. Конкретное содержание этой структуры зависит от ее типа.

Предупреждение. Поскольку структура RES_VERSION содержит поле переменной длины (vrName), ее нельзя использовать для определения переменных в программе. Используйте данную псевдоструктуру как руководство для индивидуального чтения/записи полей ресурса.

Листинг 16.16. RES_VERSION

```
typedef struct tagRES_VERSION {
  WORD vrBlockSize;
  WORD vrValueSize;
  char vrName[1];
  BYTE vrValue[1];
} RES_VERSION;
```

- vrBlockSize — общий размер данного блока в байтах. Включает суммарный размер всех вложенных в него блоков.
- vrValueSize — байтовая длина массива vrValue. Если содержит 0x0000, то массив vrValue отсутствует (т.е. блок имеет лишь имя, но без связанных с ним данных).
- vrName — ASCIIZ-строка, содержащая имя блока. Дополняется нулями до границы двойного слова.
- vrValue — данные блока в форме ASCIIZ-строки или вложенный блок. Дополняется нулями до границы двойного слова.

Ресурс версий может иметь четыре вида блоков, формат каждого из которых соответствует структуре RES_VERSION:

- Корневой блок
- Блок переменной информации
- Блок строковой информации
- Блок языковой информации

Корневой блок

Первый блок в ресурсе версий называется *корневым*. Он хранится в рассмотренном выше формате, где *vrName содержит* символьную строку VS_VERSION_INFO. В этом блоке поле vrValue заполнено объектами типа VS_FIXEDFILEINFO (листинг 16.17).

Замечание. Структура VS_FIXEDFILEINFO определена в файле ver.h, поставляемого *со всеми* Windows-совместимыми компиляторами Си и с большинством других систем разработки программ. Ищите этот файл в подкаталоге INCLUDE, созданном во время инсталляции вашего компилятора. Как правило, этот файл хранится в одном каталоге с windows.h.

Листинг 16.17. VS_FIXEDFILEINFO

```
typedef struct tagVS_FIXEDFILEINFO {
  DWORD dwSignature;
  DWORD dwStrucVersion;
  DWORD dwFileVersionMS;
  DWORD dwFileVersionLS;
  DWORD dwProductVersionMS;
  DWORD dwProductVersionLS;
  DWORD dwFileFlagsMask;
  DWORD dwFileFlags;
  DWORD dwFileOS;
  DWORD dwFileType;
  DWORD dwFileSubtype;
  DWORD dwFileDateMS;
  DWORD dwFileDateLS;
} VS_FIXEDFILEINFO;
```

- dwSignature — 32-разрядное беззнаковое целое число 0xFEEF04BD, идентифицирующее корневой блок.

> **Замечание.** Согласно способу хранения слов и двойных слов в процессорах фирмы Intel это число представляется четырьмя байтами 0xBD, 0x04, 0xEF и 0xFE, следующих именно в таком порядке.

- dwStrucVersion — номер версии данной структуры, состоящий из двух компонент. Собственно номер версии хранится в старшем слове, а номер модификации версии — в младшем. Значение должно быть больше 0x29.
- dwFileVersionMS — номер версии файла.
- dwFileVersionLS — номер модификации версии файла.
- dwProductVersionMS — номер версии продукта.
- dwProductVersionLS — номер модификации версии продукта.

> **Замечание.** Поля dwFileVersionMS и dwFileVersionLS могут рассматриваться как старшая и младшая 32-разрядные половинки единого 64-разрядного номера версии. С полями dwProductVersionMS и dwProductVersionLS можно обращаться аналогично. MS образовано от "most significant" (наиболее значащий), а LS — от "least significant" (наименее значащий). Поскольку в большинстве компиляторов Си отсутствует 64-разрядный целочисленный тип данных, номер версии обычно читается или записывается 32-разрядными полями.

- dwFileFlagsMask — содержит биты, которые, будучи установленными, активизируют соответствующие флажки в поле dwFileFlags. Все сброшенные биты маски считаются неиспользуемыми в dwFileFlags. В настоящее время маска должна содержать число 0x3FL.

- dwFileFlags — логическая ИЛИ-комбинация одной или нескольких констант, перечисленных в табл. 16.3. Эти константы определены в файле ver.h, поставляемом вместе с большинством систем разработки программ для Windows и с компиляторами языка Си.

Табл. 16.3.

Константы для поля dwFileFlags структуры VS_FIXEDFILEINFO

Константа	Значение	Назначение
VS_FF_DEBUG	0x00000001L	Код содержит отладочную информацию или другие подобные детали.
VS_FF_PRERELEASE	0x00000002L	Приложение находится в процессе разработки и не предназначено для широкого распространения.
VS_FF_PATCHED	0x00000004L	Тиражируемое приложение, которое было обновлено.
VS_FF_PRIVATEBUILD	0x00000008L	Версия приложения, предназначенная только для внутреннего использования. Возможно, откомпилированная с нестандартными опциями — например, для тестирования. Использование данного значения требует соответствующего блока строковой информации со строкой PrivateBuild.
VS_FF_INFOINFERRED	0x00000010L	Показывает, что некоторая информация в ресурсе версий не полна или отсутствует.
VS_FF_SPECIALBUILD	0x00000020L	Приложение откомпилировано со стандартными опциями, но с одной или несколькими специальными возможностями. Использование данного значения требует соответствующего блока строковой информации со строкой SpecialBuild.

- dwFileOS — целевая операционная система. Это поле может содержать одну из констант, перечисленных в табл. 16.4. Эти константы определены в файле ver.h, поставляемом вместе с большинством систем разработки программ для Windows и с компиляторами языка Си.

Табл. 16.4.

Константы для поля dwFileOS структуры VS_FIXEDFILEINFO

Константа	Значение	Назначение
VOS_UNKNOWN	0x00000000L	Неизвестная операционная система
VOS_DOS	0x00010000L	MS-DOS
VOS_OS216	0x00020000L	16-битовая OS/2
VOS_OS232	0x00030000L	32-битовая OS/2
VOS_NT	0x00040000L	Windows NT
VOS_WINDOWS16	0x00000001L	16-битовая Windows
VOS_PM16	0x00000002L	16-битовый Presentation Manager
VOS_PM32	0x00000003L	32-битовый Presentation Manager
VOS_WINDOWS32	0x00000004L	32-битовая Windows
VOS_DOS_WINDOWS16	0x00010001L	Windows 3.0 или более поздняя, запущенная под MS-DOS
VOS_DOS_WINDOWS32	0x00010004L	32-битовая Windows, запущенная под MS-DOS
VOS_OS216_PM16	0x00020002L	16-битовый Presentation Manager, запущенный под 16-битовой OS/2
VOS_OS232_PM32	0x00030003L	32-битовый Presentation Manager, запущенный под 32-битовой OS/2
VOS_NT_WINDOWS32	0x00040004L	32-битовая Windows, запущенная под Windows NT

- dwFileType — категория файла, или тип. Это поле может содержать одну из констант, перечисленных в табл. 16.5 и определенных в файле ver.h. Неиспользуемые значения зарезервированы специально фирмой Microsoft.

Табл. 16.5.

Константы для поля dwFileType структуры VS_FIXEDFILEINFO

Константа	Значение	Категория
VFT_UNKNOWN	0x00000000L	Неизвестная
VFT_APP	0x00000001L	Приложение конечного пользователя
VFT_DLL	0x00000002L	Библиотека динамической связи
VFT_DRV	0x00000003L	Драйвер устройства, дополнительно типизируемый значением поля dwFileSubType

Табл. 16.5. (продолжение)

Константа	Значение	Категория
VFT_FONT	0x00000004L	Шрифт, дополнительно типизируемый значением поля dwFileSubType
VFT_VXD	0x00000005L	Драйвер виртуального устройства
VFT_STSTIC_LIB	0x00000007L	Библиотека статической связи

- dwFileSubType — назначение файла или дополнительная типизация его назначения. Если dwFileType содержит константу VFT_FONT, dwFileSubType может равняться одному из значений, перечисленных в табл. 16.6. Если dwFileType равняется VFT_VXD, то поле dwFileSubType содержит идентификатор драйвера виртуального устройства, скопированный из его управляющего блока. Если dwFileType содержит константу VFT_DRV, dwFileSubType может равняться одному из значений, перечисленных в табл. 16.7. Все константы из табл. 16.6 и 16.7 определены в файле ver.h.

Замечание. Поскольку значения констант дублируются в таблицах 16.5 и 16.6, смысл dwFileSubType всегда зависит от значения dwFileType.

Табл. 16.6.

Константы для поля dwFileSubType структуры VS_FIXEDFILEINFO, когда dwFileType равняется VFT_FONT

Константа	Значение	Подкатегория
VFT2_UNKNOWN	0x00000000L	Неизвестная
VFT2_FONT_RASTER	0x00000001L	Растровый шрифт
VFT2_FONT_VECTOR	0x00000002L	Векторный шрифт
VFT2_FONT_TRUETYPE	0x00000003L	Шрифт TrueType (WYSIWYG)

Табл. 16.7.

Константы для поля dwFileSubType структуры VS_FIXEDFILEINFO, когда dwFileType равняется VFT_FONT

Константа	Значение	Подкатегория
VFT2_UNKNOWN	0x00000000L	Неизвестная
VFT2_DRV_PRINTER	0x00000001L	Драйвер принтера

Табл. 16.7. (продолжение)

Константа	Значение	Подкатегория
VFT2_DRV_KEYBOARD	0x00000002L	Драйвер клавиатуры
VFT2_DRV_LANGUAGE	0x00000003L	Особый языковый драйвер
VFT2_DRV_DISPLAY	0x00000004L	Дисплейный драйвер
VFT2_DRV_MOUSE	0x00000005L	Драйвер мыши
VFT2_DRV_NETWORK	0x00000006L	Драйвер сети
VFT2_DRV_SYSTEM	0x00000007L	Системный драйвер общего назначения
VFT2_DRV_INSTALLABLE	0x00000008L	Устанавливаемый драйвер
VFT2_DRV_SOUND	0x00000009L	Звуковой (мультимедиа) драйвер
VFT2_DRV_COMM	0x0000000AL	Драйвер последовательной связи

- dwFileDateMS — старшая 32-разрядная часть метки даты и времени создания файла.
- dwFileDateLS — младшая 32-разрядная часть метки даты и времени создания файла.

Блок переменной информации

Блок переменной информации хранится в формате, показанном в листинге 16.16, как структура RES_VERSION. Этот блок содержит код языка из табл. 16.8 и код набора символов из табл. 16.9.

Табл. 16.8.

Коды языка в блоке переменной информации

Язык	Код
Албанский	0x041C
Арабский	0x0401
Бахаса	0x0421
Бельгийский голландский	0x0813
Бельгийский французский	0x0813
Бразильский португальский	0x0416
Болгарский	0x0402
Канадский французский	0x0C0C

Табл. 16.8. (продолжение)

Язык	Код
Кастильский испанский	0x040A
Каталонский	0x0403
Хорвато-сербский (латынь)	0x041A
Чешский	0x0405
Датский	0x0406
Голландский	0x0413
Финский	0x040B
Французский	0x040C
Немецкий	0x0407
Греческий	0x0408
Иврит	0x040D
Венгерский	0x040E
Исландский	0x040F
Итальянский	0x0410
Японский	0x0411
Корейский	0x0412
Мексиканский испанский	0x080A
Норвежский (Nynorsk)	0x0814
Норвежский (Bokmal)	0x0414
Польский	0x0415
Португальский	0x0816
Ретро-романский	0x0417
Румынский	0x0418
Русский	0x0419
Сербо-хорватский (кириллица)	0x081A
Китайский упрощенный	0x0804
Словацкий	0x041B
Шведский	0x041D
Швейцарский французский	0x100C
Швейцарский немецкий	0x0807C
Швейцарский итальянский	0x0810C
Тайский	0x041E

Табл. 16.8. (продолжение)

Язык	Код
Китайский традиционный	0x0404
Турецкий	0x041F
Британский английский	0x0809
Американский английский	0x0409
Урду	0x0420

Табл. 16.9.

Коды символьных наборов в блоке переменной информации

Символьный набор	Код
7-разрядный ASCII	0
Уникод	0x04B0
Windows—арабский	0x04E8
Windows—кириллица	0x04E3
Windows—греческий	0x04E5
Windows—иврит	0x04E7
Windows—японский (Shift — JIS X-0208)	0x03A4
Windows—корейский (Shift — KSC 5601)	0x03B5
Windows—восточноевропейский	0x04E2
Windows—многоязычный	0x04E4
Windows—Тайвань (GB5)	0x3B6
Windows—турецкий	0x04E6

В листинге 16.18 показан фрагмент текстового описания ресурса версий, содержащий блок переменной информации. Блок начинается ключевым словом BLOCK и идентифицируется символьными строками VarFileInfo и Translation. Целые числа соответствуют шестнадцатеричному коду языка 0x0409 (американский английский) и символьному набору 0x04E4 (Windows—многоязычный).

Листинг 16.18. Блок переменной информации ресурса версий (фрагмент)

```
...
BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 1033, 1252
  END
...
```

Блоки строковой информации

Блок строковой информации хранится в формате структуры RES_VERSION, показанной в листинге 16.16. За ним следуют один или несколько вложенных блоков языковой информации, описываемые в следующем разделе. Блок строковой информации определяет код языка из табл. 16.7 и символьный набор из табл. 16.8 в форме ASCII-строки.

В листинге 16.19 показан фрагмент блока строковой информации ресурса версий. Этот блок идентифицируется символьной строкой StringFileInfo. Коды языка и набора символов даны в строке 040904E4 в шестнадцатеричной форме. Пропущенные места (...) обычно содержат языковый информационный блок. См. Следующий раздел для более полного примера.

Листинг 16.19. Блок строковой информации ресурса версий (фрагмент)

```
...
BLOK "StringFileInfo"
BEGIN
  BLOCK "040904E4"
  BEGIN
    ...
  END
END
...
```

Блоки языковой информации

Блоки языковой информации содержат различные атрибуты файлов, знаки авторского права и т.д. Они вложены в блок строковой информации. Как и все ресурсы версий, блок языковой информации хранится в формате структуры RES_VERSION, показанной в листинге 16.16.

В листинге 16.20 показан пример блока языковой информации, вложенного в блок строковой информации, который определяет коды языка и символьного набора.

Листинг 16.20. Блок языковой информации ресурса версий (фрагмент)

```
...
BLOK "StringFileInfo"
BEGIN
  BLOCK "040904E4"
  BEGIN
    VALUE "CompanyName", "Swan Software\000\000"
    VALUE "FileDescription", "Sample Application\000"
    VALUE "FileVersion", "4.75\000\000"
    VALUE "InternalName", "SampleApp\000"
    VALUE "LegalCopyright", "Copyright \251 Tom Swan1993\000\000"
    VALUE "OriginalFileName", "SAMAPP.EXE\000"
    VALUE "ProductName", "Windows File Facts\000\000"
    VALUE "ProductVersion", 1st Edition\000\000"
  END
END
...
```

Совет. Такие элементы описания, как CompanyName, содержащий строку Swan Software в листинге 16.20, может содержать более одной символьной строки, каждая из которых заканчивается нулевым байтом. Чтобы вставить дополнительную пустую строку, наберите два нуля (\000\000). Значок копирайта представляется кодом \251 в строке языковой информации LegalCopyright.

В табл. 16.10 перечислены все строки языковой информации, которые могут появляться в блоке языковой информации, а также их типичные употребления. Фактическое использование, формат и текст каждой символьной строки выбирается по вашему усмотрению. Строки, идентификаторы которых в таблице помечены плюсами, являются необходимыми, остальные — необязательными.

Табл. 16.10.

Строки языковой информации

Идентификатор строки	Наличие	Типичное использование
Comments		Диагностические сообщения
CompanyName	+	Ваше имя или название вашей компании
FileDescription	+	Установочная метка
FileVersion	+	Номер версии файла
InternalName	+	Имя файла модуля или библиотеки
LegalCopyright		Знак авторского права

Табл. 16.10. (продолжение)

Идентификатор строки	Наличие	Типичное использование
LegalTrademarks		Торговая марка
OriginalFilename	+	Имя исполняемого файла
PrivateBuild		Внутренняя или личная реализация
ProductName	+	Имя программы для элемента меню About
ProductVersion	+	Номер версии программного продукта
SpecialBuild		Обновляемая или временная версия

Совет. Включайте строковые и языковые ресурсы версий в ваши приложения для отображения этой информации в диалоговом окне About. Тем самым вы сможете обновлять данные о версиях, редактируя соответствующий ресурс, а не модифицируя константы в исходном тексте программы. Единственный набор исходных файлов может быть использован тогда для получения нескольких версий приложения, возможно, с разными опциями компиляции или с другими командами условной компиляции.

Книги издательства "БИНОМ", планируемые к выходу в свет в декабре 1994 – феврале 1995 годов:

- Novell DOS 7 ... для пользователя!*
- Quattro Pro for Windows ... для пользователя!*
- Windows: тысячи полезных рецептов*
- MS Access ... за 5 минут*
- Lotus Organizer ... за 5 минут*
- MS DOS 6.22 ... за 5 минут*
- Norton Commander 4.0 ... за 5 минут*
- CorelDRAW! 5.0 ... за 5 минут*

Серия COMPUTER CLUB

- Программирование в Paradox for Windows в примерах (версии 4.5 и 5.0)
- Программирование игр для Windows на Borland C++
- Сжатие диска
- PC изнутри
- Просто и ясно о Borland C++ (версии 4.0 и 4.5)
- Программирование в FoxPro
- Программирование звука для DOS и Windows*
- Руководство по DOS Питера Нортона (версии 6.2 и 6.22)
- Работаем в Quattro Pro for Windows

** Книги издаются совместно с Торгово-издательским бюро BHV-Киев*

Производственно-техническое издание

Том Сван

ФОРМАТЫ ФАЙЛОВ WINDOWS

Компьютерная графика "Graphic Design Group"

Подписано в печать 16.11.94. Формат 70×108 1/16. Усл. печ. л. 18.
Бумага офсетная. Печать офсетная.
Тираж 26000 экз. Заказ 360

Издательство «БИНОМ», 1994 г.
Москва, ул. Новослободская, д. 50/1, стр. 1а.
Лицензия на издательскую деятельность № 063367 от 20 мая 1994 г.

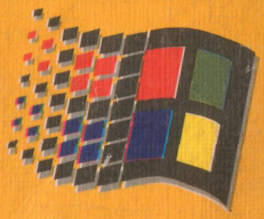
Отпечатано с готовых диапозитивов в полиграфической фирме
«Красный пролетарий»
103473, Москва, Краснопролетарская, 16

WINDOWS

ФОРМАТЫ ФАЙЛОВ

Сжатое и углубленное руководство по различным типам файлов, поставляемых Microsoft Windows. Внимание фокусируется на деталях внутренней структуры файлов **растровой графики, курсоров, пиктограмм, шрифтов, файлов Windows Write, картотеки, календаря, файлов групп, ресурсов, .PIF, .EXE** и других типов.

Эта книга даст Вам в руки все необходимые инструменты для работы с самыми разнообразными файлами графики, мультимедиа, приложений и другими файлами Windows.



ВЫ СМОЖЕТЕ:

- Создавать собственные шрифты
- Работать с различными графическими файлами
- Улучшить качество ваших программ для Windows

Для программистов, работающих в среде Windows 3.1