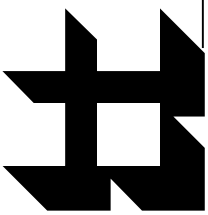

ГЛАВА 19

Windows API

Введение в Windows API

API (Application Programming Interface) — это интерфейс программирования приложений, термин, часто упоминаемый разработчиками программного обеспечения. Если разрабатываемое вами приложение имеет функцию, позволяющую обращаться к нему из других приложений, то это — API вашего приложения. Параметры, которые принимает ваша функция, образуют ее API, т. к. они являются средством, при помощи которого другие приложения взаимодействуют с данной функцией.

Операционная система Windows предоставляет большой набор функций, позволяющих различным приложениям, в том числе и приложениям Visual FoxPro, обмениваться информацией с Windows на низком уровне. Такие функции принято называть функциями Windows API. Использование Windows API в приложениях Visual FoxPro позволяет реализовать возможности, недостижимые стандартными средствами языка.

Объявление функций Windows API в Visual FoxPro

Функции Windows API скомпонованы в динамически связываемые библиотеки (Dynamic Link Library, DLL). Файлы таких библиотек имеют расширение dll и обычно располагаются в системной папке Windows. Перед тем как использовать функцию Windows API в вашем приложении, вы должны ее объявить при помощи команды DECLARE..DLL:

```
DECLARE [cFunctionType] FunctionName IN LibraryName [AS AliasName] ;
        [cParamType1 [@] ParamName1, cParamType2 [@] ParamName2, ...]
```

Параметры команды:

CFunctionType

необязательный параметр, указывающий, какой тип данных возвращает функция. Типы возвращаемых данных перечислены в табл. 19.1.

FunctionName

имя функции в DLL-библиотеке. Имя функции чувствительно к регистру символов, т. е. имена `GetDC` и `GETDC` определяют совершенно разные функции.

LibraryName

наименование файла DLL-библиотеки, в которой находится функция. Для библиотек Kernel32.dll, Gdi32.dll, User32.dll, Mpr.dll и Advapi32.dll можно использовать синоним WIN32API.

AliasName

необязательный параметр, который позволяет вместо имени функции использовать придуманный вами псевдоним. Написание псевдонима, в отличие от имени функции, не чувствительно к регистру символов. Как правило, псевдоним используется, когда имя API-функции совпадает с именем встроенной (или вашей) функции Visual FoxPro. Впрочем, если вы посмотрите, как используются Windows API-функций в примерах, поставляемых с Visual FoxPro (папка Samples), то, вероятно, обратите внимание на использование разработчиками префикса WinAPI_, добавляемого к имени функции при формирования ее псевдонима

cParamType

указывает тип данных передаваемого функции параметра. Если параметр передается по ссылке, то после *cParamType* ставится символ @. Допустимые типы данных перечислены в табл. 19.2.

cParamName

необязательный параметр; применяется для смыслового описания передаваемого значения.

Таблица 19.1. *Типы данных, возвращаемых Windows API-функцией*

Возвращаемое значение	Размер, байт	Описание
`Short`	2	Целое число
`Integer, Long`	4	Целое число
`Single`	4	Вещественное число
`Double`	8	Вещественное число
`String`	—	Символьная строка
`Object`	—	Указатель на объект (на интерфейс IDispath объекта)

Таблица 19.2. *Допустимые типы передаваемых в функцию значений параметров*

Передаваемое значение	Размер, байт	Описание
Integer, Long	4	Целое число

Таблица 19.2 *(окончание)*

Передаваемое значение	Размер, байт	Описание
Single	4	Вещественное число
Double	8	Вещественное число
String	—	Символьная строка
Object	4	Возвращаемая функцией ссылка на объект

Все функции Windows API, как, впрочем, и сама Windows, написаны на языках программирования C и C++. Поэтому для того, чтобы понять, как правильно использовать API-функции в Visual FoxPro (который, кстати, так же написан на C++, по крайней мере, его ядро), познакомимся, какие типы данных применяются в C++ и Windows, и, что не менее важно, разберемся с такими типами данных, как перечисления, структуры и указатели. Кроме того, вы узнаете, что такое прототипы функций в C++, и как, основываясь на описании прототипа функции в MSDN, правильно объявить ее в команде DECLARE..DLL.

Базовые типы данных в C++

Если вы знакомы с языком программирования C++, то знаете, как легко в нем можно создавать различные типы данных. Достаточно написать следующий код:

```
Typedef int INT32;
```

и вот у вас уже новый тип данных с именем INT32, полностью соответствующий типу int, но, с точки зрения C++, это совершенно разные типы данных, и попытка присвоить переменной типа INT32 значение переменной типа int приведет к ошибке!

Изобилие типов данных заставляет многих разработчиков думать, что программирование с использованием API является трудным. Но это не так! В языке C++ на самом деле существуют следующие базовые типы данных:

- char — символ в формате ANSI. Он требует для хранения один байт;
- wchar — символ в формате Unicode. Требует для хранения два байта;
- int — целые числа. Они делятся в C++ на три подтипа: int, short int и long int. Последние обычно сокращаются до short и long. Тип short — это 16-разрядное (занимает два байта), а типы int и long — 32-разрядные целые числа (требуют для хранения четыре байта);
- float — вещественные числа, имеющие дробную часть. Требуют для хранения четыре байта;

- `double` — вещественные числа двойной точности. Занимают восемь байт;
- `enum` — перечисляемый тип данных;
- `void` — используется для обозначения величин, не имеющих никакого значения и не требующих памяти для хранения (пустой тип данных);
- `pointer` — указатель; он не содержит информацию в общепринятом смысле, как другие типы C++; вместо этого, в каждом указателе находится адрес ячейки памяти, где хранятся реальные данные. Этот тип использует для хранения четыре байта, независимо от типа данных, на которые он указывает.

Как ни странно, но строковый тип в C++ отсутствует. Все символьные строки представлены в C++ как массивы символов.

Целочисленные типы данных могут объявляться как беззнаковые. Модификатор unsigned (без знака) используется со следующими типами данных: `char`, `short`, `int` и `long`.

Например, следующее объявление переменной в C++:

```
usigned int MyVar;
```

означает, что переменная `MyVar` — 32-разрядное целое число без знака.

Модификатор `const` указывает, что переменная указанного типа является константой, т. е. ее значение не может быть изменено.

Перечисляемый тип `enum` связывает с переменной набор именованных констант, называемых перечисляемыми константами. Объявление перечисляемого типа выглядит так:

```
enum поле_тега { const1,  const2, ... } переменная;
```

Если *поле_тега* опускается, то после закрывающей фигурной скобки необходимо указать переменную. Если *поле_тега* указано, то не указывается *переменная*.

Исторически сложилось так, что тип `enum` равнозначен типу `int` — т. е. переменная перечисляемого типа — целое число и занимает в памяти четыре байта. Каждая перечисляемая константа имеет значение, определяемое ее порядковым номером в списке; нумерация начинается с нуля. В качестве примера рассмотрим перечисление `CombineMode`:

```
enum CombineMode{
    CombineModeReplace,
    CombineModeIntersect,
    CombineModeUnion,
    CombineModeXor,
    CombineModeExclude,
    CombineModeComplement
    };
```

В этом перечислении константа `CombineModeReplace` имеет значение 0, константа `CombineModeIntersect` имеет значение 1 и т. д.; последняя константа, `CombineModeComplement`, имеет значение 5.

Значения перечисляемых констант могут быть указаны явно, как, например, в следующем примере:

```
enum DashCap{
    DashCapFlat = 0,
    DashCapRound = 2,
    DashCapTriangle = 3
    };
```

Перечисленные типы данных покрывают 99% всех типов данных, используемых в программировании Windows API. Это звучит слишком просто, не так ли? Почему же описания API-функций содержат все эти типы — `HWND`, `HINSTANCE`, `POINT` и им подобные?

Причиной тому является то, что C++ имеет особенность, называемую *strict-typing*. Однажды появившись, переменная одного типа может принимать только те значения, которые соответствуют ее типу. Вы не можете сначала сохранить в переменной строку, а затем присвоить ей число. В Visual FoxPro мы обычно стараемся следовать некоторым соглашениям об именовании переменных, таким как венгерская нотация. Например, именем `cName` обозначается переменная символьного типа, тогда как `nCount` — числовую. *Strict-typing* позволяет создать новый тип данных, присвоив существующему типу данных новое имя. Каждый новый тип данных в C++ отличается от других типов, несмотря на то, что внутренне они могут хранить одни и те же данные.

Windows еще более усложняет использование этой концепции. К примеру, тип данных `LONG` в действительности является аналогом типа `long int`, а тип `UINT` — аналогом типа `unsigned int`.

Типы данных Windows

Определение соответствия базового типа данных C++ типу данных, используемому API-функцией, является одной из тяжелейших задач в программировании API. Используйте следующее правило: если вы не можете найти слово float, double, char или str где-либо в имени функции или параметра, то это обычно 32-разрядное целое число. Потребуется некоторое время для понимания и выработки навыков, но потом вы будете запросто преобразовывать типы данных. В табл. 19.3 приведены некоторые типы данных Windows и соответствующие им типы данных Visual FoxPro, используемые при объявлении функции.

Таблица 19.3. *Типы данных Windows*

Тип данных Windows	Тип данных Visual FoxPro	Размер, байт	Описание
`BOOL`	`Long`	4	Используется для обозначения логического значения. В Visual FoxPro: 0 — ложь, не 0 — истина
`BOOLEAN`	`Long`	4	То же, что и BOOL

DOUBLE	Double	8	Вещественное число двойной точности
FLOAT	Single	4	Вещественное число
HANDLE	Long	4	Целое число без знака
HBITMAP	Long	4	Целое число без знака
HDC	Long	4	Целое число без знака
HICON	Long	4	Целое число без знака

Таблица 19.3 *(окончание)*

Тип данных Windows	Тип данных Visual FoxPro	Размер, байт	Описание
HGLOBAL	Long	4	Целое число без знака
HKL	Long	4	Целое число без знака
HINSTANCE	Long	4	Целое число без знака
HRESULT	Long	4	Целое число без знака
HWND	Long	4	Целое число без знака
LONG	Long	4	Целое число без знака
LONGLONG	String	8	Целое число размером 8 байт
LPARAM	Long	4	Целое число без знака
SHORT	Integer	2	Целое со знаком длиной 2 байта
SIZE_T	Long	4	Целое число без знака
UINT	Long	4	Целое число без знака
WNDPROC	Long	4	Целое число без знака
WORD	Integer	2	Целое без знака длиной 2 байта
WPARAM	Long	4	Целое число без знака

Указатели

Специальный тип данных, широко используемый в C++, — это указатели (pointers). Указатель представляет собой целочисленную 32-разрядную переменную, содержащую адрес области памяти, по которому хранятся данные. В наименовании типа указателя обычно указывается тип данных, на которые он указывает; но любой указатель всегда требует для хранения себя четыре байта. При создании функций Windows API применяется соглашение, по которому имя указателя начинается с символов LP (Long Pointer, или "длинный указатель"; этот префикс унаследован со времен Windows 3.x, в которой использовалась 16-разрядная адресация и существовали просто указатели и "длинные" (FAR) указатели). Например, тип LPWORD определяет указатель, содержа-

щий адрес переменной типа Word, а указатель типа LPDWORD ссылается на переменную типа DWORD.

Указатели на числовые данные при объявлении Windows API-функции передаются по ссылке. Как пример, рассмотрим функцию GetFileSize. Вот ее прототип (подробнее о прототипах функций будет рассказано ниже):

```
DWORD GetFileSize(
    HANDLE hFile,              // дескриптор файла
    LPDWORD lpFileSizeHigh     // указатель
    );
```

Второй параметр, передаваемый функции (lpFileSizeHigh), — указатель на переменную типа DWORD. Вот как будет выглядеть объявление этой функии в Visual FoxPro:

```
DECLARE Long GetFileSize IN WIN32API Long hFile, Long @ FileSizeHight
```

Как видите, параметр FileSizeHight передается функции по ссылке, потому что передача по ссылке — это и есть передача указателя.

Сложнее обстоит дело с символьными данными. Как уже говорилось, строки символов в C++ представляют собой массивы, которые при программировании API-функций определяются как тип STR. В табл. 19.4 показаны используемые в Windows типы указателей на символьные строки.

Таблица 19.4. *Указатели на символьные строки*

Тип указателя	Описание
LPSTR	Указатель на модифицируемую строку символов формата ANSI
LPWSTR	Указатель на модифицируемую строку символов формата Unicode
LPTSTR	Модифицируемая строка, адрес которой хранится в указателе, может быть как формата ANSI, так и формата Unicode
LPCSTR	Указатель на не модифицируемую строку символов формата ANSI
LPCWSTR	Указатель на не модифицируемую строку символов формата Unicode
LPCTSTR	Не модифицируемая строка, адрес которой хранится в указателе, может быть как формата ANSI, так и формата Unicode

Параметры, являющиеся указателями на символьные строки, при объявлении Windows API-функции в Visual FoxPro имеют тип String. Если строка может быть модифицирована API-функцией (т. е. функция изменяет данные в этой строке), то она должна передаваться по ссылке с использованием при объявлении параметра символа @:

```
*!* В следующем объявлении Windows API-функции
*!* передаваемый ей параметр не изменяется (подобно передаче по ссылке)
DECLARE Long FuncName IN WIN32API String TextString
*!* А при таком объявлении параметр может быть изменен в API-функции
DECLARE Long FuncName IN WIN32API String @ TextString
```

Структуры

Структуру можно рассматривать как набор переменных различных типов, образующих единое целое. В C++ структура создается при помощи ключевого слова `struct`, за которым следует необязательное поле тега и список элементов (полей) структуры:

```
struct поле_тега {
    тип_элемента элемент1;
    тип_элемента элемент2;
    .....
    тип_элемента элементN;
    };
```

Возможно и такое объявление структуры:

```
struct {
    тип_элемента элемент1;
    тип_элемента элемент2;
    .....
    тип_элемента элементN;
    } переменная;
```

В этом объявлении отсутствует поле тега; при этом создается так называемый анонимный структурный тип. Такой синтаксис позволяет связать с этим структурным типом одну или несколько переменных, как, например, в следующем примере:

```
struct {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
    } SYSTEMTIME, *PSYSTEMTIME;
```

Структуры очень похожи на записи таблиц Visual FoxPro. Так, если запись таблицы `personal` содержит поля `fio`, `address`, `tlfnumber` и `email`, то для обращения к полю `tlfnumber` используется следующий синтаксис:

```
personal.tlfnumber
```

Так же выглядит и обращение к полю структуры:

```
SYSTEMTIME.wMinute
```

Для формирования структур в Visual FoxPro используются переменные, содержащие строки символов. Например, для рассмотренной выше структуры `SYSTEMTIME` вам понадобится строка, содержащая 16 символов (16 байт). В первые два байта этой переменной заносится значение поля *`wYear`*, в следующие два байта — значение поля *`wMonth`*, в следующие два байта — значение поля *`wDayOfWeek`* и т. д., пока структура не будет полностью сформирована. А при объявлении API-функции в Visual FoxPro тип параметра, в котором передается переменная, содержащая структуру, должен иметь

тип String. Для записи в строковую переменную числовых данных используется встроенная функция BINTOC().

Как вы уже знаете, при программировании Windows API на C++ описание указателя на структуру начинается с символов LP (Long Pointer), за которыми следует наименование структуры. Так, указатель на структуру SYSTEMTIME будет иметь тип LPSYSTEMTIME, указатель на структуру POINT будет иметь тип LPPOINT и т. д. Вследствие того, что количество структур в Windows никак не ограничивается, существует чрезвычайно большое количество типов указателей на структуры.

Если данные в передаваемой структуре не должны изменяться, то указатель на такую структуру объявляется так:

```
CONST имя_структуры * имя_указателя
```

Здесь модификатор CONST означает, что данные в структуре не должны меняться, а символ (*) после имени структуры означает, что вся эта строка есть описание указателя на структуру. В следующем примере показан прототип функции CopyRect, которая копирует одну структуру в другую:

```
BOOL CopyRect(
    LPRECT lprcDst,       // Значения полей структуры могут быть изменены
    CONST RECT * lprcSrc  // Значения полей структуры не изменяются
    );
```

Прототипы функций

Теперь, когда мы разобрались с типами данных, пришло время подробно познакомиться с таким компонентом C++, как прототипы функций.

Согласно стандарта ANSI, все функции в C++ должны иметь прототипы, или описания синтаксиса. Прототип функции достаточно прост:

```
возвращаемый_тип имя_функции(тип_параметра(ов) имя_параметра(ов));
```

Если в прототипе указано, что функция возвращает значение типа VOID, то это значит, что функция не возвращает никаких значений. Если тип VOID указан как *тип_параметра*, то это означает, что функция не имеет параметров.

Вот как, например, выглядит прототип функции CopyFile:

```
BOOL CopyFile(
    LPCTSTR lpExistingFileName,
    LPCTSTR lpNewFileName,
    BOOL bFailIfExists
);
```

Функция копирует содержимое существующего файла в новый файл. Первые два передаваемых функции параметра, *lpExistingFileName* и *lpNewFileName*, имеют тип LPCTSTR. Третий параметр, *bFailInExists*, имеет тип BOOL.

Посмотрев в табл. 19.4, видим, что тип LPCTSTR — это указатель на символьные строки, которые могут хранить данные как в формате ANSI, так и в формате Unicode.

В Visual FoxPro символьные данные имеют формат ANSI; но можно ли быть уверенным, что функция `CopyFile` будет работать именно с этим форматом данных?

На самом деле проблема решается просто. Практически для всех функций Windows API, работающих с символьными данными, существует два варианта реализации, причем различаются они по завершающему символу "A" или "W", добавляемому к имени функции. Так, функция `CopyFileA` предназначена для работы с символами формата ANSI, а функция `CopyFileW` — для работы с символами формата Unicode.

Если открыть файл Winbase.h, в котором объявляется эта функция, то там можно найти следующие строки:

```
BOOL WINAPI CopyFileA(
    __in LPCSTR lpExistingFileName,
    __in LPCSTR lpNewFileName,
    __in BOOL bFailIfExists
    );
BOOL WINAPI CopyFileW(
    __in LPCWSTR lpExistingFileName,
    __in LPCWSTR lpNewFileName,
    __in BOOL bFailIfExists
    );
#ifdef UNICODE
    #define CopyFile  CopyFileW
#else
    #define CopyFile  CopyFileA
#endif // !UNICODE
```

Как видите, действительно существуют две реализации одной функции; для функции `CopyFileA` используются указатели типа `LPCSTR`, а для функции `CopyFileW` — типа `LPCWSTR`.

Какая из этих функций будет "отзываться" на имя `CopyFile`, определяется в операторе препроцессора `#ifdef`. Если переменная `UNICODE` определена, то это будет функция `CopyFileW`, иначе — `CopyFileA`. Visual FoxPro не определяет эту переменную, поэтому с высокой степенью вероятности можно предположить, что будет вызвана функция, поддерживающая формат ANSI, т. е. `CopyFileA`. Тем не менее при объявлении функции в Visual FoxPro лучше явно указать реализацию ANSI, а в качестве псевдонима использовать стандартное имя функции:

```
DECLARE Long CopyFileA IN Win32API AS CopyFile ;
    String lpExistingFileName, String lpNewFileName, Long bFailIfExists
```

Псевдоним `CopyFile`, указанный в объявлении функции `CopyFileA`, и будет тем именем функции, которое вы будете применять при ее вызове. Кстати, псевдоним, в отличие от имени Windows API-функции, не зависит от регистра символов.

MSDN как источник информации о прототипах функций

MSDN — это наиболее полный и достоверный источник, в котором вы можете найти всю необходимую информацию о прототипах функций и их использовании. К сожалению, в MSDN не приводятся значения именованных констант, используемых функциями. Описания этих констант, а также массу иной полезной информации вы можете найти в заголовочных файлах (имеющих расширение h). Эти файлы поставляются в комплекте Visual Studio; так, в составе Visual Studio 2005 они располагаются в папке ..\Microsoft Visual Studio 8\VC\Platform SDK\Include\.

Другой, более дружественный источник необходимой информации — это большое количество литературы для системного программирования в Windows, где вы можете найти описания множества Windows API-функций на русском языке.

И, наконец, еще один доступный источник — это Интернет и, в частности, сайт **http://msdn.microsoft.com**.

К сожалению, в каждой очередной редакции Visual Studio разработчики меняют как внешний вид главного окна приложения MSDN Library, так и структуру оглавления. Вот как, например, выглядит окно MSDN Library в Visual Studio 2005 (рис. 19.1).

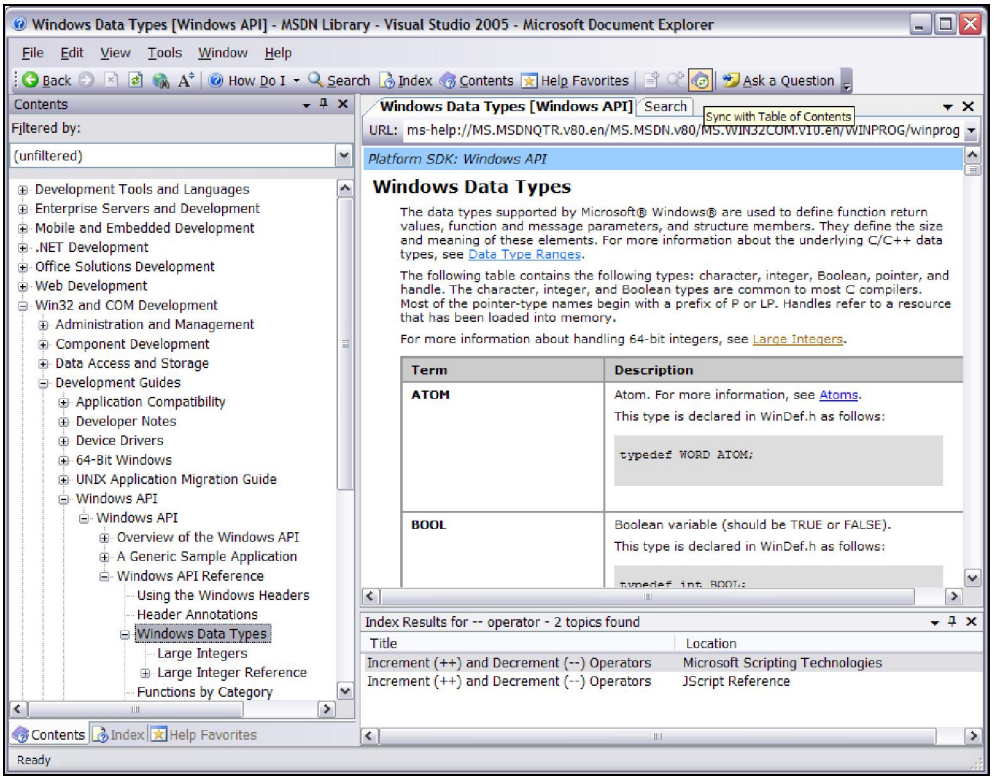

Рис. 19.1. Окно MSDN Library для Visual Studio 2005

Полный список типов данных Windows вы можете найти, открыв следующий узел в оглавлении (Contents):

```
Win32 and COM Development
  Development Guides
    Windows API
      Windows API
        Windows API Reference
          Windows Data Types
```

Описания функций сгруппированы по разделам. Если вам уже известно имя функции, то перейдите на вкладку **Index** (на рис. 19.1 — слева внизу) и в появившемся поле **Look for** введите имя искомой функции. Если в MSDN будет найдено несколько статей, в которых встречается ссылка на функцию, то их список появится в области **Index Result**. Вы должны будете выбрать в предложенном списке одну из статей.

Описание функции содержит ее прототип, список передаваемых параметров, тип возвращаемого значения и информацию, определяющую область применения и расположение, включая наименование файла DLL, в котором находится функция. Щелкните по расположенной на панели инструментов кнопке **Sync with Table of Contents** (на

рис. 19.1 она выделена) — в области **Contents** окна появится оглавление, в котором будет открыт узел, содержащий наименование статьи, в которой описывается функция.

Вот как, например, описана в MSDN функция `CopyRect`:

> ## CopyRect
>
> The CopyRect function copies the coordinates of one rectangle to another.
>
> ```
> BOOL CopyRect(
> LPRECT lprcDst,     // destination rectangle
> CONST RECT* lprcSrc // source rectangle
> );
> ```
>
> **Parameters**
>
> `lprcDst`
> [out] Pointer to the RECT structure that receives the logical coordinates of the source rectangle.
>
> `lprcSrc`
> [in] Pointer to the RECT structure whose coordinates are to be copied in logical units.
>
> **Return Values**
>
> If the function succeeds, the return value is nonzero.
> If the function fails, the return value is zero.
> **Windows NT/2000/XP:** To get extended error information, call GetLastError.
>
> **Remarks**
>
> Because applications can use rectangles for different purposes, the rectangle functions do not use an explicit unit of measure. Instead, all rectangle coordinates and dimensions are given in signed, logical values. The mapping mode and the function in which the rectangle is used determine the units of measure.
>
> **Example Code**
>
> For an example, see Using Rectangles.
>
> **Requirements**
>
> Windows NT/2000/XP: Included in Windows NT 3.1 and later.
> Windows 95/98/Me: Included in Windows 95 and later.
> Header: Declared in Winuser.h; include Windows.h.
> Library: Use User32.lib.

Как видите, информация вполне достаточная. Функция возвращает значение типа `BOOL`, ей передаются два параметра: типа `LPRECT` и `CONST RECT*` — указатели на структуры типа `RECT`.

Очень полезная информация находится в разделе **Requirements** описания функции, в частности, в строке **Header** указано, в каком заголовочном файле находится прототип функции — в данном случае это Winuser.h. А в строке **Library Use** приведено имя библиотеки, в которой находится функция. Пусть вас не смущает, что там указан

тип lib — фактически эта запись означает, что функция находится в библиотеке User32.dll.

Но это еще не все! Так как функция использует структуры, то в ее описании присутствует гиперссылка на структуру RECT. Щелкните мышью по этой гиперссылке, и на экране появится информация о структуре.

Формирование структур в Visual FoxPro

В девятой версии Visual FoxPro существенно расширены возможности встроенных функций BINTOC() и CTOBIN(). Теперь эти функции можно применять для преобразования числовых данных в формат, пригодный для использования в структурах. Напомним, что функция BINTOC() выполняет преобразование числа в строку, а функция CTOBIN() — строки в число.

Синтаксис функции BINTOC:

```
BINTOC(nExpression, eFlag)
```

где:

nExpression — преобразуемое числовое значение;

eFlag — тип преобразования.

Некоторые из допустимых значений, которые может принимать параметр eFlag, приведены в табл. 19.5.

Таблица 19.5. *Значения параметра eFlag функции BINTOC()*

eFlag	Описание
"2RS"	Преобразует 16-разрядное целое число в двухбайтовую строку
"4RS"	Преобразует 32-разрядное целое число в четырехбайтовую строку
"F"	Преобразует 32-разрядное вещественное число в четырехбайтовую строку
"B"	Преобразует 64-разрядное вещественное число в восьмибайтовую строку

Синтаксис функции CTOBIN:

```
CTOBIN(cExpression, eFlag)
```

где:

cExpression — преобразуемая строка символов;

eFlag — тип преобразования.

Возможные значения параметра eFlag приведены в табл. 19.6.

Ниже в листинге 19.1 показаны примеры использования этих функций.

Таблица 19.6. *Значения параметра eFlag функции CTOBIN ()*

eFlag	Описание
"2RS"	Преобразует двухбайтовую строку в 16-разрядное целое число
"4RS"	Преобразует четырехбайтовую строку в 32-разрядное целое число
"4N"	Преобразует четырехбайтовую строку в 32-разрядное вещественное число
"8N"	Преобразует восьмибайтовую строку в 64-разрядное вещественное число

```
? CTOBIN(BINTOC(1000.55,'4RS'), '4RS')   && Результат: 1000
? CTOBIN(BINTOC(-1000.55,'4RS'), '4RS')  && Результат: -1000
? CTOBIN(BINTOC(1000.55,'F'), '4N')      && Результат: 1000.549987929
? CTOBIN(BINTOC(-1000.55,'F'), '4N')     && Результат: -1000.549987929
? CTOBIN(BINTOC(1000.55,'B'), '8N')      && Результат: 1000.55
? CTOBIN(BINTOC(-1000.55,'B'), '8N')     && Результат: -1000.55
```

В качестве примера сформируем в Visual FoxPro структуру RECT. Эта структура используется в рассмотренной ранее функции CopyRect для описания координат прямоугольной области. Вот как эта структура описывается в MSDN:

```
typedef struct _RECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
    } RECT, *PRECT;
```

Как видите, структура RECT содержит четыре поля, в каждом из которых хранится значение типа LONG. Для ее формирования в Visual FoxPro понадобится строка длиной 16 байт.

В листинге 19.2 показан код, в котором объявляется функция CopyRect, формируются структуры Dst и Src для передачи их как параметров функции, а затем выполняется копирование одной структуры в другую. В примере используется встроенная функция BINTOC для преобразования числа в строку.

```
DECLARE Long CopyRect IN WIN32API String @ Dst, String Src
*
* Формируем структуру Src
* nLeft, nTop, nRight, nBottom — значения, помещаемые в структуру
*
cSrc = BINTOC(nLeft,'4RS') + BINTOC(nTop,'4RS') + ;
       BINTOC(nRight,'4RS') + BINTOC(nBottom,'4RS')
*
* Подготавливаем место для структуры Dst
```

```
*
cDst = REPLICATE(CHR(0),16)
nResult = CopyRect(@cDst, cSrc)   && Копирование
```

В следующем примере показано, как, используя функцию `CTOBIN`, можно "разобрать" структуру, получив числовые значения ее полей (листинг 19.3).

```
nLeft = CTOBIN(SUBSTR(cDst,1,4), '4RS')     && RECT.left
nTtop = CTOBIN(SUBSTR(cDst,5,4), '4RS')     && RECT.top
nRight = CTOBIN(SUBSTR(cDst,9,4), '4RS')    && RECT.right
nBottom = CTOBIN(SUBSTR(cDst,13,4), '4RS') && RECT.bottom
```

Передача массивов в функцию Windows API

Массив с точки зрения C++ — это переменная, содержащая несколько элементов одного типа. Доступ к каждому отдельному элементу массива осуществляется при помощи индекса. Все элементы массива имеют одинаковый размер, нельзя описывать массивы со смешанными типами данных. Все элементы массива хранятся в памяти последовательно, один за другим, при этом минимальное значение индекса соответствует первому элементу, а максимальное — последнему.

Массив в Visual FoxPro позволяет хранить в своих элементах разные типы данных. Каждая переменная Visual FoxPro на самом деле описана как структура, поэтому невозможно передать массив из Visual FoxPro в Windows API-функцию, просто указав его имя как параметр. Более того, в команде `DECLARE..DLL` нет такого типа данных, как массивы.

Тем не менее выход из этого положения есть. Как и для структур, для передачи массивов используются символьные переменные. Числовые данные из массива должны быть преобразованы в строку, которую вы и передаете как параметр в Windows API-функцию. В листинге 19.4 показан код пользовательской функции `ArrayToString`, формирующей строку из массива. Функция получает по ссылке массив `taArray` и флаг *`tlTypeOfValue`*, указывающий, как будут преобразованы значения элементов массива — как целые (`tlTypeOfValue=.F.`) или вещественные (`tlTypeOfValue=.T.`) числа:

```
FUNCTION ArrayToString
PARAMETERS taArray, tlTypeOfValue
    EXTERNAL ARRAY taArray
    LOCAL lnLenArray, lnIndex, lcStruct, lcType
    lcType = IIF(tlTypeOfValue = .t., 'F', '4RS')
    lnLenArray = ALEN(taArray)     && Количество элементов в массиве
    lcStruct = ""
    FOR lnIndex = 1 TO lnLenArray
        lcStruct = lcStruct + BINTOC(taArray[lnIndex], lcType)
```

```
    ENDFOR
    RETURN lcStruct
ENDFUNC
```

Функция работает как с одномерными, так и с двумерными массивами.

Форматы символьных данных ANSI и Unicode

В кодировке ANSI (применяемой в Visual FoxPro) каждый символ определяется одним байтом, т. е. максимальное количество символов равно 256, что, конечно, очень мало. Например, при русификации часть стандартных символов ANSI заменяется на символы кириллицы. А в некоторых алфавитах, например, японской кане, столько символов, что одного байта для их кодировки просто недостаточно. Для поддержки таких языков и, что более важно, для облегчения "перевода" программ на другие языки была разработана кодировка Unicode. Каждый символ в Unicode состоит из двух байтов, что позволило расширить набор допустимых символов до 65 536.

Достаточно большое количество функций Windows API используют при работе с символьными строками только формат Unicode. Для работы с такими функциями необходимо выполнять конвертирование символьных данных из одного формата в другой.

Встроенная функция Visual FoxPro `STRCONV` выполняет конвертирование строк как из формата ANSI в Unicode, так и обратно. Вот ее синтаксис:

```
STRCONV(cExpression, nConvSetting [, nRegID [,nRegIDType]])
```

Параметр *cExpression* — это строка, которую необходимо конвертировать. Параметр *nConvSetting* указывает характер конвертирования. Из всех его возможных значений нас интересуют только два:

- 5 — конвертирование из ANSI в Unicode;
- 6 — конвертирование из Unicode в ANSI.

Необязательные параметры *nRegID* и *nRegIDType* определяют дополнительные региональные настройки и могут быть опущены.

Ниже показаны примеры использования функции `STRCONV`:

```
cUnicodeString = STRCONV(cANSIString, 5) && Конвертирование в Unicode
cANSIString = STRCONV(cUnicodeString, 6) && Конвертирование в ANSI
```

Многократное объявление функций Windows API

Visual FoxPro регистрирует объявленные функции в специальном буфере, причем каждая из объявленных функций регистрируется в единичном экземпляре, таким образом, запоминается только первое объявление функции.

Встроенная функция `ADLLS` возвращает количество зарегистрированных API-функций. Вот ее синтаксис:

```
nCount = ADDLS(ArrayName)
```

Функция получает массив, в который заносит следующую информацию:

- Имя API-функции
- Псевдоним функции
- Имя DLL-библиотеки, в которой находится функция

Если передаваемый функции массив не существует, то он создается. Если не было объявлено ни одной API-функции, массив не создается (существующий массив не изменяется).

Удаление зарегистрированных функций

Команда `CLEAR DLLS` удаляет все зарегистрированные API-функции. Допускается выборочное удаление; список удаляемых функций должен быть передан команде как набор параметров:

```
CLEAR DLLS FuncName1 [, FuncName2, ...]
```

Имена функций можно указывать как непосредственно, так и в виде текстовых литералов. Например, команды

```
CLEAR DLLS GlobalAlloc
```

и

```
CLEAR DLLS "GlobalAlloc"
```

удаляют функцию `GlobalAlloc`.

Распределение памяти для структур с указателями

Достаточно часто встречается ситуация, когда передаваемая в функцию Windows API структура содержит указатели. В качестве примера рассмотрим функцию `StartDoc`, создающую документ для печати на принтере. Вот ее прототип:

```
int StartDoc(HDC hdc, CONST DOCINFO* lpdi);
```

Как видите, второй передаваемый функции параметр — это указатель на структуру. Имя этой структуры — `DOCINFO`. Вот как она описана в MSDN:

```
typedef struct {
    int     cbSize;
    LPCTSTR lpszDocName;
    LPCTSTR lpszOutput;
    LPCTSTR lpszDatatype;
    DWORD   fwType;
    } DOCINFO, *LPDOCINFO;
```

Первое поле структуры, *cbSize*, содержит значение длины структуры в байтах. А вот следующие три поля — это указатели на переменные, содержащие символьные данные. В частности, поле *lpszDocName* содержит указатель на строку с наименованием печатаемого документа (это то самое имя документа, которое вы видите, просматривая очередь печатаемых документов).

В Visual FoxPro достаточно сложно сформировать структуру, содержащую указатели. Во-первых, нужно выделить блок памяти и получить указатель на него. Во-вторых, необходимо переписать в эту память значение переменной Visual FoxPro — таким образом, у нас будет полностью реализован механизм указателей. Последнее, что остается сделать, — это поместить значение указателя в структуру. При этом нужно выполнить одно существенное требование: выделенная память должна быть фиксированной (не перемещаемой) — иначе в какой-то момент может оказаться, что наш указатель будет показывать на область, к которой наши данные не имеют никакого отношения!

Есть несколько возможностей получить блок памяти. Можно взять "кусочек" как из общедоступной памяти Windows, так и из памяти, выделенной процессу.

Распределение памяти Windows

Распределение памяти Windows выполняют функции `GlobalAlloc` и `LocalAlloc`. Эти функции ранее использовались для работы с 16-разрядной сегментированной моделью памяти, которая применялась в Windows 3.x. Переход к 32-разрядной виртуальной модели памяти устранил различие между этими функциями, т. к. с точки зрения 32-разрядных Windows исчезли различия между локальной и глобальной областями памяти, и обе эти функции возвращают указатели на 32-разрядные виртуальные адреса. Поэтому обе эти функции взаимозаменяемы.

К блокам памяти, распределенным функцией `GlobalAlloc`, процесс получает доступ для записи и чтения; эти блоки становятся недоступными для других процессов. Вы можете разрешить или запретить перемещение распределенной памяти (фиксированная область памяти не может быть перемещена, т. е. начальный адрес такой области не может быть изменен). Размер распределяемой памяти ограничен только доступной физической памятью, включая память в файле свопинга на диске, но не может превышать 4 Гбайт. Когда вы распределяете фиксированную память, `GlobalAlloc` возвращает указатель, который может немедленно использоваться приложением для обращения к памяти. Когда вы распределяете перемещаемую память, то после выполнения функции `GlobalAlloc` необходимо вызвать функцию `GlobalLock`, которая "заблокирует" перемещение на некоторое время и возвратит "правильный" указатель.

Вот прототип функции `GlobalAlloc`:

```
HGLOBAL GlobalAlloc(
    UINT uFlags,     // атрибуты распределения памяти
    SIZE_T dwBytes   // размер в байтах
    );
```

Параметр *uFlags* определяется как сумма значений битовых масок, которые приведены в табл. 19.7.

Таблица 19.7. *Константы, управляющие распределением памяти Windows*

Значения uFlag	**Описание**
GHND	Конкатенация значений GMEM_MOVEABLE и GMEM_ZEROINIT
GMEM_FIXED	Распределяется фиксированная (неперемещаемая) область памяти
GMEM_MOVEABLE	Распределяется перемещаемая память
GMEM_ZEROINIT	Выделенный блок памяти заполняется нулями
GPTR	Конкатенация значений GMEM_FIXED и GMEM_ZEROINIT

Для распределения фиксированной и заполненной нулями памяти параметр *uFlags* должен иметь значение, определяемое константой GPTR. Но как узнать, чему равно это значение? Найдите в MSDN описание функции GlobalAlloc, и в разделе **Requirements** посмотрите, в каком заголовочном файле находится ее прототип. Это файл Winbase.h. Именно в нем и следует искать описание значений констант. Ниже показан фрагмент этого файла.

```
/* Global Memory Flags */
#define GMEM_FIXED          0x0000
#define GMEM_MOVEABLE       0x0002
#define GMEM_NOCOMPACT      0x0010
#define GMEM_NODISCARD      0x0020
#define GMEM_ZEROINIT       0x0040
#define GMEM_MODIFY         0x0080
#define GMEM_DISCARDABLE    0x0100
#define GMEM_NOT_BANKED     0x1000
#define GMEM_SHARE          0x2000
#define GMEM_DDESHARE       0x2000
#define GMEM_NOTIFY         0x4000
#define GHND (GMEM_MOVEABLE | GMEM_ZEROINIT)
#define GPTR (GMEM_FIXED | GMEM_ZEROINIT)
```

Как видите, GPTR = GMEM_FIXED + GMEM_ZEROINIT = 0x0000 + 0x0040 = 0x0040.

Вот как выглядит объявление функции GlobalAlloc в Visual FoxPro:

```
DECLARE Long GlobalAlloc IN Kernel32.dll Long uFlags, Long MemSize
```

Функция GlobalSize возвращает размер распределенной памяти (в байтах). Вот ее прототип:

```
SIZE_T GlobalSize(HGLOBAL hMem);
```

Функции передается указатель на распределенную область памяти.

Объявление в Visual FoxPro:

```
DECLARE Long GlobalSize IN WIN32API Long hMem
```

Функция `GlobalFree` освобождает (возвращает Windows) память, распределенную функцией `GlobalAlloc`. Вот ее прототип:

```
HGLOBAL GlobalFree(HGLOBAL hMem);
```

Функция получает только один параметр — указатель на блок памяти. Вот ее объявление в Visual FoxPro:

```
DECLARE Long GlobalFree IN WIN32API Long hGlobal
= GlobalFree(hGlobal)
```

где `hGlobal` — указатель на блок памяти, возвращаемый функцией `GlobalAlloc`.

Пример использования этих функций показан в листинге 19.5.

```
#DEFINE GPTR 0x0040
DECLARE Long GlobalAlloc IN Kernel32.dll Long, Long
DECLARE Long GlobalSize IN Kernel32.dll Long
DECLARE Long GlobalFree IN Kernel32.dll Long
hGlobal = GlobalAlloc(GPTR, 1024)  && Распределяем 1 Кб памяти Windows
lnMemSize = GlobalSize(hGlobal)    && В MemSize — размер памяти в байтах
*!*
*!* команды процедуры
*!*
= GlobalFree(hGlobal)                && Освобождаем память
```

Функция `GlobalLock` блокирует перемещаемую область памяти, распределенную функцией `GlobalAlloc`. После вызова `GlobalLock` перемещение области запрещается до момента отмены блокировки функцией `GlobalUnLock`. В случае использования перемещаемой области памяти перед выполнением операций чтения/записи вы должны блокировать ее для получения "правильного" указателя.

Прототипы функций `GlobalLock` и `GlobalUnLock`:

```
LPVOID GlobalLock (HGLOBAL hMem);
LONG GlobalUnLock (HGLOBAL hMem);
```

где *hMem* — указатель на распределенную память.

Пусть вас не смущает возвращаемый функцией `GlobalLock` тип `LPVOID`. Он всего-навсего означает "указатель на что-то", т. е. функция просто не знает, указатель на какой тип данных будет возвращен.

В листинге 19.6 показан пример распределения перемещаемой памяти.

```
#DEFINE GHND 0x0042
DECLARE Long GlobalAlloc IN Kernel32.dll Long, Long
DECLARE Long GlobalLock IN Kernel32.dll Long
```

```
DECLARE Long GlobalUnLock IN Kernel32.dll Long
DECLARE Long GlobalFree IN Kernel32.dll Long
hGlobal = GlobalAlloc(GHND, 1024) && Распределяем 1 Кб памяти
hGlobal = GlobalLock(hGlobal)     && Получаем "правильный" дескриптор
*!*
*!* Код процедуры, выполняющий операции чтения / записи в память
*!*
= GlobalUnLock(hGlobal)           && Разрешаем перемещения памяти
*!*
*!* Код процедуры
*!*
= GlobalFree(hGlobal)             && Освобождаем память
```

Распределение памяти внутри процесса

В документации по Windows обычно рекомендуется использовать более быстродействующие функции, распределяющие память внутри процесса: `GetProcessHeap`, `HeapAlloc` и `HeapFree` (эту часть памяти обычно называют *кучей* — heap).

Функция `GetProcessHeap` возвращает дескриптор доступной памяти процесса, которая затем распределяется функцией `HeapAlloc` и освобождается функцией `HeapFree`.

Прототип функции `GetProcessHeap`:

```
HANDLE GetProcessHeap(void);
```

Функция `HeapAlloc` возвращает указатель на блок памяти заданного размера. Распределяемая память является неперемещаемой относительно начального адреса процесса. Вот прототип этой функции:

```
LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes);
```

Параметры:

hHeap
Дескриптор доступной памяти процесса, возвращаемый функцией `GetProcessHeap`.

dwFlags
Если равен восьми, то распределяемая память заполняется нулями; по умолчанию — нуль.

dwBytes
Размер распределяемой памяти (в байтах).

Если память не удается распределить, функция возвращает ноль.

Функция `HeapFree` освобождает память, распределенную функцией `HeapAlloc`. Вот ее прототип:

```
BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem);
```

Параметры:

hHeap

Дескриптор доступной памяти процесса, возвращаемый функцией `GetProcessHeap`.

`dwFlags`

В приложениях на Visual FoxPro всегда равен нулю.

`lpMem`

Указатель на блок памяти в куче процесса.

Объявление этих функций в Visual FoxPro выглядит так:

```
DECLARE Long GetProcessHeap IN Kernel32.DLL
DECLARE Long HeapAlloc IN Kernel32.DLL Long hHeap, Long flags, Long size
DECLARE Long HeapFree IN Kernel32.DLL Long hHeap, Long flags, Long @ pMem
```

В листинге 19.7 показано, как распределить память внутри процесса.

```
DECLARE Long GetProcessHeap IN Kernel32.DLL
DECLARE Long HeapAlloc IN Kernel32.DLL Long, Long, Long
DECLARE Long HeapFree IN Kernel32.DLL Long, Long
* Распределяем память
pMem = HeapAlloc(GetProcessHeap(), 8, nLenDocumentName)
*
* Выполняем необходимые действия и возвращаем распределенную память процессу
*
HeapFree(GetProcessHeap(), pMem)
```

Обратите внимание на необходимость получения базового адреса кучи в процессе именно в момент вызова функций `HeapAlloc` и `HeapFree`, потому что область памяти, отведенная Windows-процессу, постоянно перемещается.

Работа с указателями в Visual FoxPro

Вернемся к формированию структуры `DOCINFO`. Следующая стоящая перед нами задача — переписать содержимое переменной *`cDocumentName`* в распределенный блок памяти. Воспользуемся для этого встроенной функцией Visual FoxPro `SYS(2600)`. Вот ее синтаксис:

```
SYS(2600, dwAddress, nLenght [, cNewString])
```

Функция ведет себя по-разному в зависимости от того, указан параметр *`cNewString`* или нет.

Если параметр *`cNewString`* указан, то функция копирует *`nLenght`* байт из переменной *`cNewString`* в память по адресу, указанному в *`dwAddress`*.

Если параметр *`cNewString`* не указан, то функция возвращает *`nLenght`* байт из памяти по адресу, указанному в *`dwAddress`*.

Как видите, параметр *`dwAddress`* — это не что иное, как указатель.

В листинге 19.8 приведен полный код формирования структуры `DOCINFO` при распределении доступной памяти Windows.

```
#DEFINE GPTR     0x0040
DECLARE Long GlobalAlloc IN WIN32API Long uFlags, Long dwBytes
cDocumentName = 'Имя печатаемого документа'
nLenDocumentName = LEN(cDocumentName)
hGlobal = GlobalAlloc(GPTR, nLenDocumentName)
SYS(2600, hGlobal, nLenDocumentName, cDocumentName)
*
* Начинаем формировать структуру DOCINFO
* cDocInfo — переменная, в которой формируется структура
*
cDocInfo = BINTOC(20,'4RS')                  && DOCINFO.cbSize
cDocInfo = cDocInfo + BINTOC(hGlobal,'4RS')  && DOCINFO.lpszDocName
cDocInfo = cDocInfo + REPLICATE(CHR(0),12)   && Остальные поля структуры
*
* Конец формирования структуры DOCINFO
* После окончания печати документа следует освободить
* распределенную память:
*
* GlobalFree(hGlobal)
*
```

Структура формируется в переменной *cDocInfo*. В первые четыре байта записываем число 20 — это размер структуры (поле *cbSize*). Следующие четыре байта, добавляемые в переменную, — это указатель на область памяти, в которую записано содержимое переменной *cDocumentName* — наименование документа.

Затем к переменной *cDocInfo* добавляются еще двенадцать нулевых байтов — это поля структуры *lpszOutput*, *lpszDatatype* и *fwType*, которые, согласно документации, могут игнорироваться; это означает, что поля должны иметь нулевые значения. Таким образом, получилась строка длиной 20 байтов — что и требовалось.

Для сравнения в листинге 19.9 приведен код, в котором память распределяется внутри процесса.

```
DECLARE Long GetProcessHeap IN Kernel32.DLL
DECLARE Long HeapAlloc IN Kernel32.DLL Long, Long, Long
cDocumentName = 'Имя печатаемого документа'       && Имя документа
nLenDocumentName = LEN(cDocumentName)             && Длина строки
*
* Выделяем память в куче процесса
*
pMem = HeapAlloc(GetProcessHeap(), 8, nLenDocumentName)
```

```
SYS(2600, pMem, nLenDocumentName, cDocumentName)
*
* Начинаем формировать структуру DOCINFO
* cDocInfo — переменная, в которой формируется структура
*
cDocInfo = BINTOC(20,'4RS')                        && DOCINFO.cbSize
cDocInfo = cDocInfo + BINTOC(pMem,'4RS')  && DOCINFO.lpszDocName
cDocInfo = cDocInfo + REPLICATE(CHR(0),12)         && Остальные поля структуры
*
* Конец формирования структуры DOCINFO
* После окончания печати документа следует освободить
* распределенную память:
*
* HeapFree(GetProcessHeap())
*
```

Освобождение распределенной памяти

Применяя в ваших приложениях функции Windows API, вы должны помнить о том, что Visual FoxPro не может управлять памятью, распределяемой этими функциями. Поэтому если вы распределяете память, например, при помощи функции `GlobalAlloc`, то вы обязательно должны после использования вернуть эту память Windows, вызвав функцию `GlobalFree`. Аналогично, если вы распределяете память функцией `HeapAlloc`, то для возвращения памяти процессу используйте функцию `HeapFree`. В принципе для каждой API-функции, распределяющей память, имеется функция — антипод, возвращающая эту память.

А когда нужно освобождать распределенную память? Только тогда, когда исчезнет потребность в ее использовании. В случае распределения памяти для структур, содержащих указатели, освобождать эту память можно только после того, как будет завершено выполнение API-функции, использующей структуру.

Забывая возвращать распределенную память, вы рискуете столкнуться с проблемой, называемой утечкой памяти. В частности, если вы забудете вернуть Windows-память, распределенную функцией `GlobalAlloc`, то эта память в некоторых случаях может остаться недоступной даже после завершения вашего приложения. Последствия такой утечки могут быть весьма печальны — вплоть до краха операционной системы.

Применение функций Windows API

В этом разделе вы познакомитесь с некоторыми полезными функциями Windows API. Для каждой функции сначала приводится ее прототип, а затем — объявление в Visual FoxPro, и, наконец, пример использования. Основная задача этого раздела — научить вас, как, пользуясь описанием прототипа функции в MSDN, правильно объявить ее в Visual FoxPro и, самое главное, правильно применять.

Изменение раскладки клавиатуры

Функция `LoadKeyboardLayout` из библиотеки User32.dll загружает новую языковую раскладку клавиатуры. Вот ее прототип:

```
HKL LoadKeyboardLayout(
    LPCTSTR pwszKLID, // Наименование загружаемой языковой раскладки
    UINT Flags        // Флаг
    );
```

Объявление функции в Visual FoxPro:

```
DECLARE Long LoadKeyboardLayout IN WIN32API String KLID, Long Flags
```

Функция возвращает дескриптор раскладки клавиатуры (целое число) или ноль в случае ошибки.

Передаваемый функции параметр *KLID* определяет язык раскладки клавиатуры. В табл. 19.8 приведены некоторые возможные значения этого параметра.

Таблица 19.8. *Коды языковой раскладки*

Значение KLID	Описание	Значение KLID	Описание
"00000407"	Немецкий (стандартный)	"00000419"	Русский
"00000409"	Английский (США)	"00000422"	Украинский
"0000040C"	Французский	"00000423"	Белорусский
"0000040D"	Финский	"00000425"	Эстонский
"00000410"	Итальянский	"00000426"	Латвийский
"00000415"	Польский	"00000427"	Литовский

Второй параметр, *Flags*, указывает, как будет загружаться раскладка клавиатуры. Для приложений на Visual FoxPro его значение всегда равно единице.

Для удаления клавиатурной раскладки используется функция `UnloadKeyboardLayout`. Она также находится в библиотеке User32.dll. Вот ее прототип:

```
BOOL UnloadKeyboardLayout(HKL hkl);
```

Объявление функции в Visual FoxPro:

```
DECLARE Long UnloadKeyboardLayout IN WIN32API Long hkl
```

Передаваемый функции параметр *hkl* — это дескриптор клавиатурной раскладки, полученный при выполнении функции `LoadKeyboardLayout`. Функция возвращает отличное от нуля значение при успешном выполнении или ноль в случае ошибки.

В следующем примере загружается раскладка клавиатуры для немецкого языка:

```
HKL = LoadKeyboardLayout("00000407", 1)
```

А этот код удаляет загруженную ранее раскладку клавиатуры:

```
UnloadKeyboardLayout(HKL)
```

Информация о локальной дате и времени

Функция `GetLocalTime` из библиотеки Kernel32.dll возвращает текущие дату и время в формате UTC (Universal Time Coordinated). Вот ее прототип:

```
VOID GetLocalTime(
    LPSYSTEMTIME lpSystemTime // Адрес структуры SystemTime
    );
```

Тип данных `LPSYSTEMTIME` — это указатель на структуру `SystemTime`:

```
typedef struct _SYSTEMTIME {
    WORD wYear;             // Год
    WORD wMonth;            // Месяц
    WORD wDayOfWeek;        // День недели
    WORD wDay;              // День месяца
    WORD wHour;             // Часы
    WORD wMinute;           // Минуты
    WORD wSecond;           // Секунды
    WORD wMilliseconds;     // Миллисекунды
    } SYSTEMTIME;
```

Как видите, все поля структуры — это двухбайтовые целые числа. Месяцы нумеруются начиная с единицы, т. е. для января `wMonth`=1 и т. д. Номера дней недели начинаются с нуля, т. е. для воскресенья `wDayOfWeek`=0, для понедельника `wDayOfWeek`=1 и т. д.

Функция `GetLocalTime` не возвращает никаких значений. Вот ее объявление в Visual FoxPro:

```
DECLARE GetSystemTime IN WIN32API String @ SystemTime
```

В листинге 19.10 показан код пользовательской функции `LocalTime`, заполняющей массив значениями даты и времени.

```
FUNCTION LocalTime
PARAMETERS taTimeArray
    EXTERNAL ARRAY taTimeArray
    DIMENSION taTimeArray[8]
    DECLARE GetLocalTime IN WIN32API String @
    LOCAL lcSystemTime                  && Для структуры SYSTEMTIME
    lcSystemTime = REPLICATE(CHR(0),16) && Размечаем память для структуры
    GetLocalTime(@lcSystemTime)         && Заполняем поля структуры
*
* Заполняем переданный функции массив значениями полей структуры
*
    taTimeArray[1]=CTOBIN(LEFT(lcSystemTime,2),'2RS')      && Год
```

```
    taTimeArray[2]=CTOBIN(SUBSTR(lcSystemTime,3,2),'2RS')  && Месяц
    taTimeArray[3]=CTOBIN(SUBSTR(lcSystemTime,5,2),'2RS')  && День недели
    taTimeArray[4]=CTOBIN(SUBSTR(lcSystemTime,7,2),'2RS')  && День месяца
    taTimeArray[5]=CTOBIN(SUBSTR(lcSystemTime,9,2),'2RS')  && Часы
    taTimeArray[6]=CTOBIN(SUBSTR(lcSystemTime,11,2),'2RS') && Минуты
    taTimeArray[7]=CTOBIN(SUBSTR(lcSystemTime,13,2),'2RS') && Секунды
    taTimeArray[8]=CTOBIN(SUBSTR(lcSystemTime,15,2),'2RS') && Миллисек.
    RETURN
ENDFUNC
```

А в листинге 19.11 показан пример вызова этой функции из вашего приложения.

```
LOCAL laTime[1], laDayOfWeek[7], laMonth[12], lcResult
laDayOfWeek[1] = 'воскресенье'
laDayOfWeek[2] = 'понедельник'
laDayOfWeek[3] = 'вторник'
laDayOfWeek[4] = 'среда'
laDayOfWeek[5] = 'четверг'
laDayOfWeek[6] = 'пятница'
laDayOfWeek[7] = 'суббота'
laMonth[1] = 'января'
laMonth[2] = 'февраля'
laMonth[3] = 'марта'
laMonth[4] = 'апреля'
laMonth[5] = 'мая'
laMonth[6] = 'июня'
laMonth[7] = 'июля'
laMonth[8] = 'августа'
laMonth[9] = 'сентября'
laMonth[10] = 'октября'
laMonth[11] = 'ноября'
laMonth[12] = 'декабря'
= LocalTime(@laTime)
lcResult = 'Сейчас ' + STR(laTime[4],2) + ' ' + laMonth[laTime[2]] + ;
    STR(laTime[1],4) + ' года, ' + laDayOfWeek[laTime[3]+1] + CHR(13) + ;
    LTRIM(STR(laTime[5])) + ' час ' + LTRIM(STR(laTime[6])) + ' мин'
= MESSAGEBOX(lcResult)
```

Время, прошедшее с момента запуска системы

Функция `GetTickCount` из библиотеки Kernel32.dll возвращает количество миллисекунд, прошедших с момента запуска системы. Вот ее прототип:

```
DWORD GetTickCount(VOID)
```

Объявление функции в Visual FoxPro:

```
DECLARE Long GetTickCount IN WIN32API
```

Пример вызова функции из вашего приложения:

```
nMilliSeconds = GetTickCount()
```

Минимальный интервал времени в Windows — около 55 миллисекунд.

Приостановка выполнения приложения

Функция `Sleep` из библиотеки Kernel32.dll приостанавливает выполнение вашего приложения на указанное количество миллисекунд. Вот ее прототип:

```
VOID Sleep(
    DWORD dwMilliseconds // "Время сна" в миллисекундах
    );
```

А вот объявление этой функции в Visual FoxPro:

```
DECLARE Sleep IN WIN32API Long Milliseconds
```

В листинге 19.12 приведен пример вызова функции `Sleep` из вашего приложения.

```
FUNCTION ToSleep(tnMilliseconds)
    DECLARE Sleep IN WIN32API Long
    Sleep(tnMilliseconds)
ENDFUNC
```

Информация об использовании памяти

Функция `GlobalMemoryStatus` из библиотеки Kernel32.dll предоставляет информацию об использовании физической и виртуальной памяти компьютера. Вот ее прототип:

```
VOID GlobalMemoryStatus(
    LPMEMORYSTATUS lpBuffer // Адрес структуры MemoryStatus
    );
```

Тип данных `LPMEMORYSTATUS` — это указатель на структуру `MemoryStatus`:

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;        // Размер структуры
    DWORD dwMemoryLoad;    // Процент использования памяти
    DWORD dwTotalPhys;     // Физическая память, байт
    DWORD dwAvailPhys;     // Свободная физическая память, байт
    DWORD dwTotalPageFile; // Размер файла подкачки, байт
    DWORD dwAvailPageFile; // Свободных байт в файле подкачки
    DWORD dwTotalVirtual;  // Виртуальная память, используемая процессом
    DWORD dwAvailVirtual;  // Свободная виртуальная память
    } MEMORYSTATUS, *LPMEMORYSTATUS;
```

Функция не возвращает никаких значений. Вот ее объявление в Visual FoxPro:

```
DECLARE GlobalMemoryStatus IN WIN32API String @ MemoryStatus
```

В листинге 19.13 показан код пользовательской функции `MemoryInfo`, которая вызывает Windows API-функцию `GlobalMemoryStatus` и заполняет массив `taMemoryStatus` данными из структуры `MemoryStatus`.

```
FUNCTION MemoryInfo
PARAMETERS taMemoryStatus
    EXTERNAL ARRAY taMemoryStatus
    DIMENSION taMemoryStatus[7]
    LOCAL lcMemoryStatus
    DECLARE GlobalMemoryStatus IN WIN32API String @
*
* Выделяем память под структуру
*
    lcMemoryStatus = BINTOC(32,'4RS') + REPLICATE(CHR(0),28)
    GlobalMemoryStatus(@lcMemoryStatus)
    taMemoryStatus[1] = CTOBIN(SUBSTR(lcMemoryStatus,5,4),'4RS')
    taMemoryStatus[2] = CTOBIN(SUBSTR(lcMemoryStatus,9,4),'4RS')
    taMemoryStatus[3] = CTOBIN(SUBSTR(lcMemoryStatus,13,4),'4RS')
    taMemoryStatus[4] = CTOBIN(SUBSTR(lcMemoryStatus,17,4),'4RS')
    taMemoryStatus[5] = CTOBIN(SUBSTR(lcMemoryStatus,21,4),'4RS')
    taMemoryStatus[6] = CTOBIN(SUBSTR(lcMemoryStatus,25,4),'4RS')
    taMemoryStatus[7] = CTOBIN(SUBSTR(lcMemoryStatus,29,4),'4RS')
    RETURN
ENDFUNC
```

Структура `MemoryStatus` будет сформирована в переменной `lcMemoryStatus`. В первое поле структуры заносится значение ее длины (32 байта). Остальные поля заполняются нулями.

После выполнения функции `GlobalMemoryStatus` поля структуры `MemoryStatus` "расшифровываются" при помощи встроенной функции `CTOBIN`.

В листинге 19.14 показан пример вызова функции `GlobalMemory` из вашего приложения.

```
LOCAL lcInfo, laMemoryStatus[1]
DECLARE GlobalMemoryStatus IN WIN32API String @
MemoryInfo(@laMemoryStatus)
lcInfo = 'Физическая память, байт: ' + LTRIM(STR(laMemoryStatus[2])) + ;
          ', свободно: ' + LTRIM(STR(laMemoryStatus[3])) + CHR(13) + ;
          'Виртуальная память, байт: ' + LTRIM(STR(laMemoryStatus[6])) + ;
          ', свободно: ' + LTRIM(STR(laMemoryStatus[7]))
= MESSAGEBOX(lcInfo)
```

Получение информации о файловом томе

Функция `GetVolumeInformation` из библиотеки Kernel32.dll возвращает информацию о томе и установленной на нем файловой системе. Вот ее прототип:

```
BOOL GetVolumeInformation
    LPCTSTR lpRootPathName,
    LPTSTR  lpVolumeNameBuffer,
    DWORD   nVolumeNameSize,
    LPDWORD lpVolumeSerialNumber,
    LPDWORD lpMaximumComponentLength,
    LPDWORD lpFileSystemFlags,
    LPTSTR  lpFileSystemNameBuffer,
    DWORD   nFileSystemNameSize
    );
```

Как видите, функция использует указатели типов `LPTSTR` и `LPCTSTR`. Поэтому сразу можно предположить, что существуют две ее реализации для работы со строками форматов ANSI и Unicode.

Объявление функции `GetVolumeInformation` в Visual FoxPro:

```
DECLARE Long GetVolumeInformationA IN Kernel32.dll ;
    AS GetVolumeInformation ;
    String lpRootPathName, ;
    String @ lpVolumeNameBuffer, ;
    Long nVolumeNameSize, ;
    Long @ lpVolumeSerialNumber, ;
    Long @ lpMaximumComponentLength, ;
    Long @ lpFileSystemFlags, ;
    String lpFileSystemNameBuffer, ;
    Long nFileSystemNameSize
```

В объявлении указана реализация функции `GetVolumeInformationA`, работающая с символьными данными формата ANSI; ее псевдоним `GetVolumeInformation` вы будете использовать при вызове функции из своего приложения.

Рассмотрим типы и назначение параметров, передаваемых функции.

`lpRootPathName`

указатель типа `LPCTSTR` на строку, содержащую корневое имя тома, например, "C:\". Так как содержимое строки не должно изменяться, то в объявлении функции используем тип `String`.

`lpVolumeName`

указатель типа `LPTSTR` на строку, в которую будет помещена информация об имени тома. Передается по ссылке (в объявлении тип — String @).

`nVolumeNameSize`

четырехбайтовое целое число, в котором содержится значение длины строки, указываемой в *`lpVolumeName`*.

lpVolumeSerialNumber

указатель типа LPDWORD на целое число, в которое будет помещен серийный номер устройства. Передается по ссылке — в объявлении тип Long @.

lpMaximumComponentLength

указатель типа LPDWORD на целое число, в которое будет помещено значение, определяющее максимальную длину имени файла, поддерживаемую файловой системой. Передается по ссылке — в объявлении тип Long @.

lpFileSystemFlags

указатель типа LPDWORD на целое число, в которое будет помещено значение флага, определяющего характеристики используемой на томе файловой системы. Передается по ссылке — в объявлении тип Long @. Значение флага определяется как сумма перечисленных в табл. 19.9 битовых масок.

Таблица 19.9. *Битовые маски флага* `FileSystemFlags`

Наименование	Значение	Описание
FILE_NAMED_STREAMS	0x00040000	Файловая система поддерживает именованные потоки
FILE_READ_ONLY_VOLUME	0x00080000	Указанный том доступен только для чтения. Флаг не поддерживается в Windows 2000/NT и Windows ME/98/95
FILE_SUPPORTS_OBJECT_IDS	0x00010000	Файловая система поддерживает идентификаторы объектов
FILE_SUPPORTS_REPARSE_POINTS	0x00000080	Файловая система поддерживает точки соединения
FILE_SUPPORTS_SPARSE_FILES	0x00000040	Файловая система поддерживает распределенные файлы
FILE_VOLUME_QUOTAS	0x00000020	Файловая система поддерживает квотирование дискового пространства
FS_CASE_IS_PRESERVED	0x00000002	Файловая система поддерживает резервирование имен файлов
FS_CASE_SENSITIVE	0x00000001	Файловая система поддерживает чувствительные к регистру символа имена файлов
FS_FILE_COMPRESSION	0x00000010	Файловая система поддерживает сжатие файлов
FS_FILE_ENCRYPTION	0x00020000	Файловая система поддерживает шифрование данных (EFS)
FS_PERSISTENT_ACLS	0x00000008	Файловая система поддерживает управление доступом
FS_UNICODE_STORED_ON_DISK	0x00000004	Файловая система поддерживает формат Unicode для имен файлов
FS_VOL_IS_COMPRESSED	0x00008000	Поддерживается сжатие томов

lpFileSystemNameBuffer

указатель типа LPTSTR на строку, в которую будет помещено наименование файловой системы. Передается по ссылке, поэтому при объявлении используем для параметра тип String @.

nFileSystemNameSize

четырехбайтовое целое число, в котором содержится значение длины строки, указываемой в *lpFileSystemNameBuffer*.

В листинге 19.15 показан пример использования функции GetVolumeInformation.

```
DECLARE Long GetVolumeInformationA IN Kernel32.dll AS GetVolumeInfo ;
        String, String @, Long, Long @, Long @, Long @, String @, Long
cRootPathName = 'c:\'
nVolumeNameSize = 32
cVolumeName = SPACE(nVolumeNameSize)
nVolumeSerialNumber = 0
nMaximumComponentLength = 0
nFileSystemFlags = 0
nFileSystemNameSize = 16
cFileSystemName = SPACE(nFileSystemNameSize)
nResult = GetVolumeInfo(cRootPathName, @cVolumeName, nVolumeNameSize, ;
                        @nVolumeSerialNumber, @nMaximumComponentLength, ;
                        @nFileSystemFlags, @cFileSystemName, ;
                        nFileSystemNameSize)
```

При успешном выполнении функция возвращает отличное от нуля значение и ноль в случае ошибки.

Чем еще интересна эта функция? Например, если вы пытаетесь получить информацию о несуществующем томе (или отсутствующих дискете или компакт-диске), то будет выведено диалоговое окно, предлагающее пользователю указать правильное имя дисковода.

Соединение с сетевым ресурсом

Функция WNetAddConnection2 из библиотеки Mpr.dll создает соединение с сетевым ресурсом локальной компьютерной сети. Вот ее прототип:

```
DWORD WNetAddConnection2(
    LPNETRESOURCE lpNetResource,
    LPCTSTR lpPassword,
    LPCTSTR lpUsername,
    DWORD dwFlags
    );
```

Объявление функции в Visual FoxPro:

```
DECLARE Long WNetAddConnection2A IN Mpr.dll AS WNetAddConnection2 ;
        String lpNetResource, String lpPassword, ;
        String lpUsername, Long dwFlags
```

Передаваемые функции параметры:

lpNetResource

указатель на структуру NETRESOURCE.

lpPassword

указатель типа LPCTSTR на символьную строку, в которой передается пароль, используемый при создании нового сетевого соединения. Если этот параметр NULL, то функция использует заданный по умолчанию пароль пользователя, указанный в параметре *lpUserName*. Если *lpPassword* указывает на пустую строку, то пароль не используется. Для Windows ME/98/95 этот параметр не используется.

lpUsername

указатель типа LPCTSTR на символьную строку, содержащую имя пользователя. Если параметр — NULL, то функция использует имя пользователя, заданное по умолчанию.

dwFlags

целое число; определяет опции соединения. Значение параметра формируется как сумма перечисленных в табл. 19.10 битовых масок.

Таблица 19.10. *Битовые маски флага* `Flags`

Наименование	Значение	Описание
CONNECT_INTERACTIVE	0x00000008	Если этот флажок установлен, операционная система может взаимодействовать с пользователем, предлагая ему выбор из установок по умолчанию
CONNECT_PROMPT	0x00000010	Этот флажок указывает системе, что не нужно использовать любые установки по умолчанию для имени и пароля пользователя для того, чтобы предложить пользователю возможность выбора варианта. Этот флажок игнорируется, если флажок CONNECT_INTERACTIVE не установлен
CONNECT_REDIRECT	0x00000080	Если флажок установлен, то система автоматически определяет символ диска, связываемого с ресурсом. Для Windows XP выбор начинается с символа "Z", затем "Y" и т. д. в обратном алфавитном порядке. В предыдущих версиях Windows определение символа дисковода начиналось с "C" и продолжалось до "Z". Если символ дисковода указан в поле lpLocalName структуры NETRESOURCE, то значение флага игнорируется

CONNECT_UPDATE_PROFILE	0x00000001	Если флажок установлен, то операционная система пытается автоматически установить соединение при входе пользователя в систему. Запоминаются (и восстанавливаются) только успешные соединения
CONNECT_COMMANDLINE	0x00000800	Только для Windows XP. Если этот флажок установлен, операционная система запрашивает пользователя относительно установления его подлинности, используя командную строку вместо графического интерфейса. Этот флажок игнорируется, если флажок CONNECT_INTERACTIVE не установлен
CONNECT_CMD_SAVECRED	0x00001000	Только для Windows XP. Если этот флажок установлен, то операционная система запрашивает авторизацию. Этот флажок игнорируется, если вы не устанавливаете флажок CONNECT_COMMANDLINE

Если при установке соединения происходит сбой из-за недопустимого пароля, а флажок CONNECT_INTERACTIVE установлен, то функция выводит диалоговое окно для ввода правильного пароля.

Рассмотрим структуру NETRESOURCE. Вот ее описание:

```
typedef struct _NETRESOURCE {
    DWORD dwScope;
    DWORD dwType;
    DWORD dwDisplayType;
    DWORD dwUsage;
    LPTSTR lpLocalName;
    LPTSTR lpRemoteName;
    LPTSTR lpComment;
    LPTSTR lpProvider;
    } NETRESOURCE;
```

Согласно документации, перед вызовом функции WNetAddConnection2 в этой структуре необходимо установить значения только для следующих четырех полей:

dwType

целое число, определяет тип ресурса: 0 — все ресурсы, 1 — диски, 2 — принтеры;

lpLocalName

указатель типа LPTSTR на нуль-терминированную строку, в которой указывается имя локального устройства (например, "Q:");

lpRemoteName

указатель типа LPTSTR на нуль-терминированную строку, в которой указывается имя сетевого устройства (например, "\\ServerName\PubFolder");

lpProvider

указатель типа LPTSTR на нуль-терминированную строку, содержащую имя поставщика ресурса. Должен быть NULL, если имя поставщика ресурса неизвестно.

В листинге 19.16 показан код пользовательской функции `NetConnect`, использующей вызов функции `WNetAddConnection2` для создания локального устройства (файлового тома), связанного с указанным сетевым ресурсом.

```
FUNCTION NetConnect(tcLocalVolume, tcNetResource)
LOCAL lcLocalName, lcRemoteName, hMem1. hMem2, lcNetResource
LOCAL lcPassWord, lcUserName, lnFlag, lnResult
*
* nResult = NetConnect('Q:', '\\Serv001\D\AllUser')
*
DECLARE Long GetProcessHeap IN Kernel32.DLL
DECLARE Long HeapAlloc IN Kernel32.DLL Long, Long, Long
DECLARE Long HeapFree IN Kernel32.DLL Long, Long, Long @
DECLARE Long WNetAddConnection2A IN Mpr.dll AS WNetAddConnection ;
        String, String, String, Long
*
* Формирование указателя на строку с именем локального устройства
*
lcLocalName = tcLocalVolume + CHR(0)              && Имя локального диска
hMem1 = HeapAlloc(GetProcessHeap(), 0, LEN(lcLocalName))
SYS(2600, hMem1, LEN(lcLocalName), lcLocalName)
*
* Формирование указателя на строку с именем сетевого ресурса
*
lcRemoteName = tcNetResource + CHR(0)     && Имя сетевого ресурса
hMem2 = HeapAlloc(GetProcessHeap(), 0, LEN(lcRemoteName))
SYS(2600, hMem2, LEN(lcRemoteName), lcRemoteName)
*
* Формирование структуры NETRESOURCE
*
lcNetResource = BINTOC(0,'4RS') + ;      && dwScope
                BINTOC(1,'4RS') + ;      && dwType
                BINTOC(0,'4RS') + ;      && dwDisplayType
                BINTOC(0,'4RS') + ;      && dwUsage
                BINTOC(hMem1,'4RS') + ;  && lpLocalName
                BINTOC(hMem2,'4RS') + ;  && lpRemoteName
                BINTOC(0,'4RS') + ;      && lpComment
                BINTOC(0,'4RS'))         && lpProvider
lcPassWord = 'PassWord' + CHR(0)         && Ваш пароль
lcUserName = 'Login' + CHR(0)            && Ваш Login
lnFlag = 8                            && Выводить диалог при неверном пароле
*
* Установка соединения
*
lnResult = WNetAddConnection(lpNetResource,lcPassWord,lcUserName,lnFlag)
*
* Освобождаем распределенную память
*
HeapFree(GetProcessHeap(), 0, hMem1)
```

```
HeapFree(GetProcessHeap(), 0, hMem2)
RETURN lnResult
```

Так как структура NETRESOURCE содержит указатели на символьные строки, то необходимо распределить память под эти строки и получить указатели, значения которых должны быть указаны в полях структуры. Распределяем память процесса. Первый вызов функции HeapAlloc возвращает указатель hMem1, в распределенную функцией память заносится значение переменной lcLocalName. Второй вызов функции HeapAlloc возвращает указатель hMem2, а в распределенную память заносится значение переменной lcRemoteName. Значения указателей hMem1 и hMem2 помещаются в поля *lpLocalName* и *lpRemoteName* структуры.

Возвращаемые функцией значения перечислены в табл. 19.11.

Таблица 19.11. *Коды завершения функции WNetAddConnection2*

Код	Описание ошибки
0	Успешное завершение
5	Нет доступа к сетевому ресурсу

Таблица 19.11 *(окончание)*

Код	Описание ошибки
66	Тип локального ресурса не соответствует типу сетевого ресурса
67	Значение, определенное в lcRemoteName, неприемлемо для сетевого поставщика ресурса, или имя ресурса недопустимо, или именованный ресурс не может быть размещен
85	Локальное устройство, определенное в lcLocalName, уже соединено с сетевым ресурсом
86	Указанный пароль недопустим, а флажок CONNECT_INTERACTIVE в dwFlags не установлен
170	Маршрутизатор или провайдер занят
1200	Недопустимое значение в lcLocalName
1202	Вход для устройства, определенного в lcLocalName, уже указан в параметре пользователя
1203	Операция не может быть выполнена, потому что сетевой компонент не стартовал или потому что указанное имя не может использоваться
1204	Значение, определенное в lcProvider, не соответствует никакому поставщику
1205	Система не может открыть профиль пользователя для установки постоянного соединения
1206	Некорректный формат профиля пользователя
1208	Сбой в сети
1222	Сеть отсутствует

1223	Попытка установить соединение была отменена пользователем через диалоговое окно от одного из сетевых поставщиков ресурса, или вызываемым ресурсом

Функция `WNetCancelConnection2` разрывает указанное соединение с сетевым ресурсом. Вот ее прототип:

```
DWORD WNetCancelConnection2(
    LPCTSTR lpName,
    DWORD dwFlags,
    BOOL fForce
    );
```

Параметр *lpName* — указатель на строку, содержащую наименование связанного с ресурсом логического диска.

Параметр *dwFlags* может принимать значения 0 или 1:

- `dwFlag`=0. Операционная система не модифицирует информацию относительно соединения.

  Если соединение было отмечено как постоянное, то оно будет восстановлено при следующем входе в систему.

- `dwFlag`=1. Система не будет восстанавливать это в последующих операций входа в систему.

Параметр *fForce* определяет, как будет выполнен разрыв соединения. Если `fForce`=0, то функция разрывает соединение независимо от того, существуют ли в данный момент открытые файлы или выполняется иной обмен данными с ресурсом. Иначе — функция возвращает ошибку, соединение не разрывается.

Объявление функции в Visual FoxPro:

```
DECLARE Long WNetCancelConnection2A IN Mpr.dll AS WNetCancelConnection ;
        String lpName, Long dwFlag, Long fForce
```

Пример вызова функции из вашего приложения:

```
cDeviceMame = 'Q:' + CHR(0)
nResult = WNetCancelConnection2(cDeviceName, 1, 0)
```

Функции для работы с GUID

GUID (Global Unique ID), он же UUID, ClsID и IID — глобальный идентификатор, представляющий собой двоичный тип данных длиной 16 байт (128 бит). Microsoft гарантирует уникальность значения GUID на всей Земле в течение обозримого будущего.

В Windows имеются функции, генерирующие и преобразующие значения уникальных идентификаторов из двоичного представления в специальный формат. Здесь мы рассмотрим три из них:

- `CoCreateGUID`
- `StringFromGUID2`
- `CLSIDFromString`

Эти функции находятся в библиотеке Ole32.dll.

Функция `CoCreateGUID` генерирует новое значение GUID в виде двоичной строки длиной 16 байт. Вот ее прототип:

```
HRESULT CoCreateGuid(GUID *pguid);
```

Объявление функции в Visual FoxPro:

```
DECLARE Long CoCreateGuid IN Ole32.dll String @ guid
```

Функция возвращает ноль при успешном выполнении либо отличное от нуля значение в случае ошибки.

Полученное двоичное значение GUID может быть использовано как уникальный индекс. Хранить такие индексы можно в полях таблиц типа Character Binary.

Функция `StringFromGUID2` преобразует двоичное значение GUID в его символьное представление, которое, например, может выглядеть так:

{c200e360-38c5-11ce-ae62-08002b2b79ef}.

Вот ее прототип:

```
int StringFromGUID2(
    REFGUID rguid,   // Двоичное значение GUID
    LPOLESTR lpsz,   // Указатель на строку с символьным
                     // значением GUID (в формате Unicode)
    int cchMax       // Количество 16-битовых символов в строке
    );
```

Объявление этой функции в Visual FoxPro выглядит так:

```
DECLARE Long StringFromGUID2 IN Ole32.dll ;
        String guid, String @ unicodeGuid, Integer cchMax
```

Функция возвращает количество символов, включая нулевой завершающий символ, в случае успешного выполнения, или ноль, если произошла ошибка.

Формируемое функцией представление GUID может быть использовано, например, для формирования веток ClsID в реестре Windows. В листинге 19.17 показан пример кода, демонстрирующий применение функций `CoCreateGuid` и `StringFromGUID2`.

```
DECLARE Long CoCreateGuid IN Ole32.dll String @
DECLARE Long StringFromGUID2 IN Ole32.dll String, String @, Integer
qGuid = REPLCATE(CHR(0), 16)
```

```
CoCreateGuid(@qGuid)
cGuid = REPLICATE(CHR(0), 80)
nResult = StringFromGUID2(qGuid, @cGuid, 40)
IF nResult > 0
    cGuid = STRCONV(LEFT(cGuid,(nResult-1)*2,6)
ENDIF
```

Функция `CLSIDFromString` преобразует символьное представление уникального идентификатора в двоичную 16-байтовую строку. Вот ее прототип:

```
HRESULT CLSIDFromString(
   LPOLESTR lpsz, // Указатель на строку с символьным представлением GUID
   LPCLSID pclsid // Указатель на строку с двоичным представлением GUID
   );
```

Объявление функции в Visual FoxPro:

```
DECLARE Long CLSIDFromString IN Ole32.dll ;
        String unicodeGuid, String @ qGuid
```

Функция возвращает ноль при успешном выполнении или отличное от нуля значение в случае ошибки.

В листинге 19.18 показан пример использования функции `CLSIDFromString`.

```
*
* Предполагается, что в переменной cGuid находится символьное
* представление GUID
*
DECLARE Long StringToGuid IN Ole32.dll String, String @
qGUID = REPLICATE(CHR(0), 16)
nResult = StringToGuid(STRCONV(cGuid,5), @qGUID)
```

В результате выполнения кода в переменную `qGUID` будет записано двоичное значение GUID.

Функция *ShellExecute*

Используя функцию `ShellExecute` из библиотеки Shell32.dll, вы можете запустить на выполнение любое приложение Windows, а также инициировать процедуру поиска файлов или просмотра папок.

Прототип функции:

```
HINSTANCE ShellExecute(
    HWND hwnd,
    LPCTSTR lpOperation,
    LPCTSTR lpFile,
```

```
    LPCTSTR lpParameters,
    LPCTSTR lpDirectory,
    INT nShowCmd
    );
```

Объявление функции в Visual FoxPro:

```
DECLARE Long ShellExecuteA IN Shell32.dll AS ShellExecute ;
        Long hwnd, String Operation, String File, ;
        String Parameters, String Directory, Integer ShowCmd
```

Функции передаются следующие параметры:

Hwnd

целое число, содержащее дескриптор *hWnd* родительского окна. Этому окну будут посылаться сообщения от окон диалогов, вызываемых функцией. По умолчанию — ноль;

Operation

указатель типа LPCTSTR на строку, которая может содержать одно из следующих значений: "find", "explore", "edit", "open" или "print";

File

указатель типа LPTSTR на строку, содержащую в зависимости от значения параметра Operation имя файла или папки;

Parameters

указатель типа LPCTSTR на строку, содержащую список параметров, передаваемых загружаемому приложению;

Directory

указатель типа LPCTSTR на строку, содержащую путь к файлу, указанному в *File*;

ShowCmd

целое число, значение которого определяет вид главного окна загружаемого приложения. Возможные значения параметра перечислены в табл. 19.12.

Таблица 19.12. *Значения параметра* ShowCmd

Значение	Описание
0	Скрывает окно загружаемого приложения и активизирует другое окно
1	Отображает главное окно приложения и делает его активным. Если окно приложения минимизировано или максимизировано, Windows восстанавливает его первоначальный размер и позицию
2	Окно загружаемого приложения минимизировано
3	Распахивает окно приложения на весь экран и делает его активным

4	Отображает окно приложения в его последних сохраненных размерах, но не делает его активным

Ниже показано, как выглядит результат выполнения функции при различных значениях параметра *Operation*.

Если *Operation*="find", то функция выводит диалоговое окно для поиска файлов по условиям (рис. 19.2). Параметр *File* должен указывать путь к папке, начиная с которой будет выполняться поиск. Параметр *ShowCmd* должен быть равен единице, остальные параметры игнорируются.

Если *Operation*="explore", то функция выводит стандартное диалоговое окно — список папок и файлов (рис. 19.3). Параметр *ShowCmd* должен быть равен единице, остальные параметры игнорируются.

Если *Operation*="edit", то функция открывает файл на редактирование, загружая приложение, ассоциированное с расширением указанного файла. Параметр *Edit* должен содержать имя файла, а параметр *Directory* — путь к этому файлу; если параметр *Directory* пустой, то параметр *Edit* должен содержать полную спецификацию файла (путь и имя).

Если *Operation*="open", то функция выполняет следующие действия: если в `File` указан исполняемый файл (например, типа EXE), то он запускается на выполнение; загружаемой программе передается список параметров, указанных в `Parameters`; в противном случае файл открывается на редактирование.

Если *Operation*="print", то выполняется печать файла на принтере (фактически загружается ассоциированное с расширением файла приложение, которое и печатает документ).

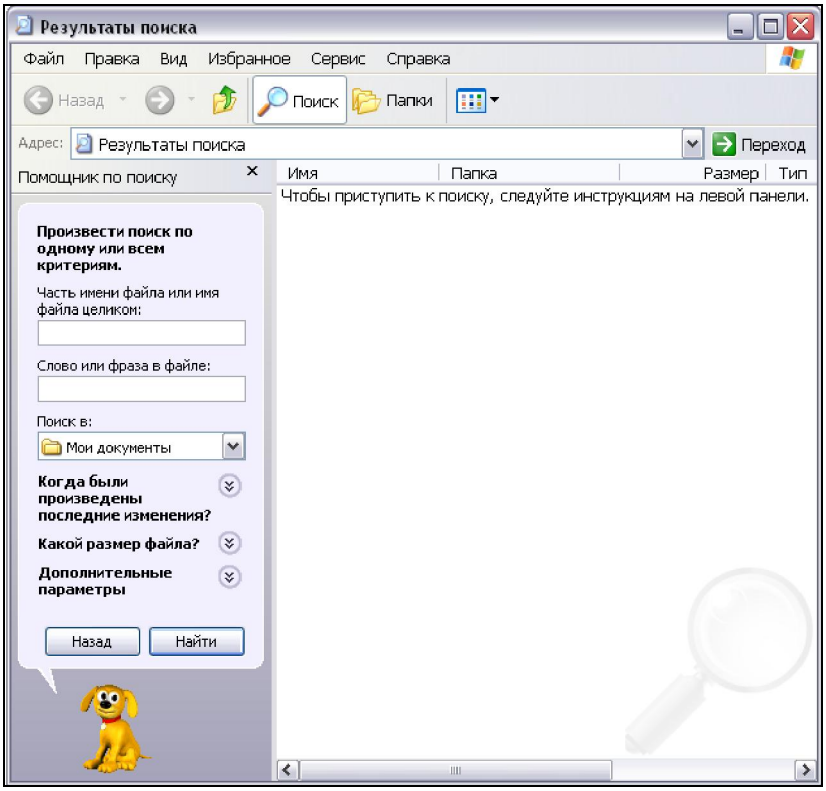

Рис. 19.2. Диалог для поиска файлов

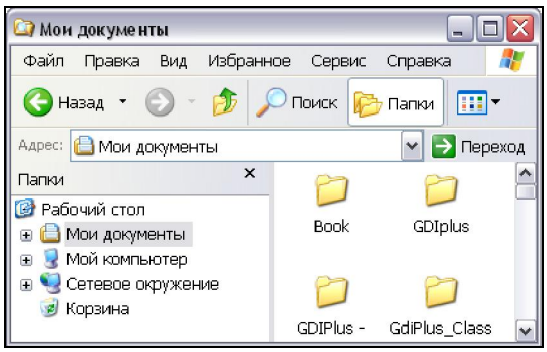

Рис. 19.3. Диалог для выбора папки/файла

В случае успешного завершения функция возвращает значение, большее 32. При возникновении ошибки функция вернет одно из значений, перечисленных в табл. 19.13.

Таблица 19.13. *Коды завершения функции* `ShellExecute`

Возвращаемое значение	Описание
0	Недостаточно памяти или ресурсов Windows
2	Указанный файл не найден
3	Указанный путь не существует
5	Операционная система не имеет доступа к указанному файлу
26	Невозможен совместный доступ к файлу
27	Невозможно загрузить приложение, ассоциированное с типом файла
28	DDE-транзакция не может быть завершена из-за истечения времени
29	Неудача при выполнении DDE-транзакции
30	DDE-транзакция не может быть завершена, т. к. обрабатываются другие DDE-транзакции
31	Нет никакого приложения, ассоциированного с расширением файла
32	Не найдена указанная DLL-библиотека

В листинге 19.19 показаны примеры использования функции `ShellExecute`.

```
*
* Запускается приложение Notepad.exe; его окно распахивается на весь
* экран
*
nReturn = ShellExecute(0,'open','c:\Windows\Notepad.exe',NULL,NULL,3)
*
* Запускается приложение WinWord и в него загружается документ
* c:\MyDoc\MyDocument.doc
*
nReturn = ShellExecute(0,'open','MyDocument.doc', NULL,'c:\MyDocs',1)
*
* Приложение WinWord запускается в фоновом режиме; в него загружается
* и направляется на печать документ c:\MyDocs\MyDocument.doc
*
nReturn = ShellExecute(0,'print','c:\MyDocs\MyDocument.doc',NULL,NULL,0)
```

Функция *SHFileOperation*

Функция `SHFileOperation` из библиотеки Shell32.dll выполняет копирование, перемещение, переименование и удаление объектов файловой системы (папок и файлов). В процессе выполнения функции выводится стандартное окно Windows для отображения процесса, подобное показанному на рис. 19.4.

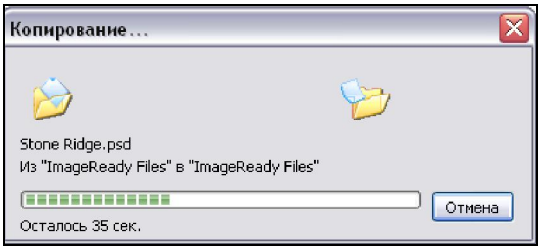

Рис. 19.4. Отображение процесса копирования файлов

Вот ее прототип:

```
int SHFileOperation(LPSHFILEOPSTRUCT lpFileOp);
```

LPSHFILEOPSTRUCT — это указатель на структуру SHFILEOPSTRUCT:

```
typedef struct _SHFILEOPSTRUCT{
    HWND hwnd;
    UINT wFunc;
    LPCSTR pFrom;
    LPCSTR pTo;
    FILEOP_FLAGS fFlags;
    BOOL fAnyOperationsAborted;
    LPVOID hNameMappings;
    LPCSTR lpszProgressTitle;
    } SHFILEOPSTRUCT, FAR *LPSHFILEOPSTRUCT;
```

Описание полей структуры приведено в табл. 19.14.

Таблица 19.14. *Описание полей структуры* SHFILEOPSTRUCT

Наименование поля	Описание
hwnd	Дескриптор родительского окна. Это может быть hWnd формы или главного окна Visual FoxPro
wFunc	Код выполняемой операции. Может принимать одно из следующих значений: wFunc=1 — перемещение файла; wFunc=2 — копирование файлов и папок; wFunc=3 — удаление файлов; wFunc=4 — переименование файлов
pFrom	Указатель типа LPCSTR на строку, в которой перечислены один или большее количество наименований исходных файлов (возможно, с указанием путей). Если используется более одного наименования файла, то каждая последующая спецификация отделяется от предыдущей нулевым байтом. Строка должна завершаться двумя нулевыми байтами

Таблица 19.14 *(окончание)*

Наименование поля	Описание
pTo	Указатель типа LPCSTR на строку, в которой указана конечная папка для перемещения (копирования) файлов. Строка должна завершаться двумя нулевыми байтами
fFlags	Двухбайтовое целое число, значение которого определяет поведение функции. Значение вычисляется как сумма битовых масок
fAnyOperationsAborted	32-разрядное целое число; получает значение "ноль", если пользователь прервал операцию над файлами
hNameMappings	32-разрядное целое число; в приложениях на Visual FoxPro должно иметь нулевое значение
lpszProgressTitle	Указатель типа LPCSTR на строку, содержащую текст, выводимый над шкалой прогресса. Если ноль, то выводятся строки с именами копируемых файлов

Ниже рассмотрено поведение функции в зависимости от значения поля fFlags структуры SHFILEOPSTRUCT.

fFlags=512.

если папка, в которую необходимо копировать файлы, не существует, то она будет автоматически создана. Иначе — выводится диалоговое окно с запросом о создании папки (рис. 19.5).

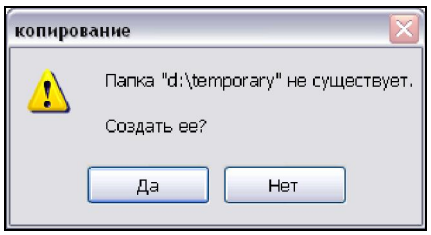

Рис. 19.5. Запрос на создание новой папки

fFlag=256

отображает шкалу прогресса, но не показывает имена файлов.

fFlags=128

операция выполняется только над файлами указанной папки, если для имени и типа файла указан шаблон (*.*). Над вложенными папками никаких действий не выполняется.

fFlags=8

если при копировании, перемещении или переименовании файл с указанным именем уже существует в папке-адресате, то он будет заменен без предупреждающего сообщения. Иначе — выводится диалог для подтверждения замены (рис. 19.6).

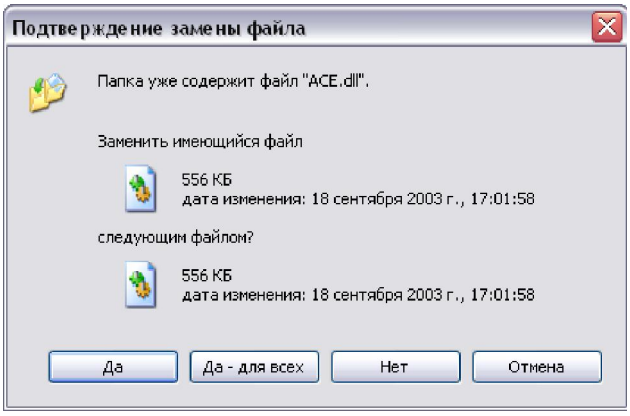

Рис. 19.6. Диалог подтверждения замены существующих файлов

`fFlags`=4

операция выполняется, окно отображения процесса не выводится.

Вот объявление функции `SHFileOperation` в Visual FoxPro:

```
DECLARE Long SHFileOperation IN Shell32.dll String @ SHFILEOPSTRUCT
```

Структура `SHFILEOPSTRUCT` может быть сформирована в символьной переменной длиной 26 байт; если вы хотите отслеживать ситуацию, когда пользователь отказался от файловой операции в процессе ее выполнения, то передавайте эту переменную по ссылке, т. к. вам в этом случае потребуется проанализировать значение поля *fAnyOperationsAborted*.

В листинге 19.20 приведен код пользовательской функции `FileOperation`, которая выполняет файловые операции, используя возможности, предоставляемые функцией `SHFileOperation`. Функция возвращает `.T.` (истину) при успешном выполнении операции.

```
FUNCTION FileOperation
LPARAMETER tcFrom, tcTo, tnOper
*
*  tcFrom — спецификация исходных файлов
*  tcTo   — спецификация результирующих файлов (папки)
*  tnOper — определяет тип операции.
*             tnOper=1 — переместить файлы
*             tnOper=2 — копировать файлы
*             tnOper=3 — удалить файлы
*             tnOper=4 — переименовать файл
*
LOCAL lcSHFO, lcFrom, lnLenFrom, lcTo, lnLenTo, hGlobalFrom, hGlobalTo
```

```
LOCAL lnFlag, lnReturn
DECLARE Long SHFileOperation IN Shell32.dll String @
DECLARE Long GlobalAlloc IN WIN32API Long, Long
DECLARE Long GlobalFree IN WIN32API Long
*
* Начинаем формировать структуру в переменной lcSHFO
*
lcSHFO = BINTOC(thisform.HWnd, '4RS')      && Поле hwnd структуры
lcSHFO = lcSHFO + BINTOC(tnOper, '4RS')    && Поле wFunc — вид операции
*
* Обработка спецификации исходных файлов
*
tcFrom = tcFrom + CHR(0) + CHR(0)             && Дописываем нули
lnLenFrom = LEN(tcFrom)                       && Длина исходной строки
hGlobalFrom = GlobalAlloc(0x0040, lnLenFrom)  && Выделяем для нее
                                              && блок памяти
SYS(2600, hGlobalFrom, lnLenFrom, tcFrom)     && и копируем туда строку
lcSHFO = lcSHFO + BINTOC(hGlobalFrom, '4RS')  && Поле pFrom
*
* Обработка спецификации результирующих файлов
*
IF tnOper = 3
    lcSHFO = lcSHFO + BINTOC(0, '4RS') && Если операции удаления
ELSE
    tcTo = tcTo + CHR(0) + CHR(0)      && Дописываем нули
    lnLenTo = LEN(tcTo)                && Длина результирующей строки
    hGlobalTo = GlobalAlloc(0x0040, lnLenTo)   && Выделяем для нее
                                               && блок памяти
    SYS(2600, hGlobalTo, lnLenTo, tcTo)        && и копируем туда строку
    lcSHFO = lcSHFO + BINTOC(hGlobalTo, '4RS') && Поле pTo структуры
ENDIF
lnFlag = 8 + 512
lcSHFO = lcSHFO + BINTOC(lnFlag, '2RS') && Поле fFlags структуры
lcSHFO = lcSHFO + REPLICATE(CHR(0), 12) && Последние 3 поля структуры
*
* Выполняем операцию
*
lnReturn = SHFileOperation(@lcSHFO)
*
* Возвращаем память Windows
*
GlobalFree(hGlobalFrom)
IF tnOper != 3
    GlobalFree(hGlobalTo)
ENDIF
*
* Если lnReturn = 0, то операция завершена успешно
*
IF lnReturn != 0
    RETURN .f.
ENDIF
```

```
RETURN .t.
```

Проанализируем код.

Структура SHFILEOPSTRUCT для функции SHFileOperation формируется в переменной *lcSHFO*. Записываем в первые четыре байта переменной значение *hwnd*, в следующие 4 байта — значение *wFunc*, определяющее характер операции. Поля *pFrom* и *pTo* содержат указатели на строки, содержащие информацию о спецификациях исходных и конечных файлов. Каждая такая строка должна заканчиваться двумя нулевыми байтами. Такой вид окончания строки используется потому, что при задании спецификации, например, исходных файлов, могут быть заданы файлы, расположенные в различных папках. В этом случае спецификации этих файлов разделяются одиночным нулевым байтом. Например:

```
cFiles = 'c:\Fld1\File1.typ'+CHR(0)+'d:\Fld2\File2.typ'+CHR(0)+CHR(0)
```

Выделяем блок в глобальной памяти Windows и получаем указатель на него:

```
hGlobalFrom = GlobalAlloc(0x0040, lnLenFrom)
```

Переписываем в выделенный блок памяти значение строки спецификации файла:

```
SYS(2600, hGlobalFrom, lnLenFrom, tcFrom)
```

Помещаем полученный указатель в поле структуры. Для преобразования целочисленного значения указателя в строку используем функцию BINTOC:

```
lcSHFO = lcSHFO + BINTOC(hGlobalFrom, '4RS')
```

Аналогичные действия выполняем и для получения указателя на строку, содержащую спецификацию конечных файлов — если код операции на удаление файлов. Для операции удаления вместо указателя помещаем в структуру ноль.

Далее формируем значение флага fFlags и добавляем его в структуру. Обратите внимание, что поле для хранения флага — двухбайтовое:

```
lnFlag = 8 + 512
lcSHFO = lcSHFO + BINTOC(lnFlag, '2RS')
```

Указанные значения флага определяют, что функция будет запрашивать необходимость создания новой папки и не будет запрашивать необходимость замены при одинаковых именах и типах исходных и конечных файлов.

Дописываем в переменную *lcSHFO* 12 нулевых байтов — это поля структуры *fAnyOperationsAborted*, *hNameMappings* и *lpszProgressTitle*.

После выполнения файловой операции возвращаем Windows-память, выделенную функциями GlobalAlloc:

```
GlobalFree(hGlobalFrom)
GlobalFree(hGlobalTo)
```

Вы можете проверить содержимое поля *fAnyOperationsAborted*, которое после успешного выполнения функции будет содержать ноль либо отличное от нуля значение, если операция была прервана пользователем. Вот этот код:

```
IF CTOBIN(SUBSTR(lcSHFO, 19, 4), '2RS') != 0
* Код, выполняющийся, если операция прервана пользователем
ENDIF
```

Заключение

Возможно, что после знакомства с материалом этой главы у вас сложилось впечатление, что работать с функциями Windows API в Visual FoxPro достаточно просто. И да, и нет. Например, некоторые Windows API-функции используют типы данных, не поддерживаемые в команде `DECLARE..DLL`. Так же есть ряд функций, называемых функциями обратного вызова. В прототипе такой функции перед описанием возвращаемого типа присутствует модификатор `CALLBACK`. К сожалению, вы не можете использовать такие функции, по крайней мере, напрямую. Так же встречаются ситуации, когда структура содержит указатель на функцию — это так же функции обратного вызова; их можно вызывать в Visual Basic и даже в VBScript, но разработчики не посчитали нужным реализовать эту возможность в Visual FoxPro.

Тем не менее, надеемся, что теперь вы сможете достаточно свободно ориентироваться в мире функций Windows API. В следующих главах книги мы постоянно будем применять различные API-функции, но уже без такого подробного обсуждения, как в этой главе.