

Библиотека системного программиста

---

двадцать седьмой том

---

© А.В. Фролов, Г.В. Фролов, 1996

Программирование для  
Windows NT

(часть вторая)

---

## АННОТАЦИЯ

В этой книге продолжается рассказ о программировании для операционной системы Microsoft Windows NT, начатый в предыдущем томе "Библиотеки системного программиста". Мы продолжим изучение файловой системы, в частности, рассмотрим работу с файлами, отображаемыми на память. Отдельная глава книги будет посвящена организации передачи данных между параллельно работающими процессами. Мы также расскажем вам о библиотеках динамической загрузки DLL, программном интерфейсе, предназначенном для создания многоязычных приложений и сервисных процессах.

Практически все сведения, изложенные в книге, пригодятся вам и при создании приложений для операционной системы Windows 95.

В книге вы найдете много исходных текстов приложений, которые можно приобрести отдельно на дискете.

## ВВЕДЕНИЕ

В предыдущем, 26 томе “Библиотеки системного программиста” мы начали изучение одной из наиболее перспективных операционных систем - Microsoft Windows NT. Мы рассказали вам о системе управления памятью, мультизадачности, рассмотрели проблемы синхронизации параллельно работающих задач и процессов, а также начали изучение файловой системы NTFS. Однако все это - только капля в море. Для того чтобы научиться создавать современные приложения для Microsoft Windows NT, вам предстоит узнать еще много нового.

В первой главе нашей новой книги мы продолжим изучение файловой системы NTFS. Прежде всего мы расскажем вам о принципиально новом для Windows способе работы с файлами - отображении их на виртуальную память. Как вы увидите, этот способ значительно упрощает выполнение файловых операций, сводя их в большинстве случаев к работе с оперативной памятью. При этом приложение не вызывает явно функции для чтения данных из файла или для записи их в файл, так как за него это делает операционная система. Причем она это делает высокоэффективными средствами, которые используются для работы с виртуальной памятью.

Помимо файлов, отображаемых на память, в первой главе мы также продолжим изучение традиционных методов работы с файлами. В частности, мы приведем исходные тексты и описания приложений, использующих для работы с файловой системой и файлами средства, описанные нами в предыдущем томе “Библиотеки системного программиста”.

Вторая глава посвящена организации передачи данных между процессами, работающими параллельно. Сложность здесь заключается в том, что такие процессы работают в изолированных адресных пространствах. В результате процессы не могут просто создавать глобальные области памяти, доступные всем приложениям (как это можно было делать в среде Microsoft Windows версии 3.1), а вынуждены пользоваться специальными средствами взаимодействия процессов, встроенными в Microsoft Windows NT.

Мы рассмотрим такие средства, как специальные сообщения, предназначенные для передачи данных между процессами, каналы передачи данных, создаваемые между процессами. Кроме того, для передачи данных между процессами можно использовать файлы, отображаемые на память. Этот способ мы также изучим на примере конкретного приложения.

Третья глава посвящена библиотекам динамической компоновки DLL, которые работают в Microsoft Windows NT совсем не так, как это было в среде Microsoft Windows версии 3.1.

В четвертой главе мы расскажем о том, как создавать приложения Microsoft Windows NT, способные работать с несколькими национальными языками и научим программно переключать раскладки клавиатуры.

Пятая глава посвящена организации сервисных процессов, которые используются в качестве драйверов или для решения других задач, таких, например, как создание систем управления базами данных.

Примеры приложений, приведенные в книге, транслировались в системе разработки Microsoft Visual C++ версии 4.0. Вы также можете воспользоваться версией 2.0, 4.1, 4.2 или 4.2 Enterprise Edition этой системы. Для того чтобы не набирать исходные тексты вручную и избежать ошибок, мы рекомендуем приобрести дискету с исходными текстами приложений, которая продается вместе с книгой.

---

## БЛАГОДАРНОСТИ

Авторы выражают благодарность сотруднику фирмы Interactive Products Inc. Максиму Синеву за многочисленные консультации.

Мы также благодарим корректора Кустова В. С. и сотрудников издательского отдела АО “Диалог-МИФИ” Голубева О. А., Голубева А. О., Дмитриеву Н. В., Виноградову Е. К., Кузьминову О. А.

## КАК СВЯЗАТЬСЯ С АВТОРАМИ

Вы можете передать нам свои замечания и предложения по содержанию этой и других наших книг через электронную почту:

<i>Сеть</i>	<i>Наш адрес</i>	<i>Сеть</i>	<i>Наш адрес</i>
Relcom	frolov@glas.apc.org	CompuServe	>internet: frolov@glas.apc.org
GlasNet	frolov@glas.apc.org	UUCP	cdp!glas!frolov
Internet	frolov@glas.apc.org		

Если электронная почта вам недоступна, присылайте ваши отзывы в АО “Диалог-МИФИ” по адресу:

115409, Москва, ул. Москворечье, 31, корп. 2,  
тел. 324-43-77

Приносим свои извинения за то что не можем ответить на каждое письмо. Мы также не занимаемся рассылкой дискет и исходных текстов к нашим книгам. По этому вопросу обращайтесь непосредственно в издательство “Диалог-МИФИ”.

# 1 СНОВА О ФАЙЛАХ

В последней главе 26 тома “Библиотеки системного программиста”, который называется “Программирование для Windows NT. Часть первая”, мы рассказали вам о том, как приложения Microsoft Windows NT работают с файлами. При этом мы привели краткое описание функций программного интерфейса операционной системы, выполняющие “классические” файловые операции, такие как открывание и закрывание файла, чтение блока данных из файла в буфер, расположенный в оперативной памяти, запись содержимого такого буфера в файл и так далее. Все эти операции знакомы вам по операционным системам MS-DOS и Microsoft Windows версии 3.1.

Что же касается операционных систем Microsoft Windows NT и Microsoft Windows 95, то в них появилось новое мощное средство, предназначенное для работы с файлами - файлы, отображаемые в память. Это средство, кстати, может использоваться еще и для обмена данными между параллельно работающими процессами, значительно упрощает программирование файловых операций, сводя их к работе с оперативной памятью.

Например, вы можете создать файл, содержащий записи реляционной базы данных. Открыв затем такой файл с использованием отображения на память, приложение может адресоваться к записям как к элементам массива, расположенного в оперативной памяти. При этом операционная система при необходимости будет самостоятельно выполнять чтение данных из файла и запись данных в файл без специальных усилий со стороны приложения.

После того как мы изучим методику работы с файлами, отображаемыми на память, мы приведем примеры приложений, работающих с файловой системой обычными средствами.

## Файлы, отображаемые на память

В операционную систему Microsoft Windows NT встроен эффективный механизм виртуальной памяти, описанный нами в предыдущем томе “Библиотеки системного программиста”. При использовании этого механизма приложениям доступно больше виртуальной оперативной памяти, чем объем физической оперативной памяти, установленной в компьютере.

Как вы знаете, виртуальная память реализована с использованием обычной оперативной памяти и дисковой памяти. Когда приложение обращается к странице виртуальной памяти, отсутствующей в физической памяти, операционная система автоматически читает ее из файла виртуальной памяти в физическую память и предоставляет приложению.

Если же приложение изменяет содержимое страницы памяти в физической оперативной памяти, операционная система сохраняет такую страницу в файл виртуальной памяти.

Почему мы вспомнили о виртуальной памяти в главе, посвященной файлам?

Потому что механизм работы с файлами, отображаемыми на память, напоминает механизм работы виртуальной памяти.

Напомним, что в операционной системе Microsoft Windows NT каждому процессу выделяется 2 Гбайта адресного пространства. Любой фрагмент этого пространства может быть отображен на фрагмент файла соответствующего размера.

На рис. 1.1 показано отображение фрагмента адресного пространства приложения, размером в 1 Гбайт, на фрагмент файла, имеющего размер 5 Гбайт.

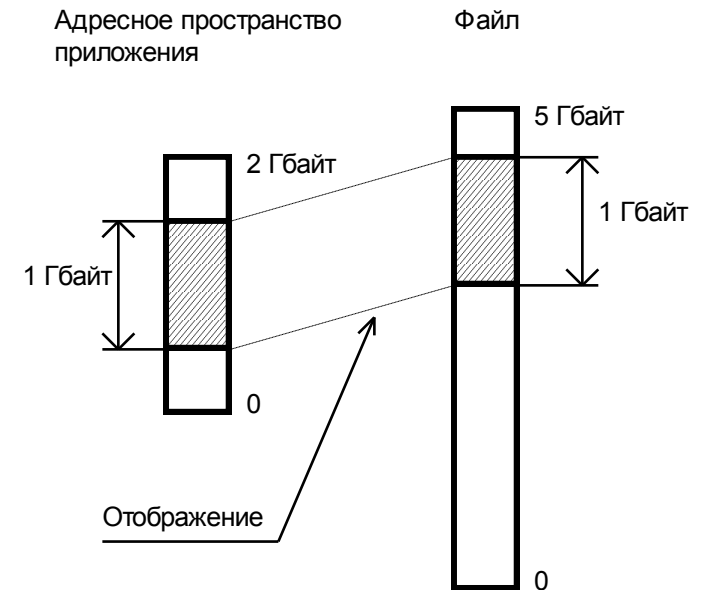


Рис. 1.1. Отображение фрагмента адресного пространства приложения на файл

С помощью соответствующей функции программного интерфейса, которую мы скоро рассмотрим, приложение Microsoft Windows NT может выбрать любой фрагмент большого файла для отображения в адресное пространство. Поэтому, несмотря на ограничение адресного пространства величиной 2 Гбайт, вы можете отображать (по частям) в это пространство

файлы любой длины, возможной в Microsoft Windows NT. В простейшем случае при работе с относительно небольшими файлами вы можете выбрать в адресном пространстве фрагмент подходящего размера и отобразить его на начало файла.

Как выполняется отображение фрагмента адресного пространства на фрагмент файла?

Если установлено такое отображение, то операционная система обеспечивает тождественность содержимого отображаемого фрагмента памяти и фрагмента файла, выполняя при необходимости операции чтения и записи в файл (с буферизацией и кешированием).

В процессе отображения адресного пространства память не выделяется, а только резервируется. Поэтому отображение фрагмента размером 1 Гбайт не вызовет переполнение файлов виртуальной памяти. Более того, при отображении файлы виртуальной памяти вообще не используются, так как страницы фрагмента связываются с отображаемым файлом.

Если приложение обращается в отображенный фрагмент для чтения, возникает исключение. Обработчик этого исключения загружает в физическую оперативную память соответствующую страницу из отображенного файла, а затем возобновляет выполнение прерванной команды. В результате в физическую оперативную память загружаются только те страницы, которые нужны приложению.

При записи происходит аналогичный процесс. Если нужной страницы нет в памяти, она подгружается из отображенного файла, затем в нее выполняется запись. Загруженная страница остается в памяти до тех пор, пока не будет вытеснена другой страницей при нехватке физической оперативной памяти. Что же касается записи измененной страницы в файл, то эта запись будет выполнена при закрытии файла, по явному запросу приложения или при выгрузке страницы из физической памяти для загрузки в нее другой страницы.

Заметим, что операционная система Microsoft Windows NT активно работает с файлами, отображаемыми в память. В частности, при загрузке исполнимого модуля приложения соответствующий exe- или dll-файл отображается на память, а затем ему передается управление. Когда пользователь запускает вторую копию приложения, для работы используется файл, который уже отображается в память. В этом случае соответствующие страницы виртуальной памяти отображаются в адресные пространства обоих приложений. При этом возникает одна интересная проблема, связанная с использованием глобальных переменных.

Представьте себе, что в приложении определены глобальные переменные. При запуске приложения область глобальных переменных загружается в память наряду с исполнимым кодом. Если запущены две копии приложения, то страницы памяти, отведенные для кода и глобальных данных, будут отображаться в адресные пространства двух разных процессов. При этом существует потенциальная возможность конфликта,

когда разные работающие копии одного и того же приложения попытаются установить разные значения для одних и тех же глобальных переменных.

Операционная система Microsoft Windows NT выходит из этой ситуации следующим способом. Если она обнаружит, что одна из копий приложения пытается изменить страницу, в которой хранятся глобальные переменные, она создает для нее еще одну копию страницы. Таким образом, между приложениями никогда не возникает интерференции.

Аналогичная методика используется и для страниц, содержащих программный код. Если приложение (например, отладчик) пытается изменить страницу, содержащую исполнимый код, операционная система Microsoft Windows NT создает еще одну копию страницы и отображает ее в адресное пространство соответствующего процесса. Подробнее об этом мы расскажем в разделе, посвященном использованию файлов, отображаемых на память, для передачи данных между различными процессами.

### Создание отображения файла

Рассмотрим процедуру создания отображения файла на память.

Прежде всего, приложение должно открыть файл при помощи функции `CreateFile`, известной вам из предыдущего тома “Библиотеки системного программиста”. Ниже мы привели прототип этой функции:

```
HANDLE CreateFile(
    LPCTSTR lpFileName,           // адрес строки имени файла
    DWORD   dwDesiredAccess,      // режим доступа
    DWORD   dwShareMode, // режим совместного использования файла
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // дескриптор
                                           // защиты
    DWORD   dwCreationDisposition, // параметры создания
    DWORD   dwFlagsAndAttributes,  // атрибуты файла
    HANDLE hTemplateFile); // идентификатор файла с атрибутами
```

Через параметр `lpFileName` вы, как обычно, должны передать этой функции адрес текстовой строки, содержащей путь к открываемому файлу.

С помощью параметра `dwDesiredAccess` следует указать нужный вам вид доступа. Если файл будет открыт только для чтения, в этом параметре необходимо указать флаг `GENERIC_READ`. Если вы собираетесь выполнять над файлом операции чтения и записи, следует указать логическую комбинацию флагов `GENERIC_READ` и `GENERIC_WRITE`. В том случае, когда будет указан только флаг `GENERIC_WRITE`, операция чтения из файла будет запрещена.

Не забудьте также про параметр `dwShareMode`. Если файл будет использоваться одновременно несколькими процессами, через этот параметр необходимо передать режимы совместного использования файла: `FILE_SHARE_READ` или `FILE_SHARE_WRITE`.

Остальные параметры этой функции мы уже описали в предыдущем томе.

В случае успешного завершения, функция `CreateFile` возвращает идентификатор открытого файла. При ошибке возвращается значение `INVALID_HANDLE_VALUE`. Здесь все как обычно, пока никакого отображения еще не выполняется.

Для того чтобы создать отображение файла, вы должны вызвать функцию `CreateFileMapping`, прототип которой приведен ниже:

```
HANDLE CreateFileMapping(
    HANDLE hFile,           // идентификатор отображаемого файла
    LPSECURITY_ATTRIBUTES lpFileMappingAttributes, // дескриптор
                                // защиты
    DWORD flProtect,        // защита для отображаемого файла
    DWORD dwMaximumSizeHigh, // размер файла (старшее слово)
    DWORD dwMaximumSizeLow,  // размер файла (младшее слово)
    LPCTSTR lpName);        // имя отображенного файла
```

Через параметр `hFile` этой функции нужно передать идентификатор файла, для которого будет выполняться отображение в память, или значение `0xFFFFFFFF`. В первом случае функция `CreateFileMapping` отобразит заданный файл в память, а во втором - создаст отображение с использованием файла виртуальной памяти. Как мы увидим позже, отображение с использованием файла виртуальной памяти удобно для организации передачи данных между процессами.

Обратим ваше внимание на одну потенциальную опасность, связанную с использованием в паре функций `CreateFile` и `CreateFileMapping`. Если функция `CreateFile` завершится с ошибкой и эта ошибка не будет обработана приложением, функция `CreateFileMapping` получит через параметр `hFile` значение `INVALID_HANDLE_VALUE`, численно равное `0xFFFFFFFF`. В этом случае она сделает совсем не то, что предполагал разработчик приложения: вместо того чтобы выполнить отображение файла в память, функция создаст отображение с использованием файла виртуальной памяти.

Параметр `lpFileMappingAttributes` задает адрес дескриптора защиты. В большинстве случаев для этого параметра вы можете указать значение `NULL`.

Теперь займемся параметром `flProtect`, задающим защиту для создаваемого отображения файла. Для этого параметра вы можете задать следующий набор значений, комбинируя их с дополнительными атрибутами, которые будут перечислены ниже:

Значение	Описание
PAGE_READONLY	К выделенной области памяти предоставляется доступ только для чтения. При создании или открывании файла необходимо указать флаг <code>GENERIC_READ</code>
PAGE_READWRITE	К выделенной области памяти предоставляется доступ для чтения и записи. При создании или

открывании файла необходимо указать флаги `GENERIC_READ` и `GENERIC_WRITE`

PAGE\_WRITECOPY

К выделенной области памяти предоставляется доступ для копирования при записи. При создании или открывании файла необходимо указать флаги `GENERIC_READ` и `GENERIC_WRITE`. Режим копирования при записи будет описан позже в главе, посвященной обмену данными между процессами

Эти значения можно комбинировать при помощи логической операции ИЛИ со следующими атрибутами:

Атрибут	Описание
SEC_COMMIT	Если указан этот атрибут, выполняется выделение физических страниц в памяти или в файле виртуальной памяти. Этот атрибут используется по умолчанию
SEC_IMAGE	Используется при отображении программного файла, содержащего исполнимый код. Этот атрибут несовместим с остальными перечисленными в этом списке атрибутами
SEC_NOCACHE	Отмена кэширования для всех страниц отображаемой области памяти. Должен использоваться вместе с атрибутами <code>SEC_RESERVE</code> или <code>SEC_COMMIT</code>
SEC_RESERVE	Если указан этот атрибут, вместо выделения выполняется резервирование страниц виртуальной памяти. Резервированные таким образом страницы можно будет получить в пользование при помощи функции <code>VirtualAlloc</code> . Атрибут <code>SEC_RESERVE</code> можно указывать только в том случае, если в качестве параметра <code>hFile</code> функции <code>CreateFileMapping</code> передается значение <code>0xFFFFFFFF</code>

С помощью параметров `dwMaximumSizeHigh` и `dwMaximumSizeLow` необходимо указать функции `CreateFileMapping` 64-разрядный размер файла. Параметр `dwMaximumSizeHigh` должен содержать старшее 32-разрядное слово размера, а параметр `dwMaximumSizeLow` - младшее 32-разрядное слово размера. Для небольших файлов, длина которых укладывается в 32 разряда, нужно указывать нулевое значение параметра `dwMaximumSizeHigh`.

Заметим, что вы можете указать нулевые значения и для параметра `dwMaximumSizeHigh`, и для параметра `dwMaximumSizeLow`. В этом случае предполагается, что размер файла изменяться не будет.



Через параметр `lpName` можно указать имя отображения, которое будет доступно всем работающим одновременно приложениям. Имя должно представлять собой текстовую строку, закрытую двоичным нулем и не содержащую символов “\”.

Если отображение будет использоваться только одним процессом, вы можете не задавать для него имя. В этом случае значение параметра `lpName` следует указать как `NULL`.

В случае успешного завершения функция `CreateFileMapping` возвращает идентификатор созданного отображения. При ошибке возвращается значение `NULL`.

Так как имя отображения глобально, возможно возникновение ситуации, когда процесс пытается создать отображение с уже существующим именем. В этом случае функция `CreateFileMapping` возвращает идентификатор существующего отображения. Такую ситуацию можно определить с помощью функции `GetLastError`, вызвав ее сразу после функции `CreateFileMapping`. Функция `GetLastError` при этом вернет значение `ERROR_ALREADY_EXISTS`.

Выполнение отображения файла в память

Итак, мы выполнили первые два шага, необходимые для работы с файлом, отображаемым на память, - открытие файла функцией `CreateFile` и создание отображения функцией `CreateFileMapping`. Теперь, получив от функции `CreateFileMapping` идентификатор объекта-отображения, мы должны выполнить само отображение, вызвав для этого функцию `MapViewOfFile` или `MapViewOfFileEx`. В результате заданный фрагмент отображенного файла будет доступен в адресном пространстве процесса.

Прототип функции `MapViewOfFile` приведен ниже:

```
LPVOID MapViewOfFile(
    HANDLE hFileMappingObject, // идентификатор отображения
    DWORD dwDesiredAccess,     // режим доступа
    DWORD dwFileOffsetHigh,    // смещение в файле (старшее слово)
    DWORD dwFileOffsetLow,     // смещение в файле (младшее слово)
    DWORD dwNumberOfBytesToMap) // количество отображаемых байт
```

Функция `MapViewOfFile` создает окно размером `dwNumberOfBytesToMap` байт, которое смещено относительно начала файла на количество байт, заданное параметрами `dwFileOffsetHigh` и `dwFileOffsetLow`. Если задать значение параметра `dwNumberOfBytesToMap` равное нулю, будет выполнено отображение всего файла.

Смещение нужно задавать таким образом, чтобы оно попадало на границу минимального пространства памяти, которое можно зарезервировать. Значение 64 Кбайта подходит в большинстве случаев.

Более точно гранулярность памяти можно определить при помощи функции `GetSystemInfo`. Этой функции в качестве единственного параметра необходимо передать указатель на структуру типа `SYSTEM_INFO`, определенную следующим образом:

```
typedef struct _SYSTEM_INFO
{
    union {
        DWORD dwOemId; // зарезервировано
        struct
        {
            WORD wProcessorArchitecture; // архитектура системы
            WORD wReserved; // зарезервировано
        };
    };
    DWORD dwPageSize; // размер страницы
    LPVOID lpMinimumApplicationAddress; // минимальный адрес,
    // доступный приложениям и библиотекам DLL
    LPVOID lpMaximumApplicationAddress; // максимальный адрес,
    // доступный приложениям и библиотекам DLL
    DWORD dwActiveProcessorMask; // маски процессоров
    DWORD dwNumberOfProcessors; // количество процессоров
    DWORD dwProcessorType; // тип процессора
    DWORD dwAllocationGranularity; // гранулярность памяти
    WORD wProcessorLevel; // уровень процессора
    WORD wProcessorRevision; // модификация процессора
} SYSTEM_INFO;
```

Функция заполнит поля этой структуры различной информацией о системе. В частности, в поле `dwAllocationGranularity` будет записан минимальный размер резервируемой области памяти.

Вернемся к описанию функции `MapViewOfFile`.

Параметр `dwDesiredAccess` определяет требуемый режим доступа к отображению, то есть режимы доступа для страниц виртуальной памяти, используемых для отображения. Для этого параметра вы можете указать одно из следующих значений:

Значение	Описание
FILE_MAP_WRITE	Доступ на запись и чтение. При создании отображения функции <code>CreateFileMapping</code> необходимо указать тип защиты <code>PAGE_READWRITE</code>
FILE_MAP_READ	Доступ только на чтение. При создании отображения необходимо указать тип защиты <code>PAGE_READWRITE</code> или <code>PAGE_READ</code>
FILE_MAP_ALL_ACCESS	Аналогично <code>FILE_MAP_WRITE</code>
FILE_MAP_COPY	Доступ для копирования при записи. При создании отображения необходимо указать атрибут <code>PAGE_WRITECOPY</code>

В случае успешного выполнения отображения функция `MapViewOfFile` возвращает адрес отображенной области памяти. При ошибке возвращается значение `NULL`.

При необходимости приложение может запросить отображение в заранее выделенную область адресного пространства. Для этого следует воспользоваться функцией `MapViewOfFileEx`:

```
LPVOID MapViewOfFileEx(
    HANDLE hFileMappingObject, // идентификатор отображения
    DWORD dwDesiredAccess,     // режим доступа
    DWORD dwFileOffsetHigh,    // смещение в файле (старшее слово)
    DWORD dwFileOffsetLow,     // смещение в файле (младшее слово)
    DWORD dwNumberOfBytesToMap, // количество отображаемых байт
    LPVOID lpBaseAddress);     // предполагаемый адрес
                                // для отображения файла
```

Эта функция аналогична только что рассмотренной функции `MapViewOfFile` за исключением того, что она имеет еще один параметр `lpBaseAddress` - предполагаемый адрес для выполнения отображения.

Выполнение отображения с использованием функции `MapViewOfFileEx` используется в тех случаях, когда с помощью файла, отображаемого на память, организуется общая область памяти, доступная нескольким работающим параллельно процессам. При этом вы можете сделать так, что начальный адрес этой области будет одним и тем же для любого процесса, работающего с данным отображением.

Заметим, что функция `MapViewOfFileEx` сама выполняет резервирование адресов, поэтому вы не должны передавать ей адрес области памяти, полученный от функции `VirtualAlloc`. Еще одно ограничение заключается в том, что адрес, указанный через параметр `lpBaseAddress`, должен находиться на границе гранулярности памяти.

Приложение может создавать несколько отображений для разных или одинаковых фрагментов одного и того же файла.

### Открытие отображения

Если несколько процессов используют совместно одно и то же отображение, первый процесс создает это отображение с помощью функции `CreateFileMapping`, указав имя отображения, а остальные должны открыть его, вызвав функцию `OpenFileMapping`:

```
HANDLE OpenFileMapping(
    DWORD dwDesiredAccess, // режим доступа
    BOOL bInheritHandle,   // флаг наследования
    LPCTSTR lpName);       // адрес имени отображения файла
```

Через параметр `lpName` этой функции следует передать имя открываемого отображения. Имя должно быть задано точно также, как при создании отображения функцией `CreateFileMapping`.

Параметр `dwDesiredAccess` определяет требуемый режим доступа к отображению и указывается точно также, как и для описанной выше функции `MapViewOfFile`.

Параметр `bInheritHandle` определяет возможность наследования идентификатора отображения. Если он равен `TRUE`, порожденные процессы могут наследовать идентификатор, если `FALSE` - то нет.

### Отмена отображения файла

Если созданное отображение больше не нужно, его следует отменить с помощью функции `UnmapViewOfFile`:

```
BOOL UnmapViewOfFile(LPVOID lpBaseAddress);
```

Через единственный параметр этой функции необходимо передать адрес области отображения, полученный от функций `MapViewOfFile` или `MapViewOfFileEx`.

В случае успеха функция возвращает значение `TRUE`. При этом гарантируется, что все измененные страницы оперативной памяти, расположенные в отменяемой области отображения, будут записаны на диск в отображаемый файл. При ошибке функция возвращает значение `FALSE`.

Если приложение создало несколько отображений для файла, перед завершением работы с файлом все они должны быть отменены с помощью функции `UnmapViewOfFile`. Далее с помощью функции `CloseHandle` следует закрыть идентификаторы отображения, полученный от функции `CreateFileMapping` и `CreateFile`.

### Принудительная запись измененных данных

Как мы только что сказали, после отмены отображения все измененные страницы памяти записываются в отображаемый файл. Если это потребуется, приложение может в любое время выполнить принудительную запись измененных страниц в файл при помощи функции `FlushViewOfFile`:

```
BOOL FlushViewOfFile(
    LPCVOID lpBaseAddr, // начальный адрес сохраняемой области
    DWORD dwNumberOfBytesToFlush); // размер области в байтах
```

С помощью параметров `lpBaseAddr` и `dwNumberOfBytesToFlush` вы можете выбрать любой фрагмент внутри области отображения, для которого будет выполняться сохранение измененных страниц на диске. Если задать значение параметра `dwNumberOfBytesToFlush` равным нулю, будут сохранены все измененные страницы, принадлежащие области отображения.

### Приложение Oem2Char

Приложение `Oem2Char`, исходные тексты которого мы приведем в этом разделе, выполняет перекодировку текстовых файлов из формата OEM

(используется программами MS-DOS) в формат ANSI (используется приложениями Microsoft Windows) и обратно.

Вы можете оттранслировать исходные тексты этого приложения таким образом, чтобы оно использовало для работы с файлами один из трех методов: синхронный или асинхронный ввод/вывод, а также отображение файлов на память.

Главное окно приложения Oem2Char, запущенного в среде Microsoft Windows 95, показано на рис. 1.2. Точно такой же вид это окно будет иметь при запуске этого приложения в среде Microsoft Windows NT версии 4.0.

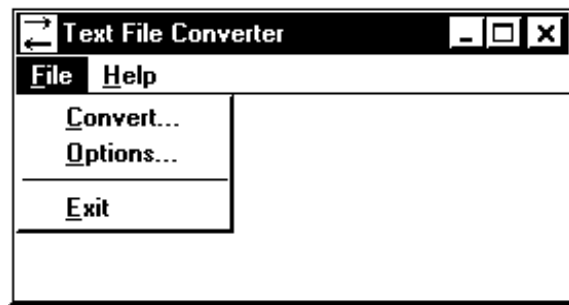


Рис. 1.2. Главное окно приложения Oem2Char

Выбирая из меню File строку Convert вы можете указать режим перекодировки: из OEM в ANSI или обратно, из ANSI в OEM. Соответствующая диалоговая панель, предназначенная для указания режима перекодировки, называется Conversion Options и показана на рис. 1.3.

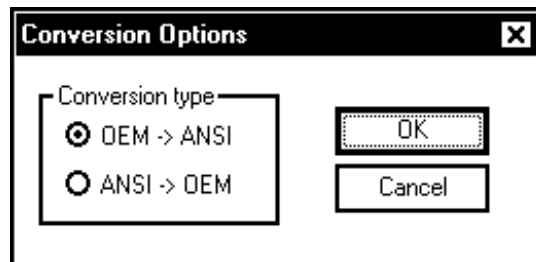


Рис. 1.3. Диалоговая панель Conversion Options панель, предназначенная для выбора режима перекодировки

По умолчанию приложение выполняет перекодировку текстовых файлов из формата OEM в формат ANSI.

Если исходные тексты приложения оттранслированы таким образом, что для работы с файлами используется синхронный либо асинхронный метод, вам предоставляется возможность выбрать исходный файл, который будет перекодироваться, и файл, в который будет записан результат перекодировки. При использовании отображения файла в память перекодировка выполняется "по месту".

Выбрав из меню File строку Convert, вы увидите на экране диалоговую панель Select source file, показанную на рис. 1.4.

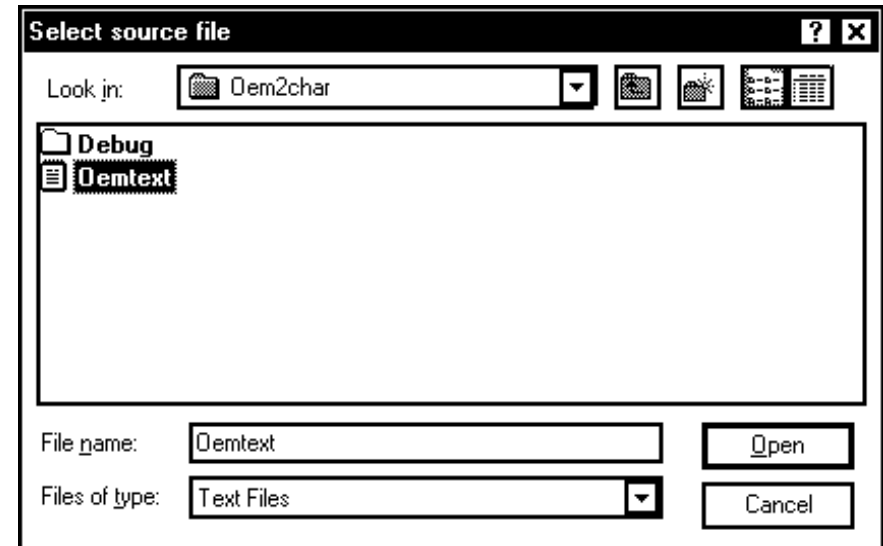


Рис. 1.4. Диалоговая панель Select source file, предназначенная для выбора исходного файла

С помощью этой диалоговой панели вы можете выбрать исходный файл для перекодировки. Напомним, что при работе с файлами в режиме отображения на память результат перекодировки будет записан в исходный файл.

Если же приложение работает с файлами в синхронном или асинхронном режиме, после выбора исходного файла вам будет представлена возможность выбрать файл для записи результата перекодировки (рис. 1.5). Различные режимы работы с файлами были описаны нами в предыдущем томе "Библиотеки системного программиста".

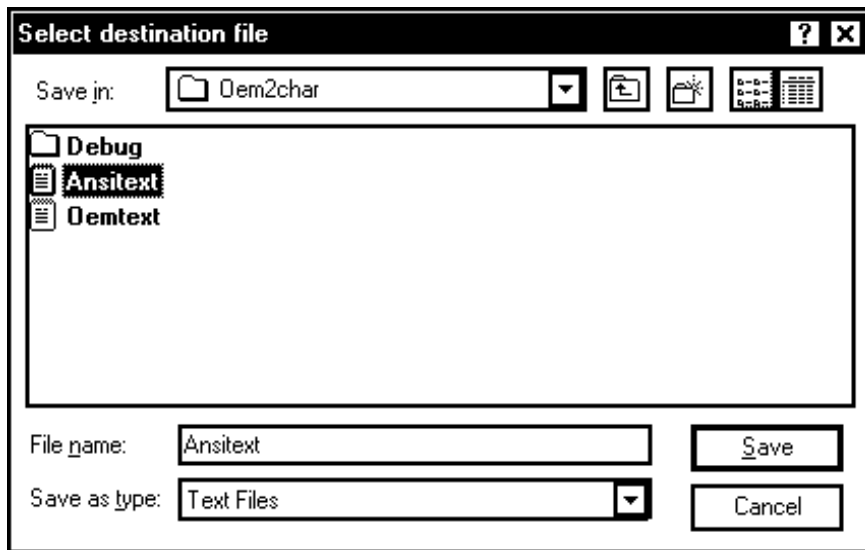


Рис. 1.5. Диалоговая панель *Select destination file*, с помощью которой можно выбрать файл для записи результата перекодировки

### Исходные тексты приложения

Главный файл исходных текстов приложения Oem2Char приведены в листинге 1.1.

Листинг 1.1. Файл oem2char/oem2char.c

```
// =====
// Приложение OEM2CHAR
// Демонстрация использования файловых операций
// для перекодировки текстовых файлов
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====

#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "resource.h"
#include "afxres.h"

// Различные режимы работы с файлами
```

```
#define SYNCHRONOUS_IO 1 // синхронные операции
#define ASYNCHRONOUS_IO 2 // асинхронные операции
#define MEMORYMAPPED_IO 3 // отображение на память
```

```
// Для использования различных режимов работы
// с файлами используйте только одну из
// приведенных ниже трех строк
```

```
//#define FILEOP SYNCHRONOUS_IO
//#define FILEOP ASYNCHRONOUS_IO
#define FILEOP MEMORYMAPPED_IO
```

```
#include "oem2char.h"
```

```
HINSTANCE hInst;
char szAppName[] = "Oem2CharApp";
char szAppTitle[] = "Text File Converter";
```

```
// Тип преобразования:
// OEM -> ANSI или ANSI -> OEM
BOOL fConversionType = OEM_TO_ANSI;
```

```
// Идентификатор файла, который будет
// перекодирован в соответствии с содержимым
// глобальной переменной fConversionType
HANDLE hSrcFile;
```

```
// Эти определения используются для всех способов
// работы с файлами, кроме отображения на память
#if FILEOP != MEMORYMAPPED_IO
```

```
// Идентификатор файла, в который будет записан
// результат перекодировки
HANDLE hDstFile;
```

```
// Буфер для работы с файлами
CHAR cBuf[2048];
```

```
#endif
```

```
// -----
// Функция WinMain
// -----

int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hWnd;
    MSG msg;
```

```

// Сохраняем идентификатор приложения
hInst = hInstance;

// Проверяем, не было ли это приложение запущено ранее
hWnd = FindWindow(szAppName, NULL);
if(hWnd)
{
    // Если было, выдвигаем окно приложения на
    // передний план
    if(IsIconic(hWnd))
        ShowWindow(hWnd, SW_RESTORE);
    SetForegroundWindow(hWnd);
    return FALSE;
}

// Регистрируем класс окна
memset(&wc, 0, sizeof(wc));
wc.cbSize = sizeof(WNDCLASSEX);
wc.hIconSm = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICONSM),
    IMAGE_ICON, 16, 16, 0);
wc.style = 0;
wc.lpfnWndProc = (WNDPROC)WndProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInst;
wc.hIcon = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICON),
    IMAGE_ICON, 32, 32, 0);
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
wc.lpszClassName = szAppName;
if(!RegisterClassEx(&wc))
    if(!RegisterClass((LPWNDCLASS)&wc.style))
        return FALSE;

// Создаем главное окно приложения
hWnd = CreateWindow(szAppName, szAppTitle,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInst, NULL);
if(!hWnd) return (FALSE);

// Отображаем окно и запускаем цикл
// обработки сообщений
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

```

```

    }
    return msg.wParam;
}

// -----
// Функция WndProc
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
    LPARAM lParam)
{
    switch(msg)
    {
        HANDLE_MSG(hWnd, WM_COMMAND, WndProc_OnCommand);
        HANDLE_MSG(hWnd, WM_DESTROY, WndProc_OnDestroy);

        default:
            return(DefWindowProc(hWnd, msg, wParam, lParam));
    }
}

// -----
// Функция WndProc_OnDestroy
// -----
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{
    PostQuitMessage(0);
    return 0L;
}

// -----
// Функция WndProc_OnCommand
// -----
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id,
    HWND hwndCtl, UINT codeNotify)
{
    switch (id)
    {
        {
            // Выполняем преобразование файла
            case ID_FILE_CONVERT:
            {
                // Если не удалось открыть файлы, выводим
                // сообщение об ошибке
                if(!StartConversion(hWnd))
                    MessageBox(hWnd,
                        "Conversion Error\n"
                        "Unable to open file(s)",
                        szAppTitle, MB_OK | MB_ICONEXCLAMATION);

                break;
            }
        }
    }
}

```

```

    }

    case ID_FILE_OPTIONS:
    {
        // Отображаем диалоговую панель, предназначенную
        // для настройки параметров преобразования
        DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1),
            hWnd, DlgProc);
        break;
    }

    case ID_FILE_EXIT:
    {
        // Завершаем работу приложения
        PostQuitMessage(0);
        return 0L;
        break;
    }

    case ID_HELP_ABOUT:
    {
        MessageBox(hWnd,
            "Text File Converter\n"
            "(C) Alexandr Frolov, 1996\n"
            "Email: frolov@glas.apc.org",
            szAppTitle, MB_OK | MB_ICONINFORMATION);
        return 0L;
        break;
    }

    default:
        break;
}

return FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify,
    DefWindowProc);
}

// -----
// Функция DlgProc
// -----
LRESULT WINAPI
DlgProc(HWND hdlg, UINT msg, WPARAM wParam,
    LPARAM lParam)
{
    switch(msg)
    {
        HANDLE_MSG(hdlg, WM_INITDIALOG, DlgProc_OnInitDialog);
        HANDLE_MSG(hdlg, WM_COMMAND, DlgProc_OnCommand);

        default:
            return FALSE;
    }
}

```

```

    }

    // -----
    // Функция DlgProc_OnInitDialog
    // -----
    BOOL DlgProc_OnInitDialog(HWND hdlg, HWND hwndFocus,
        LPARAM lParam)
    {
        // При инициализации диалоговой панели включаем
        // переключатель "OEM -> ANSI"
        CheckDlgButton(hdlg, IDC_OEMANSI, 1);
        return TRUE;
    }

    // -----
    // Функция DlgProc_OnCommand
    // -----
    #pragma warning(disable: 4098)
    void DlgProc_OnCommand(HWND hdlg, int id,
        HWND hwndCtl, UINT codeNotify)
    {
        switch (id)
        {
            // Определяем и сохраняем состояние переключателей
            case IDOK:
            {
                if(IsDlgButtonChecked(hdlg, IDC_OEMANSI))
                {
                    // Включен режим преобразования из кодировки
                    // OEM в кодировку ANSI
                    fConversionType = OEM_TO_ANSI;
                }
                else if(IsDlgButtonChecked(hdlg, IDC_ANSIOEM))
                {
                    // Включен режим преобразования из кодировки
                    // ANSI в кодировку OEM
                    fConversionType = ANSI_TO_OEM;
                }

                EndDialog(hdlg, 0);
                return TRUE;
            }

            // Если пользователь нажимает кнопку Cancel,
            // завершаем работу диалоговой панели без
            // изменения режима перекодировки
            case IDCANCEL:
            {
                EndDialog(hdlg, 0);
                return TRUE;
            }

            default:

```

```

        break;
    }
    return FALSE;
}

// -----
// Функция StartConversion
// -----
BOOL StartConversion(HWND hwnd)
{
    OPENFILENAME ofn;
    char szFile[256];
    char szDirName[256];
    char szFileTitle[256];
    char szFilter[256] =
        "Text Files\0*.txt\0Any Files\0*.*\0";
    char szDlgTitle[] = "Select source file";
    char szDlgTitleSave[] = "Select destination file";

    // Подготавливаем структуру для выбора исходного файла

    memset(&ofn, 0, sizeof(OPENFILENAME));
    GetCurrentDirectory(sizeof(szDirName), szDirName);
    szFile[0] = '\0';

    ofn.lStructSize = sizeof(OPENFILENAME);
    ofn.hwndOwner = hwnd;
    ofn.lpstrFilter = szFilter;
    ofn.lpstrInitialDir = szDirName;
    ofn.nFilterIndex = 1;
    ofn.lpstrFile = szFile;
    ofn.nMaxFile = sizeof(szFile);
    ofn.lpstrFileTitle = szFileTitle;
    ofn.nMaxFileTitle = sizeof(szFileTitle);
    ofn.lpstrTitle = szDlgTitle;
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST
        | OFN_HIDEREADONLY;

    // Выводим на экран диалоговую панель, предназначенную
    // для выбора исходного файла
    if(GetOpenFileName(&ofn))
    {
        // Если файл выбран, открываем его
        if (*ofn.lpstrFile)
        {
            if FILEOP == SYNCHRONOUS_IO
            {
                // В синхронном режиме указываем флаг
                // FILE_FLAG_SEQUENTIAL_SCAN
                hSrcFile = CreateFile(ofn.lpstrFile, GENERIC_READ,
                    FILE_SHARE_READ, NULL, OPEN_EXISTING,

```

```

                FILE_FLAG_SEQUENTIAL_SCAN, NULL);
            #elif FILEOP == ASYNCHRONOUS_IO
            {
                // В асинхронном режиме необходимо указывать
                // флаг FILE_FLAG_OVERLAPPED
                hSrcFile = CreateFile(ofn.lpstrFile, GENERIC_READ,
                    FILE_SHARE_READ, NULL, OPEN_EXISTING,
                    FILE_FLAG_OVERLAPPED, NULL);
            }
            #elif FILEOP == MEMORYMAPPED_IO
            {
                // В режиме отображения на память мы не используем
                // дополнительные флаги, однако указываем, что
                // файл будет открываться не только для чтения,
                // но и для записи
                hSrcFile = CreateFile(ofn.lpstrFile,
                    GENERIC_READ | GENERIC_WRITE,
                    0, NULL, OPEN_EXISTING, 0, NULL);
            }
        }
        else
        {
            return FALSE;
        }
        else
        {
            return FALSE;
        }

        // Если исходный файл открыть не удалось,
        // возвращаем значение FALSE
        if(hSrcFile == INVALID_HANDLE_VALUE)
            return FALSE;

        #if FILEOP == MEMORYMAPPED_IO
        {
            // В том случае, когда используется отображение
            // файла на память, этот файл перекодируется
            // "по месту". Для этого используется функция
            // Oem2Char с одним параметром.

            Oem2Char(hSrcFile);
            CloseHandle(hSrcFile);
        }
        #elif FILEOP == SYNCHRONOUS_IO || FILEOP == ASYNCHRONOUS_IO
        {
            // При использовании синхронного или асинхронного
            // режима работы с файлами результат перекодировки
            // будет записан в новый файл. Поэтому в этом
            // случае мы открываем второй файл и используем
            // другой вариант функции Oem2Char - с двумя
            // параметрами

```

```

ofn.lpstrTitle = szDlgTitleSave;
ofn.Flags      = OFN_HIDEREADONLY;

if(GetSaveFileName(&ofn))
{
    // Если файл выбран, открываем его
    if (*ofn.lpstrFile)
    {
#if FILEOP == SYNCHRONOUS_IO

        // При использовании синхронных операций указываем
        // флаг FILE_FLAG_SEQUENTIAL_SCAN
        hDstFile = CreateFile(ofn.lpstrFile, GENERIC_WRITE,
                               0, NULL, CREATE_ALWAYS,
                               FILE_FLAG_SEQUENTIAL_SCAN, NULL);

#elif FILEOP == ASYNCHRONOUS_IO

        // При использовании асинхронных операций
        // необходимо указать флаг FILE_FLAG_OVERLAPPED
        hDstFile = CreateFile(ofn.lpstrFile, GENERIC_WRITE,
                               0, NULL, CREATE_ALWAYS,
                               FILE_FLAG_OVERLAPPED, NULL);

#endif

        }
        else
            return FALSE;
    }
    else
        return FALSE;

    // Если выходной файл открыть не удалось,
    // возвращаем значение FALSE
    if(hDstFile == INVALID_HANDLE_VALUE)
        return FALSE;

    // Выполняем перекодировку файла hSrcFile с записью
    // результата перекодировки в файл hDstFile
    Oem2Char(hSrcFile, hDstFile);

    // После перекодировки закрываем оба файла
    CloseHandle(hSrcFile);
    CloseHandle(hDstFile);

#endif

    // В случае успеха возвращаем значение TRUE
    return TRUE;

```

```

}

// -----
// Функция Oem2Char
// -----
// Синхронный вариант функции
// -----

#if FILEOP == SYNCHRONOUS_IO

void Oem2Char(HANDLE hSrcFile, HANDLE hDstFile)
{
    DWORD dwBytesRead;
    DWORD dwBytesWritten;
    BOOL bResult;

    // Выполняем перекодировку файла в цикле
    while(TRUE)
    {
        // Читаем блок данных из исходного файла
        // в буфер cBuf
        bResult = ReadFile(hSrcFile, cBuf, 2048,
                           &dwBytesRead, NULL);

        // Проверяем, не был ли достигнут конец файла
        if(bResult && dwBytesRead == 0)
            break;

        // Выполняем преобразование в соответствии с
        // содержимым глобальной переменной fConversionType
        if(fConversionType == OEM_TO_ANSI)

            // Преобразование из OEM в ANSI
            OemToCharBuff(cBuf, cBuf, dwBytesRead);

        else if(fConversionType == ANSI_TO_OEM)

            // Преобразование из ANSI в OEM
            CharToOemBuff(cBuf, cBuf, dwBytesRead);

        // Запись содержимого буфера в выходной файл
        WriteFile(hDstFile, cBuf, dwBytesRead,
                  &dwBytesWritten, NULL);
    }
}

// -----
// Функция Oem2Char
// -----
// Асинхронный вариант функции
// -----

```



```

#elif FILEOP == ASYNCHRONOUS_IO

void Oem2Char(HANDLE hSrcFile, HANDLE hDstFile)
{
    DWORD dwBytesRead;
    DWORD dwBytesWritten;
    BOOL bResult;
    DWORD dwError;

    // Структуры для выполнения асинхронной работы
    OVERLAPPED ovRead;
    OVERLAPPED ovWrite;

    // Инициализация структуры для асинхронного чтения
    ovRead.Offset = 0;
    ovRead.OffsetHigh = 0;
    ovRead.hEvent = NULL;

    // Инициализация структуры для асинхронной записи
    ovWrite.Offset = 0;
    ovWrite.OffsetHigh = 0;
    ovWrite.hEvent = NULL;

    // Выполняем перекодировку файла в цикле
    while(TRUE)
    {
        // Запускаем операцию асинхронного чтения
        bResult = ReadFile(hSrcFile, cBuf, sizeof(cBuf),
            &dwBytesRead, &ovRead);

        // Проверяем результат запуска
        if(!bResult)
        {
            // Если произошла ошибка, анализируем ее код
            switch (dwError = GetLastError())
            {
                // При достижении конца файла завершаем работу
                // цикла и возвращаемся из функции
                case ERROR_HANDLE_EOF:
                {
                    return;
                }

                // Операция чтения запущена и еще выполняется
                case ERROR_IO_PENDING:
                {
                    // Здесь вы можете разместить вызов функции,
                    // которая будет выполняться параллельно с
                    // только что запущенной операцией чтения
                    //
                    // IdleWork();
                }
            }
        }
    }
}

```

```

// Перед тем как перейти к перекодировке
// считанного из файла блока, необходимо
// дождаться завершения операции
WaitForSingleObject(hSrcFile, INFINITE);

// Получаем результат выполнения асинхронной
// операции чтения
bResult = GetOverlappedResult(hSrcFile, &ovRead,
    &dwBytesRead, FALSE);

if(!bResult)
{
    switch (dwError = GetLastError())
    {
        // При достижении конца файла завершаем работу
        // цикла и возвращаемся из функции
        case ERROR_HANDLE_EOF:
        {
            return;
        }
        default:
            break;
    }
}
default:
    break;
}

// Получаем результат выполнения асинхронной
// операции чтения
GetOverlappedResult(hSrcFile, &ovRead,
    &dwBytesRead, FALSE);

// Выполняем преобразование
if(fConversionType == OEM_TO_ANSI)
    OemToCharBuff(cBuf, cBuf, dwBytesRead);

else if(fConversionType == ANSI_TO_OEM)
    CharToOemBuff(cBuf, cBuf, dwBytesRead);

// Продвигаем указатель позиции, с которой
// начнется следующая операция асинхронного
// чтения на количество считанных байт
ovRead.Offset += dwBytesRead;

// Запускаем асинхронную операцию записи
bResult = WriteFile(hDstFile, cBuf, dwBytesRead,
    &dwBytesWritten, &ovWrite);

// Если произошла ошибка, анализируем ее код

```

```

if(!bResult)
{
    switch (dwError = GetLastError())
    {
        // Операция записи запущена и еще выполняется
        case ERROR_IO_PENDING:
        {
            // Здесь вы можете разместить вызов функции,
            // которая будет выполняться параллельно с
            // только что запущенной операцией чтения
            //
            // IdleWork();

            // Получаем результат выполнения асинхронной
            // операции записи
            GetOverlappedResult(hDstFile, &ovWrite,
                &dwBytesWritten, TRUE);

            if(!bResult)
            {
                switch (dwError = GetLastError())
                {
                    default:
                        break;
                }
            }
        }
        default:
            break;
    }
}

// Продвигаем указатель позиции, с которой
// начнется следующая операция асинхронной
// записи на количество записанных байт
ovWrite.Offset += dwBytesWritten;
}

// -----
// Функция Oem2Char
// -----
// Вариант функции с использованием отображения
// на память
// -----

#elif FILEOP == MEMORYMAPPED_IO

void Oem2Char(HANDLE hSrcFile)
{
    DWORD dwFileSize;
    HANDLE hFileMapping;

```

```

LPVOID lpFileMap;

// Получаем и сохраняем размер файла
dwFileSize = GetFileSize(hSrcFile, NULL);

// Создаем объект-отображение для исходного файла
hFileMapping = CreateFileMapping(hSrcFile,
    NULL, PAGE_READWRITE, 0, dwFileSize, NULL);

// Если создать не удалось, возвращаем управление
if(hFileMapping == NULL)
    return;

// Выполняем отображение файла на память.
// В переменную lpFileMap будет записан указатель на
// отображаемую область памяти
lpFileMap = MapViewOfFile(hFileMapping,
    FILE_MAP_WRITE, 0, 0, 0);

// Если выполнить отображение не удалось,
// возвращаем управление
if(lpFileMap == 0)
    return;

// Выполняем преобразование файла за один прием
if(fConversionType == OEM_TO_ANSI)
    OemToCharBuff(lpFileMap, lpFileMap, dwFileSize);

else if(fConversionType == ANSI_TO_OEM)
    CharToOemBuff(lpFileMap, lpFileMap, dwFileSize);

// Отменяем отображение файла
UnmapViewOfFile(lpFileMap);

// Освобождаем идентификатор созданного
// объекта-отображения
CloseHandle(hFileMapping);
}

#endif

```

В листинге 1.2 приведен файл oem2char.h, в котором находятся определения и прототипы функций.

Листинг 1.2. Файл oem2char/oem2char.h

```

// Режимы перекодировки
#define OEM_TO_ANSI 1
#define ANSI_TO_OEM 2

// -----
// Описание функций

```

```
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

void WndProc_OnCommand(HWND hWnd, int id,
    HWND hwndCtl, UINT codeNotify);

void WndProc_OnDestroy(HWND hWnd);

LRESULT WINAPI
DlgProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);

BOOL DlgProc_OnInitDialog(HWND hwnd, HWND hwndFocus,
    LPARAM lParam);

void DlgProc_OnCommand(HWND hdlg, int id,
    HWND hwndCtl, UINT codeNotify);

BOOL StartConversion(HWND hwnd);

// Выбираем разный прототип функции в зависимости
// от выбранного режима работы с файлами
#if FILEOP == MEMORYMAPPED_IO

    void Oem2Char(HANDLE hSrcFile);

#elif FILEOP == SYNCHRONOUS_IO

    void Oem2Char(HANDLE hSrcFile, HANDLE hDstFile);

#elif FILEOP == ASYNCHRONOUS_IO

    void Oem2Char(HANDLE hSrcFile, HANDLE hDstFile);

#endif
```

Файл resource.h (листинг 1.3), который создается автоматически, содержит определения для файла описания ресурсов приложения.

Листинг 1.3. Файл oem2char/resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by OEM2CHAR.RC
//
#define IDR_APPMENU                102
#define IDI_APPICON                103
#define IDI_APPICONSM              104
#define IDD_DIALOG1                105
#define IDC_OEMANSI                1004
#define IDC_ANSIOEM                1005
#define ID_FILE_EXIT                40001
```

```
#define ID_HELP_ABOUT                40002
#define ID_FILE_OPTIONS            40004
#define ID_FILE_CONVERT            40005
```

```
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        106
#define _APS_NEXT_COMMAND_VALUE        40006
#define _APS_NEXT_CONTROL_VALUE        1010
#define _APS_NEXT_SYMED_VALUE        101
#endif
#endif
```

В листинге 1.4 вы найдете файл oem2char.rc. Это файл описания ресурсов приложения.

Листинг 1.4. Файл oem2char/oem2char.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
```

```

END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

////////////////////////////////////
//
// Menu
//

IDR_APPMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Convert...",          ID_FILE_CONVERT
        MENUITEM "&Options...",          ID_FILE_OPTIONS
        MENUITEM SEPARATOR
        MENUITEM "&Exit",                ID_FILE_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About",                ID_HELP_ABOUT
    END
END

////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure
// application icon
// remains consistent on all systems.
IDI_APPICON          ICON          DISCARDABLE          "OEM2CHAR.ICO"
IDI_APPICONSM         ICON          DISCARDABLE          "OEM2CHSM.ICO"

////////////////////////////////////
//
// Dialog
//

```

```

IDD_DIALOG1 DIALOG DISCARDABLE  0, 0, 169, 59
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Conversion Options"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON      "OK",IDOK,103,14,50,14
    PUSHBUTTON         "Cancel",IDCANCEL,103,31,50,14
    GROUPBOX           "Conversion type",IDC_STATIC,7,7,78,42
    CONTROL             "OEM -> ANSI",
                        IDC_OEMANSI,"Button",BS_AUTORADIOBUTTON |
                        WS_GROUP,15,18,61,10
    CONTROL             "ANSI -> OEM",
                        IDC_ANSIOEM,"Button",
                        BS_AUTORADIOBUTTON,15, 32,61,10
END

////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 162
        TOPMARGIN, 7
        BOTTOMMARGIN, 52
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////
#endif    // not APSTUDIO_INVOKED

```

### Определения и глобальные переменные

Так как объем нашей книги ограничен, мы решили не приводить по одному примеру приложений на каждый метод работы с файлами (синхронный,

асинхронный, отображение файла на память), а совместить все в одном приложении.

Для этого мы сделали три определения, показанных ниже:

```
#define SYNCHRONOUS_IO 1 // синхронные операции
#define ASYNCHRONOUS_IO 2 // асинхронные операции
#define MEMORYMAPPED_IO 3 // отображение на память
```

Каждая из этих строк соответствует одному из режимов работы с файлами.

Для того чтобы исходные тексты были оттранслированы для использования нужного вам режима работы с файлами, необходимо определить значение макропеременной FILEOP. Это можно сделать одним из трех способов:

```
// #define FILEOP SYNCHRONOUS_IO
// #define FILEOP ASYNCHRONOUS_IO
#define FILEOP MEMORYMAPPED_IO
```

Перед трансляцией исходных текстов приложения вы должны выбрать одну из этих строк, закрыв две другие символами комментария. При этом с помощью операторов условной трансляции, расположенных в исходных текстах приложения, будет выбран нужный вариант исходного текста.

Строка включения файла определений oem2char.h должна располагаться после строки определения макропеременной FILEOP, так как в этом файле тоже имеются операторы условной трансляции.

В глобальной переменной fConversionType хранится текущий режим преобразования, который можно изменить при помощи диалоговой панели Conversion Options, показанной на рис. 1.3. Сразу после запуска приложения в этой переменной хранится значение OEM\_TO\_ANSI, в результате чего приложение будет выполнять перекодировку из OEM в ANSI. Функция диалога диалоговой панели Conversion Options может записать в переменную fConversionType значение ANSI\_TO\_OEM. В результате приложение будет перекодировать текстовые файлы из ANSI в OEM.

Далее в области глобальных переменных определены переменные для хранения идентификаторов файлов hSrcFile, hDstFile, а также буфер cBuf размером 2048 байт:

```
HANDLE hSrcFile;
#ifdef FILEOP != MEMORYMAPPED_IO
    HANDLE hDstFile;
    CHAR cBuf[2048];
#endif
```

В переменной hSrcFile хранится идентификатор исходного файла. Что же касается переменной hDstFile, то она предназначена для хранения идентификатора файла, в который будет записан результат перекодировки. Эта переменная, а также буфер для временного хранения перекодированных данных cBuf не нужны при работе с использованием отображения файла в память. Поэтому если значение макропеременной FILEOP не равно

константе MEMORYMAPPED\_IO, строки определения переменной hDstFile и буфера cBuf пропускаются при трансляции исходных текстов приложения.

## Описание функций

Приведем описание функций, определенных в нашем приложении.

### Функция WinMain

Функция WinMain не имеет никаких особенностей. Сразу после запуска приложения она сохраняет идентификатор приложения в глобальной переменной hInst и проверяет, нет ли в памяти копии приложения, запущенной раньше. Для такой проверки используется методика, основанная на вызове функции FindWindow и описанная в предыдущем томе “Библиотеки системного программиста”, посвященного программированию для операционной системы Microsoft Windows NT. Если будет найдена копия работающего приложения, его окно выдвигается на передний план при помощи функций ShowWindow и SetForegroundWindow.

Далее функция WinMain регистрирует класс главного окна приложения, создает и отображает это окно, а затем запускает цикл обработки сообщений.

### Функция WndProc

В задачу функции WndProc входит обработка сообщений, поступающих в главное окно приложения. Если от главного меню приложения приходит сообщение WM\_COMMAND, вызывается функция WndProc\_OnCommand. При уничтожении главного окна приложения в его функцию поступает сообщение WM\_DESTROY, для обработки которого вызывается функция WndProc\_OnDestroy.

Все остальные сообщения передаются функции DefWindowProc.

### Функция WndProc\_OnDestroy

Эта функция вызывается при уничтожении главного окна приложения для обработки сообщения WM\_DESTROY. Функция WndProc\_OnDestroy вызывает функцию PostQuitMessage, в результате чего цикл обработки сообщений завершает свою работу.

### Функция WndProc\_OnCommand

Функция WndProc\_OnCommand обрабатывает сообщение WM\_COMMAND, поступающее от главного меню приложения. Для обработки мы использовали макрокоманду HANDLE\_MSG, описанную в предыдущем томе “Библиотеки системного программиста”.

Если пользователь выберет из меню File строку Convert, будет выполняться преобразование файла. Для этого функция WndProc\_OnCommand вызовет функцию StartConversion, передав ей в

качестве единственного параметра идентификатор главного окна приложения.

При выборе из меню File строки Options приложение выведет на экран диалоговую панель Conversion Options, вызвав для этого функцию DialogBox:

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, DlgProc);
```

Диалоговая панель имеет идентификатор IDD\_DIALOG1 и определена в файле описания ресурсов приложения.

### Функция DlgProc

Функция диалога DlgProc обрабатывает сообщения WM\_INITDIALOG и WM\_COMMAND, поступающие от диалоговой панели Conversion Options. Для обработки этих сообщений вызываются, соответственно, функции DlgProc\_OnInitDialog и DlgProc\_OnCommand.

### Функция DlgProc\_OnInitDialog

Сообщение WM\_INITDIALOG поступает в функцию диалога при инициализации диалоговой панели. Функция DlgProc\_OnInitDialog, обрабатывая это сообщение, выполняет единственную задачу: она включает переключатель OEM -> ANSI, устанавливая таким образом режим перекодировки из OEM в ANSI:

```
CheckDlgButton(hDlg, IDC_OEMANSI, 1);
```

### Функция DlgProc\_OnCommand

Когда в диалоговой панели Conversion Options пользователь нажимает одну из кнопок или клавиши <Esc> и <Enter>, в функцию диалога поступает сообщение WM\_COMMAND. Обработчик этого сообщения, расположенный в функции DlgProc\_OnCommand, определяет текущее состояние переключателей режима перекодировки, расположенных на поверхности диалоговой панели, и записывает соответствующее значение в глобальную переменную fConversionType:

```
if(IsDlgButtonChecked(hDlg, IDC_OEMANSI))
{
    fConversionType = OEM_TO_ANSI;
}
else if(IsDlgButtonChecked(hDlg, IDC_ANSIOEM))
{
    fConversionType = ANSI_TO_OEM;
}
```

Если при работе с диалоговой панелью пользователь нажимает кнопку Cancel или клавишу <Esc>, содержимое глобальной переменной fConversionType не изменяется.

### Функция StartConversion

В задачу функции StartConversion входит выбор и открытие исходного файла и файла, в который будет записан результат перекодировки. Когда приложение работает с файлом в режиме отображения на память, открывается только один файл - исходный.

Для выбора файла мы использовали функцию GetOpenFileName, хорошо знакомую вам по предыдущим томам "Библиотеки системного программиста", посвященным программированию для операционной системы Microsoft Windows версии 3.1.

Выбранные файлы открываются при помощи функции CreateFile. Однако способ открывания зависит от режима работы с файлами. Ниже мы привели фрагмент исходного текста приложения, в котором открывается исходный файл:

```
#if FILEOP == SYNCHRONOUS_IO
    hSrcFile = CreateFile(ofn.lpstrFile, GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_FLAG_SEQUENTIAL_SCAN, NULL);

#elif FILEOP == ASYNCHRONOUS_IO
    hSrcFile = CreateFile(ofn.lpstrFile, GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_FLAG_OVERLAPPED, NULL);

#elif FILEOP == MEMORYMAPPED_IO
    hSrcFile = CreateFile(ofn.lpstrFile,
        GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_EXISTING, 0, NULL);
#endif
```

В синхронном режиме исходный файл будет читаться последовательно, поэтому мы указали режим GENERIC\_READ (только чтение) и флаг FILE\_FLAG\_SEQUENTIAL\_SCAN. Так как в процессе перекодировки исходный файл не будет изменяться, нет причин запрещать чтение этого файла для других процессов. Чтобы предоставить доступ другим процессам на чтение исходного файла, мы указали режим совместного использования файла FILE\_SHARE\_READ.

В асинхронном режиме необходимо указывать флаг FILE\_FLAG\_OVERLAPPED, что мы и сделали в нашем примере.

Что же касается режима отображения файла на память, то здесь при открывании файла мы указали режимы GENERIC\_READ и GENERIC\_WRITE. В результате файл открывается и для чтения, и для записи.

После того как в режиме отображения файла на память исходный файл будет открыт, функция StartConversion вызывает функцию Oem2Char, передавая ей в качестве единственного параметра идентификатор исходного файла:

```
Oem2Char(hSrcFile);
CloseHandle(hSrcFile);
```

Функция Oem2Char выполняет перекодировку файла “по месту”. Далее идентификатор исходного файла закрывается функцией CloseHandle, после чего функция StartConversion возвращает управление.

В синхронном и асинхронном режиме функция StartConversion после открывания исходного файла дополнительно открывает выходной файл, в который будет записан результат перекодировки. Для выбора выходного файла вызывается функция GetSaveFileName.

Так же как и исходный файл, выходной файл открывается при помощи функции CreateFile, причем в синхронном и асинхронном режиме этот файл открывается по-разному:

```
#if FILEOP == SYNCHRONOUS_IO
    hDstFile = CreateFile(ofn.lpszFile, GENERIC_WRITE,
        0, NULL, CREATE_ALWAYS,
        FILE_FLAG_SEQUENTIAL_SCAN, NULL);

#elif FILEOP == ASYNCHRONOUS_IO
    hDstFile = CreateFile(ofn.lpszFile, GENERIC_WRITE,
        0, NULL, CREATE_ALWAYS,
        FILE_FLAG_OVERLAPPED, NULL);
#endif
```

В синхронном режиме мы указываем режим доступа на запись GENERIC\_WRITE и флаг FILE\_FLAG\_SEQUENTIAL\_SCAN (так как запись в выходной файл будет выполняться последовательно от начала до конца).

В асинхронном режиме необходимо указать флаг FILE\_FLAG\_OVERLAPPED.

Кроме того, в обоих случаях мы указали режим открывания файла CREATE\_ALWAYS. В результате выходной файл будет создан заново даже в том случае, если в выбранном каталоге уже есть файл с таким именем. При этом содержимое старого файла будет уничтожено.

После открывания исходного и выходного файла вызывается функция Oem2Char, выполняющая перекодировку, а затем оба файла закрываются при помощи функции CloseHandle:

```
Oem2Char(hSrcFile, hDstFile);
CloseHandle(hSrcFile);
CloseHandle(hDstFile);
```

Обратите внимание, что в синхронном и асинхронном режиме работы с файлами в нашем приложении используется другой вариант функции Oem2Char - вариант с двумя параметрами. Как вы сейчас увидите, в нашем приложении используются три варианта этой функции.

### Функция Oem2Char (синхронные операции с файлами)

Если приложение подготовлено таким образом, что оно работает с файлами при помощи синхронных операций, используется следующий вариант исходного текста функции Oem2Char:

```
void Oem2Char(HANDLE hSrcFile, HANDLE hDstFile)
```

```
{
    DWORD dwBytesRead;
    DWORD dwBytesWritten;
    BOOL bResult;

    while(TRUE)
    {
        bResult = ReadFile(hSrcFile, cBuf, 2048,
            &dwBytesRead, NULL);
        if(bResult && dwBytesRead == 0)
            break;

        if(fConversionType == OEM_TO_ANSI)
            OemToCharBuff(cBuf, cBuf, dwBytesRead);
        else if(fConversionType == ANSI_TO_OEM)
            CharToOemBuff(cBuf, cBuf, dwBytesRead);

        WriteFile(hDstFile, cBuf, dwBytesRead,
            &dwBytesWritten, NULL);
    }
}
```

Здесь мы работаем с файлом обычным образом.

Перекодировка файла выполняется в цикле, который прерывается при достижении конца исходного файла.

Функция Oem2Char при помощи функции ReadFile читает фрагмент исходного файла в буфер cBuf, расположенный в области глобальных переменных. Количество прочитанных байт записывается при этом в локальную переменную dwBytesRead.

При достижении конца исходного файла количество байт, прочитанных функцией ReadFile из файла, а также значение, возвращенное этой функцией, равно нулю. Этот факт мы используем для завершения цикла перекодировки.

После прочтения блока данных из исходного файла функция Oem2Char анализирует содержимое глобальной переменной fConversionType, определяя тип преобразования, который нужно выполнить. В зависимости от содержимого этой переменной вызывается либо функция OemToCharBuff, выполняющая преобразование из кодировки OEM в кодировку ANSI, либо функция CharToOemBuff, которая выполняет обратное преобразование.

На следующем этапе преобразованное содержимое буфера cBuf записывается в выходной файл при помощи функции WriteFile. При этом количество действительно записанных байт сохраняется в локальной переменной dwBytesWritten, однако в нашем приложении оно никак не используется.

Сделаем важное замечание относительно функций OemToCharBuff и CharToOemBuff.

Для преобразования текстовых строк из кодировки OEM в кодировку ANSI и обратно в программном интерфейсе операционной системы Microsoft



Windows версии 3.1 имелся набор функций, описанный нами в 12 томе “Библиотеки системного программиста”, который называется “Операционная система Microsoft Windows 3.1 для программиста. Часть вторая”. Это такие функции как OemToAnsi, AnsiToOem, OemToAnsiBuff, AnsiToOemBuff.

В программном интерфейсе операционной системы Microsoft Windows NT эти функции оставлены для совместимости, однако ими не рекомендуется пользоваться. Причина заключается в том, что в Microsoft Windows NT можно работать с символами в кодировке Unicode, когда для представления каждого символа используется не один, а два байта.

Соответственно, вместо перечисленных выше функций необходимо использовать функции OemToChar, CharToOem, OemToCharBuff, CharToOemBuff, которые имеют такие же параметры, что и их 16-разрядные прототипы (за исключением того, что функциям OemToCharBuff и CharToOemBuff можно передавать 32-разрядную длину преобразуемой текстовой строки). В зависимости от того, используется ли приложением кодировка Unicode, эти функции могут преобразовывать строки из OEM в строки ANSI или Unicode (и обратно).

#### Функция Oem2Char (асинхронные операции с файлами)

Вариант функции Oem2Char, предназначенный для использования асинхронных операций с файлами, выглядит сложнее, однако он позволяет приложению выполнять дополнительную работу в то время когда происходит чтение или запись блока данных.

В асинхронном режиме для чтения и записи необходимо подготовить структуры типа OVERLAPPED. Мы делаем это следующим образом:

```
OVERLAPPED ovRead;
OVERLAPPED ovWrite;

ovRead.Offset = 0;
ovRead.OffsetHigh = 0;
ovRead.hEvent = NULL;

ovWrite.Offset = 0;
ovWrite.OffsetHigh = 0;
ovWrite.hEvent = NULL;
```

Структура ovRead используется для выполнения операций чтения. В поля OffsetHigh и Offset этой структуры мы записываем нулевые значения, поэтому чтение будет выполняться с самого начала файла.

Что же касается поля hEvent, то в него мы записываем значение NULL. При этом мы не будем создавать для синхронизации отдельный объект-событие, а воспользуемся идентификатором файла.

Структура ovWrite, которая предназначена для выполнения операций записи, используется аналогичным образом.

После подготовки структур ovRead и ovWrite функция Oem2Char начинает цикл перекодировки.

Прежде всего, в этом цикле запускается операция асинхронного чтения, для чего вызывается функция ReadFile:

```
bResult = ReadFile(hSrcFile, cBuf, sizeof(cBuf),
&dwBytesRead, &ovRead);
```

В качестве последнего параметра мы передаем этой функции адрес заранее подготовленной структуры ovRead.

Заметим, что в данном случае функция ReadFile вернет управление еще до завершения операции чтения, поэтому в переменную dwBytesRead не будет записано количество прочитанных байт (так как пока неизвестно, сколько их удастся прочитать).

Далее функция Oem2Char проверяет код возврата функции ReadFile. При этом дополнительно вызывается функция GetLastError.

Если при запуске процедуры чтения был достигнут конец файла, эта функция вернет значение ERROR\_HANDLE\_EOF. В этом случае функция Oem2Char просто возвращает управление.

Если же функция GetLastError вернула значение ERROR\_IO\_PENDING, то это означает, что в настоящий момент происходит выполнение операции чтения. Приложение может вызвать функцию, например, IdleWork, для выполнения какой-либо фоновой работы.

Перед тем как продолжить свою работу, функция дожидается завершения операции чтения, вызывая для этого функцию WaitForSingleObject, описанную в предыдущем томе “Библиотеки системного программиста”:

```
WaitForSingleObject(hSrcFile, INFINITE);
```

При этом главная задача приложения перейдет в состояние ожидания до тех пор, пока не закончится операция чтения. Напомним, что в состоянии ожидания задача не отнимает циклы процессорного времени и не снижает производительность системы.

Далее функция проверяет результат выполнения асинхронной операции чтения, вызывая функцию GetOverlappedResult:

```
bResult = GetOverlappedResult(hSrcFile, &ovRead,
&dwBytesRead, FALSE);
```

Помимо всего прочего, эта функция записывает в локальную переменную dwBytesRead количество байт, прочитанных из исходного файла.

После дополнительных проверок ошибок функция Oem2Char выполняет преобразование содержимого буфера, заполненного прочитанными данными.

Вслед за этим в структуре ovRead изменяется содержимое поля Offset:

```
ovRead.Offset += dwBytesRead;
```

Значение, которое находится в этом поле, увеличивается на количество прочитанных байт. В результате при очередном вызове функции ReadFile будет запущено чтение для следующего блока данных. Так как мы не изменяем поле OffsetHigh, наше приложение способно работать с файлами, имеющими размер не более 4 Гбайт (что, однако, вполне достаточно).



Асинхронная операция записи прочитанных данных запускается при помощи функции WriteFile:

```
bResult = WriteFile(hDstFile, cBuf, dwBytesRead,
    &dwBytesWritten, &ovWrite);
```

В качестве последнего параметра этой функции передается адрес заранее подготовленной структуры ovWrite.

После анализа кода возврата функции WriteFile вызывается функция GetOverlappedResult, с помощью которой определяется результат завершения операции записи:

```
GetOverlappedResult(hDstFile, &ovWrite,
    &dwBytesWritten, TRUE);
```

Так как через последний параметр этой функции передается значение TRUE, функция GetOverlappedResult выполняет ожидание завершения операции записи. Кроме того, в локальную переменную dwBytesWritten эта функция заносит количество байт данных, записанных в выходной файл.

Так как асинхронные операции чтения и записи не изменяют текущую позицию в файле, после выполнения записи мы изменяем соответствующим образом содержимое поля Offset в структуре ovWrite:

```
ovWrite.Offset += dwBytesWritten;
```

Далее цикл перекодировки продолжает свою работу.

### Функция Oem2Char (отображение файла на память)

Третий вариант функции Oem2Char, который используется для работы через отображение файла на память, выглядит очень просто. В нем даже нет цикла, а перекодировка выполняется за один прием:

```
void Oem2Char(HANDLE hSrcFile)
{
    DWORD dwFileSize;
    HANDLE hFileMapping;
    LPVOID lpFileMap;

    dwFileSize = GetFileSize(hSrcFile, NULL);

    hFileMapping = CreateFileMapping(hSrcFile,
        NULL, PAGE_READWRITE, 0, dwFileSize, NULL);
    if(hFileMapping == NULL)
        return;

    lpFileMap = MapViewOfFile(hFileMapping,
        FILE_MAP_WRITE, 0, 0, 0);
    if(lpFileMap == 0)
        return;

    if(fConversionType == OEM_TO_ANSI)
        OemToCharBuff(lpFileMap, lpFileMap, dwFileSize);
    else if(fConversionType == ANSI_TO_OEM)
        CharToOemBuff(lpFileMap, lpFileMap, dwFileSize);
}
```

```
UnmapViewOfFile(lpFileMap);
CloseHandle(hFileMapping);
}
```

Вначале функция Oem2Char определяет размер исходного файла, идентификатор которого передается ей через параметр hSrcFile. Для определения размера файла используется функция GetFileSize.

На следующем шаге с помощью функции CreateFileMapping создается объект-отображение. Страницы этого объекта будут доступны и на чтение, и на запись, так как указан режим PAGE\_READWRITE.

Для того чтобы получить доступ к памяти, отображенной на файл, наше приложение вызывает функцию MapViewOfFile. Отображение выполняется для чтения и записи, поэтому мы указали флаг FILE\_MAP\_WRITE. В случае успешного завершения функции адрес отображенной области записывается в локальную переменную lpFileMap.

Что же касается перекодировки файла, то она выполняется исключительно просто: мы передаем через первые два параметра функции перекодировки адрес, на который отображен файл, а через третий параметр - размер файла.

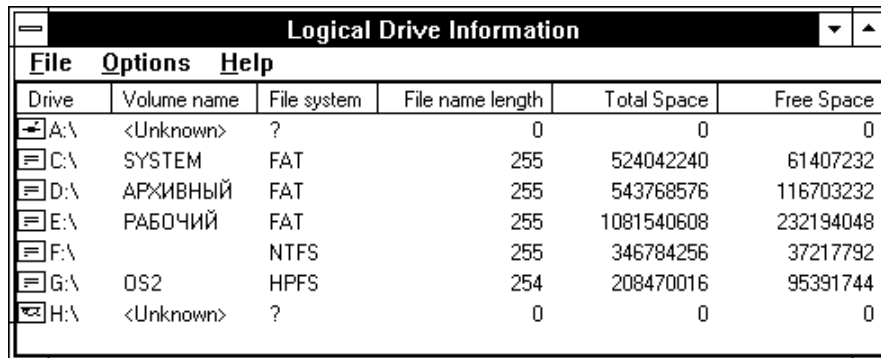
После того как перекодировка будет выполнена, необходимо вначале отменить отображение, а затем освободить идентификатор объекта-отображения. Первая задача решается в нашем приложении спомощью функции UnmapViewOfFile, а вторая - с помощью функции CloseHandle, которой в качестве единственного параметра передается идентификатор объекта-отображения.

## Приложение DiskInfo

Если ваше приложение просто открывает файлы или создает новые, возможно, для выбора файлов вам будет достаточно использовать стандартные диалоговые панели, которые мы создавали в предыдущем приложении. Однако во многих случаях вам необходимо предоставить пользователю детальную информацию о дисковых устройствах, такую, например, как тип файловой системы, общий объем диска, размер свободного пространства на диске, а также тип диска (локальный, сетевой, со сменным или несменным носителем данных, устройство чтения CD-ROM и так далее).

В этой главе мы приведем исходные тексты приложения DiskInfo, которое получает и отображает подробную информацию о всех дисковых устройствах, имеющихся в системе, как локальных, так и удаленных (сетевых). Информация отображается в табличном виде с помощью органа управления List View, который мы подробно описали в 22 томе "Библиотеки системного программиста", посвященному операционной системе Microsoft Windows 95.

Внешний вид главного окна приложения DiskInfo, запущенного в одном из режимов отображения, показан на рис. 1.6.



Drive	Volume name	File system	File name length	Total Space	Free Space
A:\	<Unknown>	?	0	0	0
C:\	SYSTEM	FAT	255	524042240	61407232
D:\	АРХИВНЫЙ	FAT	255	543768576	116703232
E:\	РАБОЧИЙ	FAT	255	1081540608	232194048
F:\		NTFS	255	346784256	37217792
G:\	OS2	HPFS	254	208470016	95391744
H:\	<Unknown>	?	0	0	0

Рис. 1.6. Просмотр информации о дисках в табличном виде

В столбце Drive отображаются пиктограммы и названия дисковых устройств, имеющихся в системе. Для каждого типа устройства используется своя пиктограмма.

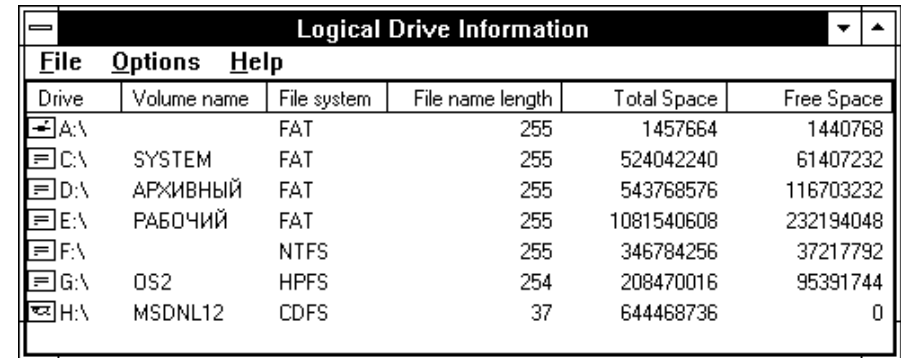
В столбце Volume name для каждого устройства располагается метка тома (если она есть). В столбце File system мы отображаем имя файловой системы.

Так как максимальная длина имени файлов и каталогов разная для различных файловых систем, то в столбце File name length мы отображаем эту длину для каждого дискового устройства.

В столбцах Total Space и Free Space выводится, соответственно, емкость диска в байтах и размер свободного пространства на диске (также в байтах).

Заметим, что при первом запуске приложение DiskInfo не пытается определить параметры для устройств со сменными носителями данных (устройства НГМД, устройства чтения CD-ROM, магнитооптические накопители и так далее). Это связано с тем, что при попытке определить параметры устройства операционная система Microsoft Windows NT выполняет обращение к соответствующему накопителю. В том случае, если в накопителе нет носителя данных, на экране появится предупреждающее сообщение.

Тем не менее, наше приложение может определить параметры устройств со сменными носителями данных. Для этого вы должны вставить носитель (дискету, компакт-диск и так далее) в устройство, а затем сделать двойной щелчок левой клавишей мыши по пиктограмме устройства. После этого соответствующая строка таблицы будет заполнена, как это показано на рис. 1.7.



Drive	Volume name	File system	File name length	Total Space	Free Space
A:\		FAT	255	1457664	1440768
C:\	SYSTEM	FAT	255	524042240	61407232
D:\	АРХИВНЫЙ	FAT	255	543768576	116703232
E:\	РАБОЧИЙ	FAT	255	1081540608	232194048
F:\		NTFS	255	346784256	37217792
G:\	OS2	HPFS	254	208470016	95391744
H:\	MSDNL12	CDFS	37	644468736	0

Рис. 1.7. Полностью заполненная таблица параметров дисковых устройств

Кроме того, после двойного щелчка по пиктограмме любого дискового устройства на экране появляется диалоговая панель Logical Drive Information, в которой отображается имя устройства, имя файловой системы, серийный номер, а также системные флаги (рис. 1.8 - 1.11). Системные флаги мы описали в последней главе предыдущего тома "Библиотеки системного программиста".

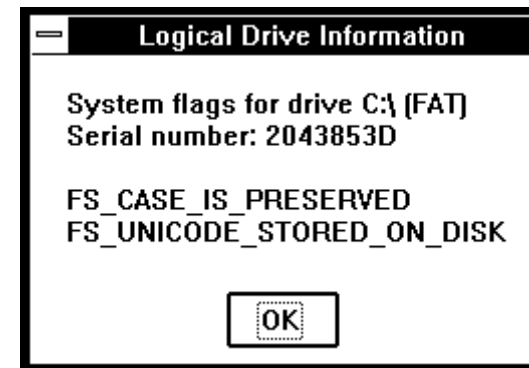


Рис. 1.8. Просмотр дополнительной информации о диске FAT

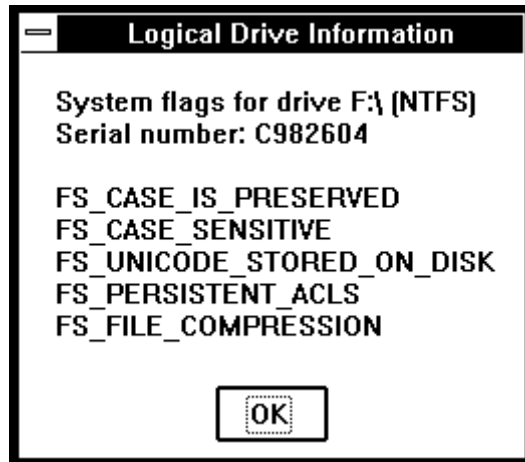


Рис. 1.9. Просмотр дополнительной информации о диске NTFS

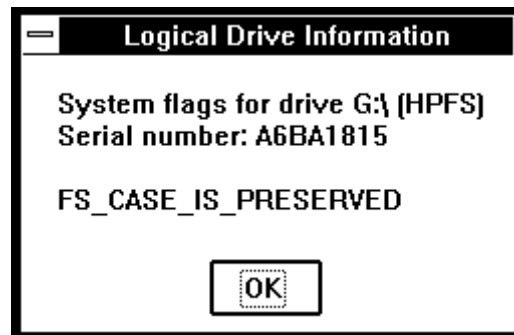


Рис. 1.10. Просмотр дополнительной информации о диске HPFS

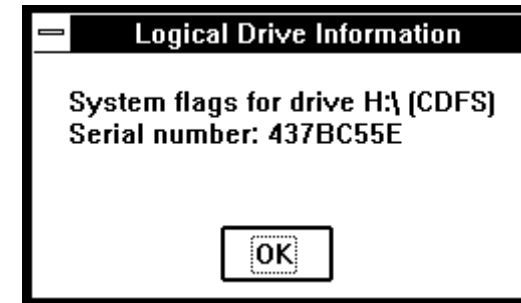


Рис. 1.11. Просмотр дополнительной информации о диске CDFS

С помощью меню Options, показанного на рис. 1.12, вы можете изменить режим отображения списка дисковых устройств.

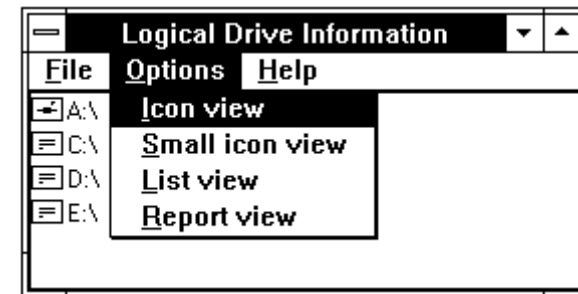


Рис. 1.12. Меню Options, предназначенное для изменения режима отображения списка дисковых устройств

Если выбрать из меню Options строку Icon view, внешний вид главного окна приложения изменится. Этот список будет отображаться в виде набора пиктограмм стандартного размера с подписью (рис. 1.13).

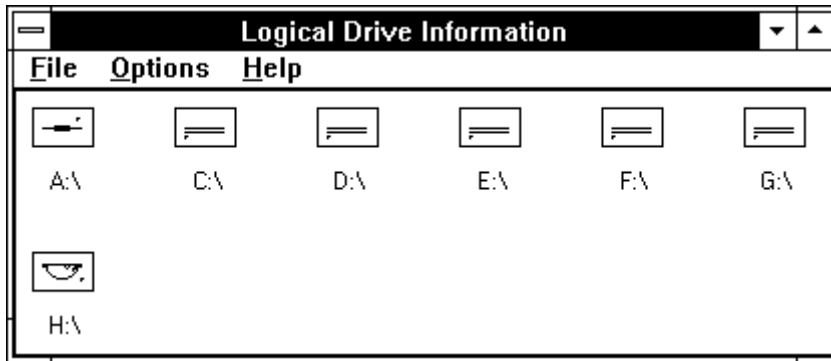


Рис. 1.13. Просмотр дисковых устройств в виде пиктограмм стандартного размера

Если же из этого меню выбрать строку Small icon view, для отображения списка устройств будут использованы маленькие пиктограммы (рис. 1.14).

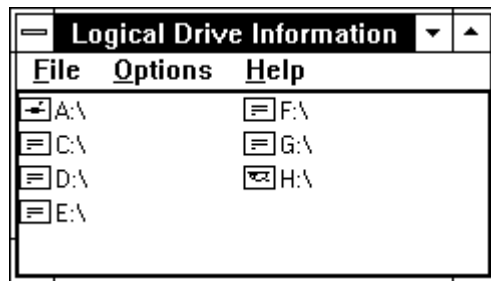


Рис. 1.14. Просмотр дисковых устройств в виде пиктограмм маленького размера

Есть и еще один вариант, показанный на рис. 1.15. Он используется при выборе из меню Options строки List View.

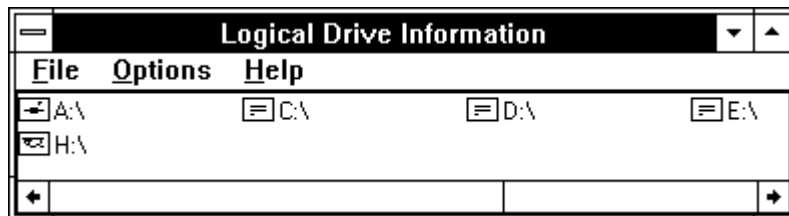


Рис. 1.15. Просмотр дисковых устройств в виде списка

Если же из меню Options выбрать строку Report view, список дисковых устройств будет отображаться в виде таблицы, как это было показано на рис. 1.6 и 1.7.

Приложение DiskInfo может работать и в среде операционной системы Microsoft Windows 95 (рис. 1.16).

Logical Drive Information					
File Options Help					
Drive	Volume name	File system	File name length	Total Space	Free Space
a:\		FAT	255	1457664	1239040
b:\		FAT	255	1213952	1213952
c:\	SYSTEM	FAT	255	398434304	74039296
d:\	DISTRIBUTVS	FAT	255	652214272	148045824
e:\	<Unknown>	?	0	0	0
f:\		FAT	255	210505728	207724544
g:\		NTFS	255	1885831168	504102912
s:\	SYSTEM	NWCOMPA	12	102400000	23879680
t:\	SOURCE	NWCOMPA	12	940670976	34340864
u:\	WORK	NWCOMPA	12	1048248320	80134144

Рис. 1.16. Просмотр информации о дисковых устройствах в среде операционной системы Microsoft Windows 95

Компьютер, на котором была запущена программа в этом случае, был оборудован 3,5 дюймовым НГМД (устройство A:), 5,25 дюймовым НГМД (устройство B:), обычным жестким диском (устройство C:), магнитооптическим накопителем MaxOptix с емкостью 1,3 Гбайт (устройство D:), и устройством чтения CD-ROM (устройство E:).

Кроме того, этот компьютер был подключен к сети, в которой есть серверы на базе операционных систем Microsoft Windows NT Server и Novell NetWare. Устройства F: и G: отображаются на сетевые тома сервера Microsoft Windows NT Server, а устройства S:, T: и U: - на сетевые тома сервера Novell NetWare.

Заметим, что в отличие от сервера Microsoft Windows NT Server, сервер Novell NetWare не был настроен таким образом, чтобы можно было работать с длинными именами файлов и каталогов.

### Исходные тексты приложения

Главный файл исходных текстов приложения DiskInfo приведен в листинге 1.5.

Заметим, что так как в приложении используется орган управления List View, в файле проекта вы должны подключить библиотеку comctl32.lib. В противном случае редактор связей выдаст сообщения об ошибках.

#### Листинг 1.5. Файл DiskInfo/DiskInfo.c

```
// =====
// Приложение DiskInfo
// Получение и отображение информации о дисковых
// устройствах, имеющихся в системе
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====

#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <stdio.h>
#include "resource.h"
#include "afxres.h"
#include "diskinfo.h"

// Структура для хранения информации о
// логическом диске
typedef struct tagDISKINFO
{
    char    szDriveName[10];    // имя диска
    UINT    nDriveType;        // тип диска
    char    szVolumeName[30];   // имя тома
    DWORD   dwVolumeSerialNumber; // серийный номер
    DWORD   dwMaxFileNameLength; // длина имени
    DWORD   dwFileSystemFlags;  // системные флаги
    char    szFileSystemName[10]; // имя файловой системы
    int     iImage;            // номер пиктограммы
    DWORD   dwFreeSpace;        // свободное пространство
    DWORD   dwTotalSpace;       // общий объем диска
} DISKINFO;

// -----
// Глобальные переменные
// -----

HINSTANCE hInst;
char szAppName[] = "DriveInfoApp";
char szAppTitle[] = "Logical Drive Information";
HWND hwndList;

// Указатель на область памяти, в которую будет
// записан массив строк имен дисков
LPSTR lpLogicalDriveStrings;
```

```
// Указатель на массив структур типа DISKINFO,
// в котором будет храниться информация о дисках
DISKINFO *pdi;

// Количество логических дисков в системе
int nNumDrives;

// -----
// Функция WinMain
// -----
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hWnd;
    MSG msg;

    hInst = hInstance;

    // Проверяем, не было ли это приложение запущено ранее
    hWnd = FindWindow(szAppName, NULL);
    if (hWnd)
    {
        if (IsIconic(hWnd))
            ShowWindow(hWnd, SW_RESTORE);
        SetForegroundWindow(hWnd);
        return FALSE;
    }

    // Регистрируем класс окна
    memset(&wc, 0, sizeof(wc));
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.hIconSm = LoadImage(hInst,
        MAKEINTRESOURCE(IDI_APPICONSM), IMAGE_ICON, 16, 16, 0);
    wc.style = 0;
    wc.lpfnWndProc = (WNDPROC)WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInst;
    wc.hIcon = LoadImage(hInst,
        MAKEINTRESOURCE(IDI_APPICON), IMAGE_ICON, 32, 32, 0);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
    wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
    wc.lpszClassName = szAppName;
    if (!RegisterClassEx(&wc))
        if (!RegisterClass((LPWNDCLASS)&wc.style))
            return FALSE;

    // Создаем главное окно приложения
    hWnd = CreateWindow(szAppName, szAppTitle,
```

```

    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInst, NULL);
if (!hWnd) return (FALSE);

// Отображаем окно и запускаем цикл обработки сообщений
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;
}

// -----
// Функция WndProc
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        HANDLE_MSG(hWnd, WM_CREATE,      WndProc_OnCreate);
        HANDLE_MSG(hWnd, WM_DESTROY,     WndProc_OnDestroy);
        HANDLE_MSG(hWnd, WM_COMMAND,     WndProc_OnCommand);
        HANDLE_MSG(hWnd, WM_NOTIFY,      WndProc_OnNotify);
        HANDLE_MSG(hWnd, WM_SIZE,        WndProc_OnSize);

        default:
            return (DefWindowProc(hWnd, msg, wParam, lParam));
    }
}

// -----
// Функция WndProc_OnCreate
// -----
BOOL WndProc_OnCreate(HWND hWnd,
                      LPCREATESTRUCT lpCreateStruct)
{
    int i;
    RECT rc;
    HIMAGELIST himlSmall;
    HIMAGELIST himlLarge;
    HICON hIcon;
    LV_COLUMN lvc;
    LV_ITEM lvi;

    // Получаем информацию о логических
    // дисковых устройствах
    GetDiskInfo();

```

```

// Определяем размеры внутренней области главного окна
GetClientRect(hWnd, &rc);

// Инициализируем библиотеку стандартных
// органов управления
InitCommonControls();

// Создаем орган управления List View
hwndList = CreateWindowEx(0L, WC_LISTVIEW, "",
    WS_VISIBLE | WS_CHILD | WS_BORDER | LVS_REPORT,
    0, 0, rc.right - rc.left, rc.bottom - rc.top,
    hWnd, (HMENU) IDC_LISTVIEW, hInst, NULL);

if (hwndList == NULL)
    return FALSE;

// Создаем списки изображений
himlSmall = ImageList_Create(GetSystemMetrics(SM_CXSMICON),
    GetSystemMetrics(SM_CYSMICON), ILC_MASK, 8, 1);
himlLarge = ImageList_Create(
    GetSystemMetrics(SM_CXICON), GetSystemMetrics(SM_CYICON),
    ILC_MASK, 8, 1);

// Добавляем в списки пиктограммы
// изображений дисковых устройств
// Изображения с номером 0
hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_DREMOVE));
ImageList_AddIcon(himlLarge, hIcon);

hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_DREMOVSM));
ImageList_AddIcon(himlSmall, hIcon);

// Изображения с номером 1
hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_DFIXED));
ImageList_AddIcon(himlLarge, hIcon);

hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_DFIXEDSM));
ImageList_AddIcon(himlSmall, hIcon);

// Изображения с номером 2
hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_DRCD));
ImageList_AddIcon(himlLarge, hIcon);

hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_DRCDSM));
ImageList_AddIcon(himlSmall, hIcon);

// Изображения с номером 3
hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_DNET));
ImageList_AddIcon(himlLarge, hIcon);

hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_DNETSM));
ImageList_AddIcon(himlSmall, hIcon);

```

```
// Добавляем списки изображений
ListView_SetImageList(hwndList, himlSmall, LVSIL_SMALL);
ListView_SetImageList(hwndList, himlLarge, LVSIL_NORMAL);
```

```
// Вставляем столбцы
memset(&lvc, 0, sizeof(lvc));
```

```
lvc.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;
lvc.fmt = LVCFMT_LEFT;
lvc.cx = (rc.right - rc.left) / 10;
```

```
lvc.iSubItem = 0;
lvc.pszText = "Drive";
ListView_InsertColumn(hwndList, 0, &lvc);
```

```
lvc.iSubItem = 1;
lvc.pszText = "Volume name";
ListView_InsertColumn(hwndList, 1, &lvc);
```

```
lvc.iSubItem = 2;
lvc.pszText = "File system";
ListView_InsertColumn(hwndList, 2, &lvc);
```

```
lvc.fmt = LVCFMT_RIGHT;
```

```
lvc.iSubItem = 3;
lvc.pszText = "File name length";
ListView_InsertColumn(hwndList, 3, &lvc);
```

```
lvc.iSubItem = 4;
lvc.pszText = "Total Space";
ListView_InsertColumn(hwndList, 4, &lvc);
```

```
lvc.iSubItem = 5;
lvc.pszText = "Free Space";
ListView_InsertColumn(hwndList, 5, &lvc);
```

```
// Вставляем строки
memset(&lvi, 0, sizeof(lvi));
```

```
lvi.mask = LVIF_IMAGE | LVIF_TEXT | LVIF_PARAM;
lvi.pszText = LPSTR_TEXTCALLBACK;
```

```
// Цикл по всем имеющимся логическим устройствам
for(i=0; i<nNumDirves; i++)
```

```
{
    lvi.iItem = i;
    lvi.iSubItem = 0;
    lvi.cchTextMax = 40;
    lvi.lParam = (LPARAM) (pdi + i)->szDriveName;
```

```
lvi.iImage = (pdi + i)->iImage;
ListView_InsertItem(hwndList, &lvi);
```

```
lvi.iItem = i;
lvi.iSubItem = 1;
ListView_InsertItem(hwndList, &lvi);
```

```
lvi.iItem = i;
lvi.iSubItem = 2;
ListView_InsertItem(hwndList, &lvi);
```

```
lvi.iItem = i;
lvi.iSubItem = 3;
ListView_InsertItem(hwndList, &lvi);
```

```
lvi.iItem = i;
lvi.iSubItem = 4;
ListView_InsertItem(hwndList, &lvi);
```

```
lvi.iItem = i;
lvi.iSubItem = 5;
ListView_InsertItem(hwndList, &lvi);
}
```

```
return TRUE;
}
```

```
// -----
// Функция WndProc_OnDestroy
// -----
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
```

```
{
    DestroyWindow(hwndList);
```

```
// Освобождаем память
free(lpLogicalDriveStrings);
free(pdi);
```

```
PostQuitMessage(0);
return 0L;
}
```

```
// -----
// Функция WndProc_OnCommand
// -----
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id,
    HWND hwndCtl, UINT codeNotify)
{
    DWORD dwStyle = 0;
```





```

        ltoa(lpDiskInfo->dwFreeSpace, szBuf, 10);
        lpLvdi->item.pszText = szBuf;
        break;
    default:
        break;
    }
    break;
}
}

case NM_DBLCLK:
{
    int index;
    char szBuf[256];

    // Определяем номер выбранного элемента списка
    index = ListView_GetNextItem(hwndList,
        -1, LVNI_ALL | LVNI_SELECTED);
    if(index == -1)
        return 0;

    // Получаем информацию о выбранном устройстве
    GetVolumeInformation((pdi + index)->szDriveName,
        (pdi + index)->szVolumeName, 30,
        &((pdi + index)->dwVolumeSerialNumber),
        &((pdi + index)->dwMaxFileNameLength),
        &((pdi + index)->dwFileSystemFlags),
        (pdi + index)->szFileSystemName, 10);

    // Определяем объем свободного пространства
    // на диске и общую емкость диска
    GetDiskFreeSpace((pdi + index)->szDriveName,
        &dwSectors, &dwBytes, &dwFreeClusters, &dwClusters);

    (pdi + index)->dwFreeSpace =
        dwSectors * dwBytes * dwFreeClusters;

    (pdi + index)->dwTotalSpace =
        dwSectors * dwBytes * dwClusters;

    // Подготавливаем для отображения имя диска,
    // имя файловой системы, серийный номер и
    // список системных флагов
    sprintf(szBuf, "System flags for drive %s (%s)\n"
        "Serial number: %lX\n",
        (pdi + index)->szDriveName,
        (pdi + index)->szFileSystemName,
        (pdi + index)->dwVolumeSerialNumber);

    if((pdi + index)->dwFileSystemFlags
        & FS_CASE_IS_PRESERVED)
        strcat(szBuf, "\nFS_CASE_IS_PRESERVED");

```

```

        if((pdi + index)->dwFileSystemFlags
            & FS_CASE_SENSITIVE)
            strcat(szBuf, "\nFS_CASE_SENSITIVE");

        if((pdi + index)->dwFileSystemFlags
            & FS_UNICODE_STORED_ON_DISK)
            strcat(szBuf, "\nFS_UNICODE_STORED_ON_DISK");

        if((pdi + index)->dwFileSystemFlags
            & FS_PERSISTENT_ACLS)
            strcat(szBuf, "\nFS_PERSISTENT_ACLS");

        if((pdi + index)->dwFileSystemFlags
            & FS_FILE_COMPRESSION)
            strcat(szBuf, "\nFS_FILE_COMPRESSION");

        if((pdi + index)->dwFileSystemFlags
            & FS_VOL_IS_COMPRESSED)
            strcat(szBuf, "\nFS_VOL_IS_COMPRESSED");

        // Перерисовываем главное окно приложения для
        // отражения изменений в окне списка
        InvalidateRect(hwnd, NULL, TRUE);

        MessageBox(hwnd, szBuf, szAppTitle, MB_OK);
        return 0L;
        break;
    }
}
return 0L;
}

// -----
// Функция WndProc_OnSize
// -----
#pragma warning(disable: 4098)
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy)
{
    MoveWindow(hwndList, 0, 0, cx, cy, TRUE);
    return FORWARD_WM_SIZE(hwnd, state, cx, cy, DefWindowProc);
}

// -----
// Функция GetDiskInfo
// -----
void GetDiskInfo(void)
{
    DWORD dwDriveStringsSpace;
    LPSTR lpTemp;
    int i;
    DWORD dwSectors, dwClusters, dwFreeClusters, dwBytes;

```

```

// Определяем размер блока памяти, необходимый для
// записи имен всех логических дисков
dwDriveStringsSpace = GetLogicalDriveStrings(0, NULL);

// Получаем память
lpLogicalDriveStrings = malloc(dwDriveStringsSpace);

// Заполняем полученный блок памяти именами дисков
GetLogicalDriveStrings(dwDriveStringsSpace,
    lpLogicalDriveStrings);

// Подсчитываем количество дисков, сканируя список
// имен, полученный на предыдущем шаге
nNumDirves = 0;
for(lpTemp = lpLogicalDriveStrings;
    *lpTemp != 0; nNumDirves++)
{
    lpTemp = strchr(lpTemp, 0) + 1;
}

// Заказываем память для хранения информации
// о всех дисках
pdi = malloc(nNumDirves * sizeof(DISKINFO));

// Заполняем массив структур DISKINFO информацией о дисках
for(i = 0, lpTemp = lpLogicalDriveStrings;
    i < nNumDirves; i++)
{
    // Получаем имя очередного диска
    strcpy((pdi + i)->szDriveName, lpTemp);

    // Определяем тип диска
    (pdi + i)->nDriveType = GetDriveType(lpTemp);

    // В зависимости от типа диска выбираем способ
    // заполнения соответствующей структуры DISKINFO
    switch ((pdi + i)->nDriveType)
    {
        // Для сменных устройств и для CD-ROM
        // записываем пустые значения
        case DRIVE_REMOVABLE:
        {
            // Выбираем пиктограмму с номером 0
            (pdi + i)->iImage = 0;

            strcpy((pdi + i)->szVolumeName, "<Unknown>");
            (pdi + i)->dwVolumeSerialNumber = 0;
            (pdi + i)->dwMaxFileNameLength = 0;
            (pdi + i)->dwFileSystemFlags = 0;
            strcpy((pdi + i)->szFileSystemName, "?");
            (pdi + i)->dwFreeSpace = 0;
        }
    }
}

```

```

        (pdi + i)->dwTotalSpace = 0;
    }
    break;
}

case DRIVE_CDROM:
{
    (pdi + i)->iImage = 2;
    strcpy((pdi + i)->szVolumeName, "<Unknown>");
    (pdi + i)->dwVolumeSerialNumber = 0;
    (pdi + i)->dwMaxFileNameLength = 0;
    (pdi + i)->dwFileSystemFlags = 0;
    strcpy((pdi + i)->szFileSystemName, "?");
    (pdi + i)->dwFreeSpace = 0;
    (pdi + i)->dwTotalSpace = 0;
    break;
}

// Получаем информацию для несменных устройств
case DRIVE_FIXED:
{
    (pdi + i)->iImage = 1;
    GetVolumeInformation(lpTemp,
        (pdi + i)->szVolumeName, 30,
        &((pdi + i)->dwVolumeSerialNumber),
        &((pdi + i)->dwMaxFileNameLength),
        &((pdi + i)->dwFileSystemFlags),
        (pdi + i)->szFileSystemName, 10);

    GetDiskFreeSpace(lpTemp, &dwSectors, &dwBytes,
        &dwFreeClusters, &dwClusters);

    (pdi + i)->dwFreeSpace =
        dwSectors * dwBytes * dwFreeClusters;

    (pdi + i)->dwTotalSpace =
        dwSectors * dwBytes * dwClusters;
    break;
}

// Получаем информацию для сетевых томов
case DRIVE_REMOTE:
{
    (pdi + i)->iImage = 3;
    GetVolumeInformation(lpTemp,
        (pdi + i)->szVolumeName, 30,
        &((pdi + i)->dwVolumeSerialNumber),
        &((pdi + i)->dwMaxFileNameLength),
        &((pdi + i)->dwFileSystemFlags),
        (pdi + i)->szFileSystemName, 10);

    GetDiskFreeSpace(lpTemp, &dwSectors, &dwBytes,

```

```

        &dwFreeClusters, &dwClusters);

(pdi + i)->dwFreeSpace =
    dwSectors * dwBytes * dwFreeClusters;

(pdi + i)->dwTotalSpace =
    dwSectors * dwBytes * dwClusters;
break;
}

// Прочие дисковые устройства
default:
{
    (pdi + i)->iImage = 1;
    GetVolumeInformation(lpTemp,
        (pdi + i)->szVolumeName, 30,
        &((pdi + i)->dwVolumeSerialNumber),
        &((pdi + i)->dwMaxFileNameLength),
        &((pdi + i)->dwFileSystemFlags),
        (pdi + i)->szFileSystemName, 10);

    GetDiskFreeSpace(lpTemp, &dwSectors, &dwBytes,
        &dwFreeClusters, &dwClusters);

    (pdi + i)->dwFreeSpace =
        dwSectors * dwBytes * dwFreeClusters;

    (pdi + i)->dwTotalSpace =
        dwSectors * dwBytes * dwClusters;
    break;
}
}

// Переходим к следующей строке в списке
// имен дисков
lpTemp = strchr(lpTemp, 0) + 1;
}
}

```

Файл diskinfo.h (листинг 1.6) содержит прототипы функций, определенных в приложении, а также определение идентификатора органа управления List View с именем IDC\_LISTVIEW.

Листинг 1.6. Файл DiskInfo/diskinfo.h

```

#define IDC_LISTVIEW 1234
// -----
// Описание функций
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

```

```

BOOL WndProc_OnCreate(HWND hWnd,
    LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
void WndProc_OnCommand(HWND hWnd, int id,
    HWND hwndCtl, UINT codeNotify);
LRESULT WndProc_OnNotify(HWND hWnd, int idFrom,
    NMHDR FAR* pnmhdr);
void WndProc_OnSize(HWND hWnd, UINT state, int cx, int cy);
void WndProc_OnDrawItem(HWND hWnd,
    const DRAWITEMSTRUCT * lpDrawItem);
void GetDiskInfo(void);

```

Файл resource.h (листинг 1.7) создается автоматически и содержит определения констант для файла описания ресурсов приложения, который будет приведен ниже.

Листинг 1.7. Файл DiskInfo/resource.h

```

//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by DiskInfo.RC
//
#define IDR_APPMENU 102
#define IDI_APPICON 103
#define IDI_APPICONSM 104
#define IDI_DREMOVE 115
#define IDI_DREMOVSM 116
#define IDI_DFIXED 117
#define IDI_DFIXEDSM 118
#define IDI_DRCD 119
#define IDI_DRCDSDM 120
#define IDI_DNET 121
#define IDI_DNETSM 122
#define ID_FILE_EXIT 40001
#define ID_HELP_ABOUT 40003
#define ID_OPTIONS_ICONVIEW 40004
#define ID_OPTIONS_SMALLICONVIEW 40005
#define ID_OPTIONS_LISTVIEW 40006
#define ID_OPTIONS_REPORTVIEW 40007

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 123
#define _APS_NEXT_COMMAND_VALUE 40008
#define _APS_NEXT_CONTROL_VALUE 1000
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif

```

Файл описания ресурсов приложения diskinfo.rc представлен в листинге 1.8.

#### Листинг 1.8. Файл DiskInfo/diskinfo.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

////////////////////
//
// Menu
//

IDR_APPMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",          ID_FILE_EXIT
    END
    POPUP "&Options"
    BEGIN
        MENUITEM "&Icon view",      ID_OPTIONS_ICONVIEW
        MENUITEM "&Small icon view", ID_OPTIONS_SMALLICONVIEW
        MENUITEM "&List view",       ID_OPTIONS_LISTVIEW
        MENUITEM "&Report view",     ID_OPTIONS_REPORTVIEW
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",       ID_HELP_ABOUT
    END
END

#ifdef APSTUDIO_INVOKED
```

```
////////////////////
//
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include \"afxres.h\"\\r\\n"
    "\\0"
END
3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\\r\\n"
    "\\0"
END
#endif // APSTUDIO_INVOKED

////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure
// application icon
// remains consistent on all systems.
IDI_APPICON          ICON          DISCARDABLE    "diskinfo.ico"
IDI_APPICONSM        ICON          DISCARDABLE    "dinfosm.ico"
IDI_DREMOVE          ICON          DISCARDABLE    "DREMOV.ICO"
IDI_DREMOVSM         ICON          DISCARDABLE    "DREMOVSM.ICO"
IDI_DFIXED           ICON          DISCARDABLE    "DFIXED.ICO"
IDI_DFIXEDSM         ICON          DISCARDABLE    "DFIXEDSM.ICO"
IDI_DRCD             ICON          DISCARDABLE    "DRCD.ICO"
IDI_DRCDSM           ICON          DISCARDABLE    "DRCDSM.ICO"
IDI_DNET             ICON          DISCARDABLE    "DNET.ICO"
IDI_DNETSM           ICON          DISCARDABLE    "DNETSM.ICO"

////////////////////
//
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    ID_FILE_EXIT        "Quits the application"
    ID_OPTIONS_ICONVIEW  "Each item appears as a full-sized icon"
    ID_OPTIONS_SMALLICONVIEW "Each item appears as a small icon"
    ID_OPTIONS_LISTVIEW  "Each item appears as a small icon"
    arranged in columns"
END
```

```
ID_OPTIONS_REPORTVIEW  "Each item appears with subitems arranged
in columns"
END
```

```
#endif    // English (U.S.) resources
//
//
// Generated from the TEXTINCLUDE 3 resource.
//
//
//
#endif    // not APSTUDIO_INVOKED
```

### Определения и глобальные переменные

В начале своей работы приложение DiskInfo получает список имен дисков в виде текстовых строк. Каждая такая строка закрыта двоичным нулем, а последняя - двумя нулевыми байтами. Адрес списка приложение записывает в глобальную переменную `lpLogicalDriveStrings`.

После получения списка имен приложение сканирует его с целью подсчета количества дисковых устройств. Это количество сохраняется в глобальной переменной `nNumDirves`.

Далее приложение заказывает память для массива структур типа `DISKINFO`:

```
typedef struct tagDISKINFO
{
    char  szDriveName[10];    // имя диска
    UINT  nDriveType;        // тип диска
    char  szVolumeName[30];   // имя тома
    DWORD dwVolumeSerialNumber; // серийный номер
    DWORD dwMaxFileNameLength; // длина имени
    DWORD dwFileSystemFlags;  // системные флаги
    char  szFileSystemName[10]; // имя файловой системы
    int   iImage;            // номер пиктограммы
    DWORD dwFreeSpace;        // свободное пространство
    DWORD dwTotalSpace;       // общий объем диска
} DISKINFO;
```

В каждой структуре этого массива будет храниться информация о соответствующем логическом диске, такая, например, как имя и тип диска, имя тома и так далее. В процессе работы приложение будет отображать и обновлять информацию, записанную в массиве.

Адрес массива структур `DISKINFO` хранится в глобальной переменной `pdi`.

### Описание функций

В этом разделе мы кратко расскажем о функциях, определенных в приложении DiskInfo.

#### Функция *WinMain*

Функция `WinMain` проверяет существование запущенной ранее копии приложения. Если такая копия будет обнаружена, окно работающей копии активизируется и выдвигается на передний план. Если нет - функция `WinMain` выполняет обычную инициализацию приложения, создавая его главное окно и запуская цикл обработки сообщений.

#### Функция *WndProc*

В задачу функции `WndProc` входит обработка сообщений `WM_CREATE`, `WM_DESTROY`, `WM_COMMAND`, `WM_NOTIFY` и `WM_SIZE`. Для обработки этих сообщений вызываются, соответственно, функции `WndProc_OnCreate`, `WndProc_OnDestroy`, `WndProc_OnCommand`, `WndProc_OnNotify` и `WndProc_OnSize`.

Все остальные сообщения передаются функции `DefWindowProc`, выполняющей обработку по умолчанию.

#### Функция *WndProc\_OnCreate*

Эта функция обрабатывает сообщение `WM_CREATE`, поступающее в функцию окна при его создании.

Вначале функция `WndProc_OnCreate` вызывает функцию `GetDiskInfo`, которая получает информацию о логических дисках, имеющихс в системе, и сохраняет ее в массиве структур `DISKINFO`.

Далее обработчик сообщения `WM_CREATE` определяет размеры внутренней области главного окна приложения, инициализирует библиотеку стандартных органов управления и создает орган управления `List View` на базе предопределенного класса окна `WC_LISTVIEW`, вызывая для этого функцию `CreateWindowEx`.

На следующем этапе приложение создает два списка изображений. Первый из них (с идентификатором `himiSmall`) будет содержать пиктограммы дисковых устройств маленького размера, а второй (с идентификатором `himiLarge`) - эти же пиктограммы, но стандартного размера.

С помощью макрокоманды `ImageList_AddIcon` в эти списки добавляются пиктограммы с изображениями дисков. Каждое такое изображение хранится в списке под собственным номером. Например, под номером 0 хранятся пиктограммы с идентификатором `IDI_DREMOVE` и `IDI_DREMOVSM` (сетевые устройства), под номером 1 - пиктограммы с идентификатором `IDI_DFIXED` и `IDI_DFIXEDSM` (диск с несменным носителем данных) и так далее. Номера пиктограмм используются при формировании строк списка, отображаемого при помощи органа управления `List View`.

После добавления всех пиктограмм сформированные списки подключаются к органу управления List View с помощью макрокоманды ListView\_SetImageList.

Далее обработчик сообщения WM\_CREATE вставляет столбцы, задавая для них различное выравнивание текста. Текст в столбцах Drive, Volume name и File system выравнивается по левой границе, а текст в столбцах File name length, Total Space и Free Space - по правой.

Вставка строк выполняется в цикле с помощью макрокоманды ListView\_InsertItem.

Более подробную информацию о работе с органом управления List View вы можете найти в 22 томе "Библиотеки системного программиста", посвященному программированию для операционной системы Microsoft Windows 95.

### Функция WndProc\_OnDestroy

При уничтожении главного окна приложения обработчик сообщения WM\_DESTROY удаляет орган управления List View и освобождает память, заказанную у операционной системы для списка строк с именами дисков и массива структур DISKINFO.

Далее обработчик вызывает функцию PostQuitMessage, иницилируя завершения цикла обработки сообщений.

### Функция WndProc\_OnCommand

Эта функция обрабатывает сообщение WM\_COMMAND, поступающее от главного меню приложения.

Выбирая строки меню Options, вы можете изменить внешний вид окна органа управления List View, выбрав один из четырех режимов отображения. Соответствующие процедуры мы описали в 22 томе "Библиотеки системного программиста".

### Функция WndProc\_OnNotify

Функция WndProc\_OnNotify обрабатывает извещения, поступающие от органа управления List View в главное окно приложения.

Обработчик извещения LVN\_GETDISPINFO выбирает информацию из элементов массива структур DISKINFO, и предоставляет ее для отображения в окне органа управления List View.

Для нас сейчас больший интерес представляет обработчик извещения NM\_DBLCLK, который получает управление, когда пользователь делает двойной щелчок левой клавишей мыши по пиктограмме дискового устройства в окне органа управления List View.

Вначале с помощью макрокоманды ListView\_GetNextItem обработчик извещения NM\_DBLCLK определяет номер выбранного элемента и записывает его в локальную переменную index.

Далее вызывается функция GetVolumeInformation, с помощью которой выполняется заполнение соответствующего элемента массива структур DISKINFO. Значение, записанное в переменную index служит при этом индексом в массиве структур.

Заполнение структуры завершается функцией GetDiskFreeSpace, с помощью которой определяется такая информация, как общий объем диска в байтах и объем свободного пространства на диске. Эта функция сохраняет в локальных переменных dwClusters и dwFreeClusters, соответственно, общее количество кластеров, имеющихся на диске, и количество свободных кластеров.

В локальные переменные dwSectors и dwBytes записывается количество секторов в одном кластере и размер сектора в байтах. Эти значения используются для вычисления общего и свободного объема диска исходя из общего количества кластеров и количества свободных кластеров.

После обновления элемента массива структур DISKINFO, соответствующего выбранному диску, обработчик извещения NM\_DBLCLK отображает на экране диалоговую панель с такой информацией, как название диска, имя файловой системы, серийный номер диска и системные флаги.

Перед выводом указанной диалоговой панели мы перерисовываем содержимое окна органа управления List View, для чего вызываем функцию InvalidateRect.

Зачем мы это делаем?

При запуске приложения мы получаем список логических дисковых устройств и определяем их параметры, однако только для устройств с несменными носителями данных. В противном случае операционная система попытается выполнить обращение к устройству (например, к НГМД) и, если вы заранее не вставите в него носитель данных, на экране появится сообщение об ошибке.

Если же вы вначале вставите носитель в устройство, а затем в окне нашего приложения DiskInfo сделаете двойной щелчок левой клавишей мыши по пиктограмме устройства, обработчик извещения NM\_DBLCLK получит параметры этого устройства и сохранит их в соответствующем элементе массива структур DISKINFO. Однако само по себе это не вызовет никаких изменений в списке параметров устройств, который отображается нашим приложением.

Вызывая функцию InvalidateRect, мы выполняем перерисовку главного окна приложения и его дочернего окна - окна органа управления List View. При этом обработчик извещения LVN\_GETDISPINFO получает и отображает в окне обновленную информацию о параметрах устройства.

### Функция WndProc\_OnSize

В задачу функции WndProc\_OnSize, обрабатывающей сообщение WM\_SIZE, входит изменение размеров органа управления List View при изменении размеров главного окна приложения.

### Функция GetDiskInfo

Функция GetDiskInfo вызывается обработчиком сообщения WM\_CREATE при создании главного окна приложения. Она получает и сохраняет информацию о всех логических дисках, имеющихся в системе.

Из предыдущего тома “Библиотеки системного программиста” вы знаете, что с помощью функции GetLogicalDriveStrings можно получить список имен всех логических дисковых устройств. Через второй параметр этой функции необходимо передать адрес блока памяти, в который будет записан указанный выше список, а через первый - размер этого блока.

Однако заранее приложение не знает, сколько логических дисков имеется в системе и, соответственно, не знает, сколько места потребуется ему для сохранения списка логических устройств. Поэтому вначале мы вызываем функцию GetLogicalDriveStrings, передав ей значение 0 в качестве первого параметра и значение NULL - в качестве второго:

```
dwDriveStringsSpace = GetLogicalDriveStrings(0, NULL);
```

В результате функция GetLogicalDriveStrings возвращает размер блока памяти, необходимый для записи всех имен логических дисков.

На следующем шаге наше приложение получает блок памяти необходимого размера, вызывая для этого функцию malloc. Адрес полученного блока и его размер затем передаются функции GetLogicalDriveStrings, которая в этом случае заполняет блок необходимой информацией:

```
GetLogicalDriveStrings(dwDriveStringsSpace,
    lpLogicalDriveStrings);
```

Если в компьютере имеются, например, логические диски A:, C: и D:, в блок памяти с адресом lpLogicalDriveStrings будут записаны следующие три строки:

```
A:\<0>
C:\<0>
D:\<0><0>
```

В конце каждой строки записывается байт с нулевым значением, а в конце последней строки - два нулевых байта.

Получив список имен устройств, наше приложение подсчитывает количество устройств. Для этого оно с помощью функции strchr сканирует список в цикле до тех пор, пока не будет найден его конец, подсчитывая количество проходов в глобальной переменной nNumDirves:

```
nNumDirves = 0;
for(lpTemp = lpLogicalDriveStrings; *lpTemp != 0;
    nNumDirves++)
{
    lpTemp = strchr(lpTemp, 0) + 1;
}
```

Определив таким образом общее количество логических дисков, приложение заказывает память для массива структур типа DISKINFO, в котором будут храниться параметры логических дисков:

```
pdi = malloc(nNumDirves * sizeof(DISKINFO));
```

Заполнение массива структур DISKINFO выполняется в цикле.

Для каждого диска прежде всего выполняется копирование имени диска из соответствующей строки списка, полученного ранее при помощи функции GetLogicalDriveStrings.

Далее приложение определяет тип диска, вызывая для этого функцию GetDriveType (в локальной переменной lpTemp хранится имя диска):

```
(pdi + i)->nDriveType = GetDriveType(lpTemp);
```

Заполнение полей структуры DISKINFO выполняется по-разному в зависимости от типа диска.

Если устройство со сменным носителем данных, то в поле ilmage, предназначенное для хранения номера пиктограммы диска, записывается нулевое значение. Именно под этим номером мы занесли пиктограмму диска со сменным носителем в список пиктограмм для органа управления List View.

В поле szVolumeName мы записываем строку <Unknown>, так как определение фактических параметров устройств со сменным носителем выполняется при обработке извещения NM\_DBLCLK. Аналогичным образом заполняются и остальные поля структуры.

Заполнение структуры DISKINFO для устройств чтения CD-ROM выполняется точно так же, как и устройств со сменным носителем данных, за исключением того что в поле номера пиктограммы ilmage записывается значение 2. Это номер пиктограммы с изображением накопителя CD-ROM в списке пиктограмм органа управления List View.

Если текущим устройством, для которого мы определяем параметры, является диск с несменным носителем данных, функция GetDiskInfo получает большинство этих параметров при помощи функции GetVolumeInformation, как это показано ниже:

```
GetVolumeInformation(lpTemp, (pdi + i)->szVolumeName, 30,
    &((pdi + i)->dwVolumeSerialNumber),
    &((pdi + i)->dwMaxFileNameLength),
    &((pdi + i)->dwFileSystemFlags),
    (pdi + i)->szFileName, 10);
```

Для определения общего объема диска и объема свободного пространства дополнительно вызывается функция GetDiskFreeSpace:

```
GetDiskFreeSpace(lpTemp, &dwSectors, &dwBytes, &dwFreeClusters,
    &dwClusters);
```

Значения, полученные от этой функции, обрабатываются следующим образом:

```
(pdi + i)->dwFreeSpace = dwSectors * dwBytes * dwFreeClusters;
(pdi + i)->dwTotalSpace = dwSectors * dwBytes * dwClusters;
```



Объем свободного пространства в байтах записывается в поле `dwFreeSpace`. Он вычисляется как произведение следующих величин: количества свободных кластеров на диске `dwFreeClusters`, количество секторов в одном кластере `dwSectors` и количества байт в одном секторе `dwBytes`.

Общий объем диска записывается в поле `dwTotalSpace` и подсчитывается аналогично, за исключением того что вместо количества свободных кластеров используется общее количество кластеров на диске `dwClusters`.

В поле `ilImage` записывается значение 1. Это номер пиктограммы с изображением диска с несменным носителем данных.

Получение и заполнение информации об удаленных (сетевых) дисковых устройствах выполняется аналогично, однако в поле `ilImage` записывается значение 3.

Если тип устройства не поддается классификации, наше приложение получает информацию о его параметрах, а в поле `ilImage` записывает значение 1.



## 2 ПЕРЕДАЧА ДАННЫХ МЕЖДУ ПРОЦЕССАМИ

Когда вы создавали приложения для операционной системы Microsoft Windows версии 3.1, вы могли организовать передачу данных между параллельно работающими приложениями либо через общую область памяти, либо с использованием механизма динамического обмена данными DDE, либо при помощи средств привязки и вставки объектов OLE, либо через универсальный буфер обмена Clipboard.

Что же касается Microsoft Windows NT, то в среде этой операционной системы вы по-прежнему можете пользоваться DDE, OLE и Clipboard, однако прямое создание глобальных областей памяти, доступных всем приложениям, невозможно. Причина этого лежит в том, что адресные пространства 32-разрядных приложений, работающих под управлением операционных систем Microsoft Windows NT и Microsoft Windows 95, полностью изолированы.

Что же предлагается взамен?

В программном интерфейсе Microsoft Windows NT предусмотрен достаточно широкий набор средств организации передачи данных между процессами, который вовсе не ограничивается относительно медленными механизмами DDE и OLE.

Прежде всего, вы можете организовать передачу данных между процессами, работающими в разных адресных пространствах, с использованием файлов, отображенных на память. О том, как работать с такими файлами, вы узнали из предыдущей главы.

Методика использования файлов, отображенных на память, для передачи данных между процессами заключается в следующем.

Один из процессов создает такой файл, задавая при этом имя отображения. Это имя является глобальным и доступно для всех процессов, запущенных в системе. Другие процессы могут воспользоваться именем отображения, открыв созданный ранее файл. В результате оба процесса могут получить указатели на область памяти, для которой выполнено отображение, и эти указатели будут ссылаться на одни и те же страницы виртуальной памяти. Обмениваясь данными через эту область, процессы должны обеспечить синхронизацию своей работы, например, с помощью критических секций, событий, объектов Mutex или семафоров (в зависимости от логики процесса обмена данными).

Если вы привыкли передавать данные вместе с сообщениями (что является общепринятой практикой в программировании для Microsoft Windows), то мы советуем вам обратить внимание на сообщение WM\_COPYDATA. Это сообщение позволяет передавать данные между

различными процессами за счет копирования передаваемых данных из адресного пространства одного процесса в адресное пространство другого процесса.

Еще один способ организации обмена данными между процессами заключается в организации специально предназначенных для этого каналов Pipe. Такие каналы напоминают файлы и достаточно удобны в работе. После их создания для обмена данными вы можете вызывать хорошо знакомые вам функции ReadFile и WriteFile, предназначенные для работы с файлами.

В том случае, когда требуется обеспечить передачу данных только в одном направлении, можно использовать так называемые каналы Mailslot.

Каналы Mailslot удобны тем, что их можно использовать для организации широкоэвентуальной передачи данных между процессами, запущенными на различных рабочих станциях сети. Что же касается каналов Pipe, то с их помощью можно организовать взаимодействие только двух процессов, но не широкоэвентуальную передачу данных.

### Обмен через файлы, отображаемые на память

Изучение средств обмена данными между процессами мы начнем с файлов, отображенных на память. Этот способ обладает высоким быстродействием, так как данные передаются между процессами непосредственно через виртуальную память.

Методика работы с файлами, отображаемыми на память, была описана в первой главе. Эта методика может быть использована без изменений для организации передачи данных между процессами, однако мы все же сделаем некоторые замечания.

Напомним, что отображение создается функцией CreateFileMapping.

Вот фрагмент кода из приложения Oem2Char, в котором создается отображение файла, а затем выполняется отображение этого файла в память:

```
hFileMapping = CreateFileMapping(hSrcFile,
    NULL, PAGE_READWRITE, 0, dwFileSize, NULL);
if(hFileMapping == NULL)
    return;

lpFileMap = MapViewOfFile(hFileMapping,
    FILE_MAP_WRITE, 0, 0, 0);
if(lpFileMap == 0)
    return;
```

Здесь в качестве первого параметра для функции CreateFileMapping мы передаем идентификатор файла, открытого функцией CreateFile. Последний параметр указан как NULL, поэтому отображение не имеет имени.

Если отображение будет использоваться для передачи данных между процессами, удобно указать для него имя. Пользуясь этим именем, другие процессы смогут открыть отображение функцией OpenFileMapping.

Другое замечание касается идентификатора файла, передаваемого функции `CreateFileMapping` через первый параметр. Если вы создаете отображение только для того чтобы обеспечить передачу данных между процессами, вам не нужно создавать файл на диске компьютера. Указав в качестве идентификатора файла значение `(HANDLE)0xFFFFFFFF`, вы создадите отображение непосредственно в виртуальной памяти без использования дополнительного файла.

Ниже мы привели фрагмент кода, в котором создается отображение с именем `$MyVerySpecialFileShareName$`, причем это отображение создается в виртуальной памяти:

```
CHAR lpFileShareName[] =
    "$MyVerySpecialFileShareName$";
hFileMapping = CreateFileMapping((HANDLE)0xFFFFFFFF,
    NULL, PAGE_READWRITE, 0, 100, lpFileShareName);
```

После того как вы создали объект-отображение, следует выполнить отображение файла в память при помощи функции `MapViewOfFile`, как это было показано выше. В случае успеха эта функция вернет указатель на отображенную область памяти.

Итак, первый процесс создал отображение. Второй процесс, который будет выполнять обмен данными с первым процессом, должен открыть это отображение по имени при помощи функции `OpenFileMapping`, например, так:

```
hFileMapping = OpenFileMapping(
    FILE_MAP_READ | FILE_MAP_WRITE, FALSE, lpFileShareName);
```

Далее второе приложение выполняет отображение, вызывая функцию `MapViewOfFile`:

```
lpFileMap = MapViewOfFile(hFileMapping,
    FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);
```

Пользуясь значением, полученным от функции `MapViewOfFile`, второе приложение получает указатель на отображенную область памяти. Физически эта область находится в тех же страницах виртуальной памяти, что и область, созданная первым процессом. Таким образом, два процесса получили указатели на общие страницы памяти.

Перед завершением своей работы процессы должны отменить отображение файла и освободить идентификатор созданного объекта-отображения:

```
UnmapViewOfFile(lpFileMap);
CloseHandle(hFileMapping);
```

### Приложение Fmap/Server

Для иллюстрации методики обмена данными между различными процессами с использованием файлов, отображаемых на память, мы подготовили исходные тексты двух консольных приложений: `Fmap/Server` и `Fmap/Client`. Эти приложения работают в паре (рис. 2.1).

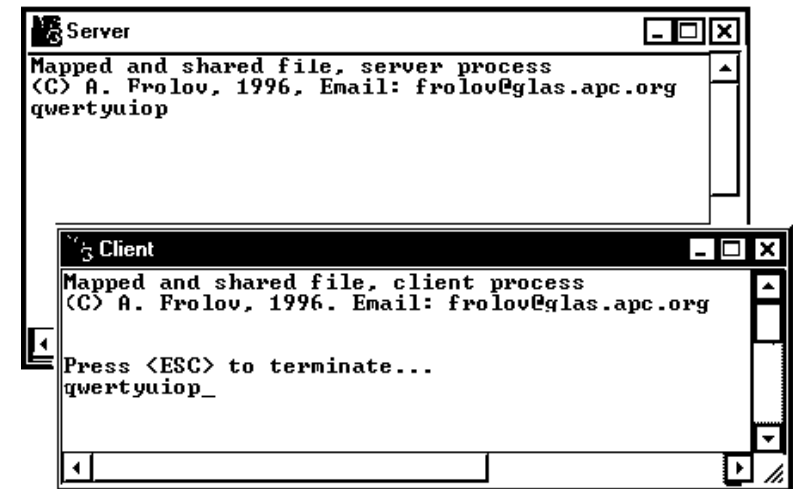


Рис. 2.1. Взаимодействие консольных приложений `Fmap/Server` и `Fmap/Client`

Приложение `Fmap/Server` создает отображение и два объекта-события. Первый объект предназначен для работы с клавиатурой, второй - для обнаружения момента завершения приложения `Fmap/Client`. Объекты-события были описаны нами в предыдущем томе "Библиотеки системного программиста", посвященном программированию для операционной системы Microsoft Windows NT.

Приложение `Fmap/Client` открывает созданное отображение и объекты-события, а затем в цикле вводит символы с клавиатуры, переключая один из объектов-событий в отмеченное состояние при вводе каждого символа. Коды введенных символов записываются в отображенную память.

По мере того как пользователь вводит символы в окне приложения `Fmap/Client`, приложение `Fmap/Server` отображает их в своем окне, получая коды введенных символов из отображенной памяти. Для синхронизации используется объект-событие, выделенное для работы с клавиатурой.

Если пользователь нажимает клавишу `<Esc>` в окне приложения `Fmap/Client`, это приложение отмечает оба события и завершает свою работу. Приложение `Fmap/Server`, обнаружив, что второй объект-событие оказался в отмеченном состоянии, также завершает свою работу. Таким образом, если завершить работу приложения `Fmap/Client`, то приложение `Fmap/Server` также будет завершено.

Исходный текст приложения `Fmap/Server` представлен в листинге 2.1.

Листинг 2.1. Файл `fmap/server/server.c`  

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

```

#include <stdio.h>
#include <conio.h>

// Идентификаторы объектов-событий, которые используются
// для синхронизации задач, принадлежащих разным процессам
HANDLE hEventChar;
HANDLE hEventTermination;
HANDLE hEvents[2];

// Имя объекта-события для синхронизации ввода и отображения
CHAR lpEventName[] =
    "$MyVerySpecialEventName$";

// Имя объекта-события для завершения процесса
CHAR lpEventTerminationName[] =
    "$MyVerySpecialEventTerminationName$";

// Имя отображения файла на память
CHAR lpFileShareName[] =
    "$MyVerySpecialFileShareName$";

// Идентификатор отображения файла на память
HANDLE hFileMapping;

// Указатель на отображенную область памяти
LPVOID lpFileMap;

int main()
{
    DWORD dwRetCode;

    printf("Mapped and shared file, server process\n"
        "(C) A. Frolov, 1996, Email: frolov@glas.apc.org\n");

    // Создаем объект-событие для синхронизации
    // ввода и отображения, выполняемого в разных процессах
    hEventChar = CreateEvent(NULL, FALSE, FALSE, lpEventName);

    // Если произошла ошибка, получаем и отображаем ее код,
    // а затем завершаем работу приложения
    if(hEventChar == NULL)
    {
        fprintf(stdout, "CreateEvent: Error %ld\n",
            GetLastError());
        getch();
        return 0;
    }

    // Если объект-событие с указанным именем существует,
    // считаем, что приложение EVENT уже было запущено
    if(GetLastError() == ERROR_ALREADY_EXISTS)
    {

```

```

        printf("\nApplication EVENT already started\n"
            "Press any key to exit...");
        getch();
        return 0;
    }

    // Создаем объект-событие для определения момента
    // завершения работы процесса ввода
    hEventTermination = CreateEvent(NULL,
        FALSE, FALSE, lpEventTerminationName);

    if(hEventTermination == NULL)
    {
        fprintf(stdout, "CreateEvent (Termination): Error %ld\n",
            GetLastError());
        getch();
        return 0;
    }

    // Создаем объект-отображение
    hFileMapping = CreateFileMapping((HANDLE)0xFFFFFFFF,
        NULL, PAGE_READWRITE, 0, 100, lpFileShareName);

    // Если создать не удалось, выводим код ошибки
    if(hFileMapping == NULL)
    {
        fprintf(stdout, "CreateFileMapping: Error %ld\n",
            GetLastError());
        getch();
        return 0;
    }

    // Выполняем отображение файла на память.
    // В переменную lpFileMap будет записан указатель на
    // отображаемую область памяти
    lpFileMap = MapViewOfFile(hFileMapping,
        FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

    // Если выполнить отображение не удалось,
    // выводим код ошибки
    if(lpFileMap == 0)
    {
        fprintf(stdout, "MapViewOfFile: Error %ld\n",
            GetLastError());
        getch();
        return 0;
    }

    // Готовим массив идентификаторов событий
    // для функции WaitForMultipleObjects
    hEvents[0] = hEventTermination;
    hEvents[1] = hEventChar;

```

```

// Цикл отображения. Этот цикл завершает свою работу
// при завершении процесса ввода
while(TRUE)
{
    // Выполняем ожидание одного из двух событий:
    // - завершение клиентского процесса;
    // - завершение ввода символа
    dwRetCode = WaitForMultipleObjects(2,
        hEvents, FALSE, INFINITE);

    // Если ожидание любого из двух событий было отменено,
    // если произошло первое событие (завершение клиентского
    // процесса) или если произошла ошибка, прерываем цикл
    if(dwRetCode == WAIT_ABANDONED_0 ||
        dwRetCode == WAIT_ABANDONED_0 + 1 ||
        dwRetCode == WAIT_OBJECT_0 ||
        dwRetCode == WAIT_FAILED)
        break;

    // Читаем символ из первого байта отображенной
    // области памяти, записанный туда клиентским
    // процессом, и отображаем его в консольном окне
    putchar(*(LPSTR)lpFileMap);
}

// Закрываем идентификаторы объектов-событий
CloseHandle(hEventChar);
CloseHandle(hEventTermination);

// Отменяем отображение файла
UnmapViewOfFile(lpFileMap);

// Освобождаем идентификатор созданного
// объекта-отображения
CloseHandle(hFileMapping);

return 0;
}

```

В глобальных переменных `hEventChar` и `hEventTermination` хранятся идентификаторы объектов-событий, предназначенных, соответственно, для работы с клавиатурой и для фиксации момента завершения работы приложения `Fmap/Client`. Эти же идентификаторы записываются в глобальный массив `hEvents`, который используется функцией `WaitForMultipleObjects`.

Глобальные имена объектов-событий хранятся в переменных `lpEventName` и `lpEventTerminationName`.

Имя отображения записывается в массив `lpFileShareName`, а идентификатор этого отображения - в глобальную переменную `hFileMapping`.

После выполнения отображения адрес отображенной области памяти, предназначенной для обмена данными с другим процессом, сохраняется в глобальной переменной `lpFileMap`.

Функция `main` приложения `Fmap/Server` создает два объекта-события, пользуясь для этого функцией `CreateEvent`. Описание этой функции вы найдете в предыдущем томе “Библиотеки системного программиста”.

Далее функция `main` создает объект-отображение и выполняет отображение, вызывая для этого, соответственно, функции `CreateFileMapping` и `MapViewOfFile`. Так как в качестве идентификатора файла функции `CreateFileMapping` передается значение `(HANDLE)0xFFFFFFFF`, отображение будет создано непосредственно в виртуальной памяти без использования файла, расположенного на диске.

После инициализации массива `hEvents` функция `main` запускает цикл, в котором выполняется ожидание событий и вывод символов, записанных приложением `Fmap/Client` в отображенную область виртуальной памяти.

Для ожидания двух событий используется функция `WaitForMultipleObjects`. Через третий параметр этой функции передается значение `FALSE`, поэтому ожидание прекращается в том случае, если любое из событий переходит в отмеченное состояние.

В том случае, когда в отмеченное состояние перешел объект-событие `hEventTermination`, функция `WaitForMultipleObjects` возвращает значение `WAIT_OBJECT_0`. Обнаружив это, функция `main` завершает свою работу, потому что событие `hEventTermination` отмечается при завершении работы клиентского приложения `Fmap/Client`.

Если же в отмеченное состояние переходит объект-событие `hEventChar`, функция `WaitForMultipleObjects` возвращает значение `WAIT_OBJECT_0 + 1`. В этом случае функция `main` читает первый байт из отображенной области памяти и выводит его в консольное окно при помощи хорошо знакомой вам из программирования для MS-DOS функции `putch`:

```
putch(*(LPSTR)lpFileMap);
```

Перед своим завершением функция `main` закрывает идентификаторы объектов-событий, отменяет отображение и освобождает идентификатор этого отображения.

### Приложение `Fmap/Client`

Исходные тексты приложения `Fmap/Client`, предназначенного для совместной работы с приложением `Fmap/Server`, представлены в листинге 2.2.

Листинг 2.2. Файл `fmap/client/client.c`

```

#include <windows.h>
#include <stdio.h>
#include <conio.h>

```

```
// Идентификаторы объектов-событий, которые используются
// для синхронизации задач, принадлежащих разным процессам
HANDLE hEvent;
HANDLE hEventTermination;

// Имя объекта-события для синхронизации ввода и отображения
CHAR lpEventName[] =
    "$MyVerySpecialEventName$";

// Имя объекта-события для завершения процесса
CHAR lpEventTerminationName[] =
    "$MyVerySpecialEventTerminationName$";

// Имя отображения файла на память
CHAR lpFileShareName[] =
    "$MyVerySpecialFileShareName$";

// Идентификатор отображения файла на память
HANDLE hFileMapping;

// Указатель на отображенную область памяти
LPVOID lpFileMap;

int main()
{
    CHAR chr;

    printf("Mapped and shared file, client process\n"
        "(C) A. Frolov, 1996, Email: frolov@glas.apc.org\n"
        "\n\nPress <ESC> to terminate...\n");

    // Открываем объект-событие для синхронизации
    // ввода и отображения
    hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, lpEventName);

    if(hEvent == NULL)
    {
        fprintf(stdout, "OpenEvent: Error %ld\n",
            GetLastError());
        getch();
        return 0;
    }

    // Открываем объект-событие для сигнализации о
    // завершении процесса ввода
    hEventTermination = OpenEvent(EVENT_ALL_ACCESS,
        FALSE, lpEventTerminationName);

    if(hEventTermination == NULL)
    {
        fprintf(stdout, "OpenEvent (Termination): Error %ld\n",
            GetLastError());
    }
}
```

```
getch();
return 0;
}

// Открываем объект-отображение
hFileMapping = OpenFileMapping(
    FILE_MAP_READ | FILE_MAP_WRITE, FALSE, lpFileShareName);

// Если открыть не удалось, выводим код ошибки
if(hFileMapping == NULL)
{
    fprintf(stdout, "OpenFileMapping: Error %ld\n",
        GetLastError());
    getch();
    return 0;
}

// Выполняем отображение файла на память.
// В переменную lpFileMap будет записан указатель на
// отображаемую область памяти
lpFileMap = MapViewOfFile(hFileMapping,
    FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

// Если выполнить отображение не удалось,
// выводим код ошибки
if(lpFileMap == 0)
{
    fprintf(stdout, "MapViewOfFile: Error %ld\n",
        GetLastError());
    getch();
    return 0;
}

// Цикл ввода. Этот цикл завершает свою работу,
// когда пользователь нажимает клавишу <ESC>,
// имеющую код 27
while(TRUE)
{
    // Проверяем код введенной клавиши
    chr = getche();

    // Если нажали клавишу <ESC>, прерываем цикл
    if(chr == 27)
        break;

    // Записываем символ в отображенную память,
    // доступную серверному процессу
    *((LPSTR)lpFileMap) = chr;

    // Устанавливаем объект-событие в отмеченное
    // состояние
    SetEvent(hEvent);
}
```

```

}

// После завершения цикла переключаем оба события
// в отмеченное состояние для отмены ожидания в
// процессе отображения и для завершения этого процесса
SetEvent(hEvent);
SetEvent(hEventTermination);

// Закрываем идентификаторы объектов-событий
CloseHandle(hEvent);
CloseHandle(hEventTermination);

// Отменяем отображение файла
UnmapViewOfFile(lpFileMap);

// Освобождаем идентификатор созданного
// объекта-отображения
CloseHandle(hFileMapping);

return 0;
}

```

После создания объектов-событий, предназначенных для синхронизации работы с приложением Fmap/Server, функция main приложения Fmap/Client открывает отображение при помощи функции OpenFileMapping, как это показано ниже:

```
hFileMapping = OpenFileMapping(
    FILE_MAP_READ | FILE_MAP_WRITE, FALSE, lpFileShareName);
```

В качестве имени отображения здесь указывается строка \$MyVerySpecialFileShareName\$ - точно такая же, что и в приложении Fmap/Server.

Далее в случае успеха выполняется отображение в память:

```
lpFileMap = MapViewOfFile(hFileMapping,
    FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);
```

Если отображение выполнено успешно, в глобальную переменную lpFileMap записывается указатель на отображенную область памяти, а затем запускается цикл ввода символов с клавиатуры.

Символы вводятся при помощи функции консольного ввода getch. Результат сохраняется в первом байте отображенной области памяти, откуда его будет брать для вывода приложение Fmap/Server:

```
chr = getch();
if(chr == 27)
    break;
*((LPSTR)lpFileMap) = chr;
```

После выполнения записи функция main устанавливает в отмеченное состояние объект-событие, предназначенное для работы с клавиатурой.

Если пользователь нажимает в окне приложения Fmap/Client клавишу <Esc>, имеющую код 27, цикл прерывается. Оба объекта-события переводятся в отмеченное состояние, после чего идентификаторы этих объектов освобождаются.

Перед завершением работы функция main отменяет отображение файла и освобождает идентификатор объекта-отображения.

## Передача сообщений между приложениями

Метод передачи данных между процессами, основанный на использовании файлов, отображенных на виртуальную память, работает достаточно быстро, так как процессы имеют прямой доступ к общим страницам памяти. Тем не менее, при использовании этого метода необходимо заботиться о синхронизации процессов, для чего следует использовать объекты синхронизации.

Если скорость передачи данных не является критичной, можно воспользоваться удобным способом передачи данных, не требующим синхронизации. Этот метод основан на передаче сообщения WM\_COPYDATA из одного приложения в другое при помощи функции SendMessage (функцию PostMessage для передачи этого сообщения использовать нельзя).

Сообщение WM\_COPYDATA использует оба параметра - wParam и lParam. Через параметр wParam необходимо передать идентификатор окна, посылающего сообщение. Параметр lParam используется для передачи указателя на предварительно заполненную структуру COPYDATASTRUCT, в которой находится ссылка на передаваемые данные.

Если приложение обрабатывает сообщение WM\_COPYDATA, то соответствующий обработчик должен вернуть значение TRUE, а если нет - FALSE.

Ниже мы привели формат структуры COPYDATASTRUCT:

```
typedef struct tagCOPYDATASTRUCT
{
    DWORD dwData; // 32-разрядные данные
    DWORD cbData; // размер передаваемого буфера с данными
    PVOID lpData; // указатель на буфер с данными
} COPYDATASTRUCT;
```

Перед посылкой сообщения WM\_COPYDATA приложение должно заполнить структуру COPYDATASTRUCT.

В поле dwData можно записать произвольное 32-разрядное значение, которое будет передано вместе с сообщением.

В поле lpData вы дополнительно можете записать указатель на область данных, полученную, например, при помощи функции HeapAlloc. Размер этой области следует записать в поле cbData.

Когда функция окна принимающего приложения получит сообщение WM\_COPYDATA, она должна скопировать в свой локальный буфер данные,



которые были переданы вместе с сообщением. И это единственное, что можно с этими данными сделать. Их, например, нельзя изменять. Не следует также надеяться, что данные сохранятся до обработки другого сообщения, поэтому копирование следует выполнить во время обработки сообщения WM\_COPYDATA.

Если вам достаточно передать из одного приложения в другое 32-разрядное значение, в поле lpData можно записать константу NULL.

### Приложение RCLOCK

Для иллюстрации методов работы с сообщением WM\_COPYDATA мы подготовили два приложения, которые называются RCLOCK и STIME.

Приложение RCLOCK раз в секунду получает от приложения STIME сообщение WM\_COPYDATA, вместе с которым передается строка текущего времени. Полученная строка отображается в небольшом окне, расположенном в левом нижнем углу рабочего стола (рис. 2.2).



Рис. 2.2. Окно приложения RCLOCK

Заметим, что сразу после запуска (если приложение STIME еще не активно) в окне приложения RCLOCK отображается строка <Unknown>, как это показано на рис. 2.3.

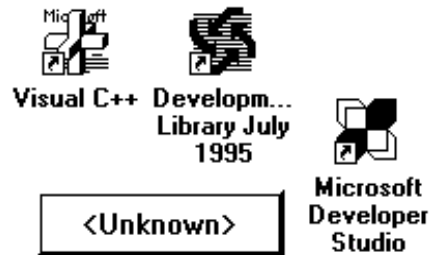


Рис. 2.3. Исходное состояние окна приложения RCLOCK

Если при активном приложении RCLOCK завершить приложение STIME, последнее передаст строку <Terminated>, которая будет показана в окне приложения RCLOCK.

Для того чтобы завершить работу приложения RCLOCK вы должны сделать его окно текущим, щелкнув по нему левой клавишей мыши, а затем нажать комбинацию клавиш <Ctrl+F4>. При необходимости вы сможете реализовать более удобный способ самостоятельно. В 11 томе "Библиотеки системного программиста" мы привели исходные тексты приложения TMCLOCK, из которого вы можете взять некоторые идеи для улучшения пользовательского интерфейса. Например, вы можете организовать перемещение окна приложения мышью, обрабатывая сообщение WM\_NCHITTEST.

### Исходные тексты приложения RCLOCK

Главный файл исходных текстов приложения RCLOCK представлен в листинге 2.3.

```
Листинг 2.3. Файл rclock/rclock.c
// =====
// Приложение RCLOCK (серверное)
// Демонстрация использования сообщения WM_COPYDATA
// для передачи данных между процессами
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====

#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "resource.h"
#include "afxres.h"
#include "rclock.h"

HINSTANCE hInst;
char szAppName[] = "RclockApp";
char szAppTitle[] = "Remote Clock";

// Метрики шрифта с фиксированной шириной символов
LONG cxChar, cyChar;

RECT rc;
char szBuf[80];

// -----
// Функция WinMain
// -----
int APIENTRY
```

```

WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hWnd;
    MSG msg;

    // Сохраняем идентификатор приложения
    hInst = hInstance;

    // Проверяем, не было ли это приложение запущено ранее
    hWnd = FindWindow(szAppName, NULL);
    if (hWnd)
    {
        // Если было, выдвигаем окно приложения на
        // передний план
        if (IsIconic(hWnd))
            ShowWindow(hWnd, SW_RESTORE);
        SetForegroundWindow(hWnd);
        return FALSE;
    }

    // Регистрируем класс окна
    memset(&wc, 0, sizeof(wc));
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.hIconSm = LoadImage(hInst,
        MAKEINTRESOURCE(IDI_APPICONSM),
        IMAGE_ICON, 16, 16, 0);
    wc.style = 0;
    wc.lpfnWndProc = (WNDPROC)WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInst;
    wc.hIcon = LoadImage(hInst,
        MAKEINTRESOURCE(IDI_APPICON),
        IMAGE_ICON, 32, 32, 0);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
    wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
    wc.lpszClassName = szAppName;
    if (!RegisterClassEx(&wc))
        if (!RegisterClass((LPWNDCLASS)&wc.style))
            return FALSE;

    // Создаем главное окно приложения
    hWnd = CreateWindow(szAppName, szAppTitle,
        WS_POPUPWINDOW | WS_THICKFRAME,
        100, 100, 100, 100,
        NULL, NULL, hInst, NULL);
    if (!hWnd) return (FALSE);

    // Размещаем окно в нижней левой части рабочего стола

```

```

    GetWindowRect(GetDesktopWindow(), &rc);

    MoveWindow(hWnd,
        rc.right - cxChar * 25,
        rc.bottom - cyChar * 3,
        cxChar * 10, cyChar * 2, TRUE);

    // Отображаем окно и запускаем цикл
    // обработки сообщений
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

// -----
// Функция WndProc
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
    switch (msg)
    {
        // Это сообщение посылается приложением STIME
        case WM_COPYDATA:
        {
            // Копируем данные, полученные от приложения STIME,
            // во внутренний буфер
            strcpy(szBuf, ((PCOPYDATASTRUCT)lParam)->lpData);

            // Перерисовываем содержимое окна, отображая в нем
            // полученную строку символов
            InvalidateRect(hWnd, NULL, TRUE);
            break;
        }

        HANDLE_MSG(hWnd, WM_CREATE, WndProc_OnCreate);
        HANDLE_MSG(hWnd, WM_DESTROY, WndProc_OnDestroy);
        HANDLE_MSG(hWnd, WM_PAINT, WndProc_OnPaint);

        default:
            return (DefWindowProc(hWnd, msg, wParam, lParam));
    }
}

// -----
// Функция WndProc_OnCreate

```



```
// -----
BOOL WndProc_OnCreate(HWND hWnd,
                     LPCREATESTRUCT lpCreateStruct)
{
    HDC hdc;
    TEXTMETRIC tm;

    hdc = GetDC(hWnd);

    // Выбираем в контекст отображения шрифт с фиксированной
    // шириной букв
    SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));

    // Определяем и сохраняем метрики шрифта
    GetTextMetrics(hdc, &tm);

    cxChar = tm.tmMaxCharWidth;
    cyChar = tm.tmHeight + tm.tmExternalLeading;

    ReleaseDC(hWnd, hdc);

    // Выполняем инициализацию буфера szBuf, содержимое
    // которого отображается в окне приложения
    strcpy(szBuf, (LPSTR)"<Unknown>");

    return TRUE;
}

// -----
// Функция WndProc_OnDestroy
// -----
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{
    PostQuitMessage(0);
    return 0L;
}

// -----
// Функция WndProc_OnPaint
// -----
#pragma warning(disable: 4098)
void WndProc_OnPaint(HWND hWnd)
{
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rc;

    // Перерисовываем внутреннюю область окна
    hdc = BeginPaint(hWnd, &ps);

    GetClientRect(hWnd, &rc);
```

```
// Рисуем в окне строку символов, полученную от
// приложения STIME
DrawText(hdc, szBuf, -1, &rc,
        DT_SINGLELINE | DT_CENTER | DT_VCENTER);

    EndPaint(hWnd, &ps);
    return 0;
}
```

### Определения и глобальные переменные

В глобальном массиве szAppName хранится текстовая строка с названием приложения. Это название будет использовано приложением STIME для поиска главного окна приложения RCLOCK.

В глобальных переменных cxChar и cyChar хранятся метрики шрифта с фиксированной шириной символов, которые будут определены на этапе создания главного окна приложения при обработке сообщения WM\_CREATE.

Структура rc типа RECT предназначена для хранения размеров окна рабочего стола.

Буфер szBuf используется для хранения данных, передаваемых из приложения STIME при помощи сообщения WM\_COPYDATA.

### Функция WinMain

Функция WinMain приложения RCLOCK сразу после запуска приложения выполняет поиск своей копии, используя для этого функцию FindWindow. Если такая копия найдена, главное окно этой копии выдвигается на передний план функцией SetForegroundWindow, после чего работа функции WinMain завершается. Такая техника уже использовалась нами ранее.

В том случае, когда запускается первая копия приложения RCLOCK, функция WinMain выполняет обычные действия. Она регистрирует класс окна и создает главное окно приложения. Для того чтобы это окно имело вид, показанный на рис. 2.2, для него указываются стили WS\_POPUPWINDOW и WS\_THICKFRAME:

```
hWnd = CreateWindow(szAppName, szAppTitle,
    WS_POPUPWINDOW | WS_THICKFRAME,
    100, 100, 100, 100, NULL, NULL, hInst, NULL);
```

Для определения размеров и расположения главного окна приложения RCLOCK функция WinMain определяет размеры окна рабочего стола, сохраняя их в глобальной переменной rc:

```
GetWindowRect(GetDesktopWindow(), &rc);
```

Размещение главного окна приложения RCLOCK выполняется функцией MoveWindow, как это показано ниже:

```
MoveWindow(hWnd,
    rc.right - cxChar * 25, rc.bottom - cyChar * 3,
```

```
cxChar * 10, cyChar * 2, TRUE);
```

Заметим, что метрики шрифта `cxChar` и `cyChar` определяются при обработке сообщения `WM_CREATE`, который получает управление при вызове функции `CreateWindow`. Поэтому после возвращения из функции `CreateWindow` содержимое глобальных переменных `cxChar` и `cyChar` будет отражать размеры рабочего стола.

После изменения размеров и расположения главного окна приложения `RCLOCK` выполняется отображение этого окна и запуск обычного цикла обработки сообщений.

### Функция `WndProc`

В задачу функции `WndProc` входит обработка сообщений `WM_CREATE`, `WM_DESTROY`, `WM_PAINT` и `WM_COPYDATA`. Для обработки первых трех сообщений при помощи макрокоманды `HANDLE_MSG` вызываются функции `WndProc_OnCreate`, `WndProc_OnDestroy` и `WndProc_OnPaint`, соответственно.

Для сообщения `WM_COPYDATA` в файле `windowsx.h`, к сожалению, не предусмотрены специальные макрокоманды. Мы могли бы подготовить такую макрокоманду самостоятельно, однако, так как обработка сообщения `WM_COPYDATA` очень проста, мы использовали классический способ:

```
case WM_COPYDATA:
{
    strcpy(szBuf, ((PCOPYDATASTRUCT)lParam)->lpData);
    InvalidateRect(hWnd, NULL, TRUE);
    break;
}
```

Напомним, что вместе с параметром `lParam` сообщения `WM_COPYDATA` передается указатель на структуру `COPYDATASTRUCT`. Приложение, посылающее другому приложению сообщение `WM_COPYDATA`, подготавливает область данных, записывая ее адрес в поле `lpData` структуры типа `COPYDATASTRUCT`. Принимающее приложение должно скопировать эти данные в свой внутренний буфер.

В нашем случае в качестве данных передается строка символов, закрытая двоичным нулем, поэтому для копирования мы используем функцию `strcpy`.

После выполнения копирования обработчик сообщения `WM_COPYDATA` вызывает функцию `InvalidateRect`, что в результате приводит к перерисовке главного окна приложения. В этом окне обработчик сообщения `WM_PAINT` нарисует текстовую строку, полученную от другого приложения и скопированную только что в буфер `szBuf`.

### Функция `WndProc_OnCreate`

Функция `WndProc_OnCreate` вызывается при создании главного окна приложения. Эта функция получает контекст отображения, выбирает в него шрифт с фиксированной шириной букв и определяет его метрики. Ширина и

высота символов сохраняются, соответственно, в глобальных переменных `cxChar` и `cyChar`. Эти значения используются для вычисления размеров главного окна приложения.

В завершении функция `WndProc_OnCreate` записывает в глобальный буфер `szBuf` строку `<Unknown>`:

```
strcpy(szBuf, (LPSTR)"<Unknown>");
```

Эта строка будет отображаться в главном окне приложения до тех пор, пока вы не запустите приложение `STIME`.

### Функция `WndProc_OnDestroy`

Эта функция завершает цикл обработки сообщений, вызывая для этого функцию `PostQuitMessage`.

### Функция `WndProc_OnPaint`

При обработке сообщения `WM_PAINT` функция `WndProc_OnPaint` отображает в главном окне приложения `RCLOCK` текстовую строку, записанную в буфере `szBuf`. Для рисования строки используется функция `DrawText`, так как с ее помощью легко выполнить центровку строки в окне по вертикали и горизонтали.

### Файл `rclock.h`

В файле `rclock.h` (листинг 2.4) находятся прототипы функций, определенных в приложении `RCLOCK`.

Листинг 2.4. Файл `rclock/rclock.h`

```
// -----
// Описание функций
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

void WndProc_OnDestroy(HWND hWnd);
BOOL WndProc_OnCreate(HWND hWnd,
                      LPCREATESTRUCT lpCreateStruct);
void WndProc_OnPaint(HWND hWnd);
```

### Файл `resource.h`

Файл `resource.h` (листинг 2.5) создается автоматически и содержит определения для файла описания ресурсов приложения `RCLOCK`.

Листинг 2.5. Файл `rclock/resource.h`

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by RCLOCK.RC
//
#define IDR_APPMENU
```

```
#define IDI_APPICON          103
#define IDI_APPICONSM       104

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        106
#define _APS_NEXT_COMMAND_VALUE        40006
#define _APS_NEXT_CONTROL_VALUE        1010
#define _APS_NEXT_SYMED_VALUE          101
#endif
#endif
```

### Файл rclock.rc

В файле rclock.rc (листинг 2.6) определены ресурсы приложения RCLOCK.

#### Листинг 2.6. Файл rclock/rclock.rc

//Microsoft Developer Studio generated resource script.

```
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

#ifdef APSTUDIO_INVOKED
////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
```

```
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include \"afxres.h\"\\r\\n"
    "\\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\\r\\n"
    "\\0"
END

#endif // APSTUDIO_INVOKED

////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure
// application icon
// remains consistent on all systems.
IDI_APPICON          ICON          DISCARDABLE    "RCLOCK.ICO"
IDI_APPICONSM        ICON          DISCARDABLE    "RCLOCKSM.ICO"

////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 162
        TOPMARGIN, 7
        BOTTOMMARGIN, 52
    END
END
#endif // APSTUDIO_INVOKED

#endif // English (U.S.) resources

////////////////////
#ifdef APSTUDIO_INVOKED
////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
```



```

wc.hInstance = hInst;
wc.hIcon = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICON),
    IMAGE_ICON, 32, 32, 0);
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
wc.lpszClassName = szAppName;
if(!RegisterClassEx(&wc))
    if(!RegisterClass((LPWNDCLASS)&wc.style))
        return FALSE;

// Создаем главное окно приложения
hWnd = CreateWindow(szAppName, szAppTitle,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInst, NULL);
if(!hWnd) return(FALSE);

// Отображаем окно и запускаем цикл
// обработки сообщений
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;
}

// -----
// Функция WndProc
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
    LPARAM lParam)
{
    switch(msg)
    {
        HANDLE_MSG(hWnd, WM_TIMER, WndProc_OnTimer);
        HANDLE_MSG(hWnd, WM_CREATE, WndProc_OnCreate);
        HANDLE_MSG(hWnd, WM_DESTROY, WndProc_OnDestroy);

        default:
            return(DefWindowProc(hWnd, msg, wParam, lParam));
    }
}

// -----
// Функция WndProc_OnCreate
// -----

```

```

BOOL WndProc_OnCreate(HWND hWnd,
    LPCREATESTRUCT lpCreateStruct)
{
    // Создаем таймер с периодом работы 1 сек
    SetTimer(hWnd, CLOCK_TIMER, 1000, NULL);
    return TRUE;
}

// -----
// Функция WndProc_OnTimer
// -----
#pragma warning(disable: 4098)
void WndProc_OnTimer(HWND hWnd, UINT id)
{
    time_t t;
    struct tm *ltime;

    // Определяем текущее время
    time(&t);
    ltime = localtime(&t);

    // Формируем текстовую строку времени
    sprintf(szBuf, "%02d:%02d:%02d",
        ltime->tm_hour, ltime->tm_min, ltime->tm_sec);

    // Записываем адрес и размер строки в структуру
    // типа COPYDATASTRUCT
    cd.lpData = szBuf;
    cd.cbData = strlen(szBuf) + 1;

    // Посылаем сообщение серверному приложению RCLOCK
    SendMessage(hWndServer, WM_COPYDATA,
        (WPARAM)hWnd, (LPARAM)&cd);
    return 0;
}

// -----
// Функция WndProc_OnDestroy
// -----
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{
    // Перед завершением работы приложения передаем
    // серверу строку <Terminated>
    cd.lpData = szTerminated;
    cd.cbData = strlen(szTerminated) + 1;

    SendMessage(hWndServer, WM_COPYDATA,
        (WPARAM)hWnd, (LPARAM)&cd);

    // Удаляем таймер
    KillTimer(hWnd, CLOCK_TIMER);
}

```

```
PostQuitMessage(0);
return 0L;
}
```

### Определения и глобальные переменные

Помимо всего прочего в области глобальных переменных определен массив `szServerAppName`, в котором хранится имя приложения RCLOCK. Это имя будет использовано в функции `WinMain` для поиска главного окна серверного приложения.

Идентификатор главного окна найденного приложения RCLOCK хранится в глобальной переменной `hWndServer`. Этот идентификатор используется для отправки сообщения `WM_COPYDATA` функцией `SendMessage`.

В области глобальных переменных определена структура `cd` типа `COPYDATASTRUCT`, которая совместно с глобальным буфером `szBuf` используется для передачи текстовой строки в приложение RCLOCK.

Кроме того, определен буфер `szTerminated`, в котором находится строка символов `Terminated`. Эта строка передается в приложение RCLOCK перед завершением работы приложения STIME.

### Функция WinMain

После поиска своей собственной копии приложение STIME ищет окно серверного приложения RCLOCK:

```
hWndServer = FindWindow(szServerAppName, NULL);
```

Если это приложение не найдено, выдается сообщение об ошибке, вслед за чем работа приложения STIME завершается.

В случае успешного поиска идентификатор найденного окна приложения RCLOCK записывается в глобальную переменную `hWndServer`. Вслед за этим выполняется процедура регистрации класса главного окна приложения STIME, создается главное окно приложения и запускается обычный цикл обработки сообщений.

### Функция WndProc

Функция `WndProc` обрабатывает сообщения `WM_CREATE`, `WM_DESTROY` и `WM_TIMER`, вызывая для этого функции `WndProc_OnCreate`, `WndProc_OnDestroy` и `WndProc_OnTimer`, соответственно.

### Функция WndProc\_OnCreate

При создании главного окна приложения STIME обработчик сообщения `WM_CREATE` создает таймер с периодом работы 1 сек, вызывая для этого функцию `SetTimer`:

```
SetTimer(hWnd, CLOCK_TIMER, 1000, NULL);
```

Созданный таймер будет иметь идентификатор `CLOCK_TIMER`.

### Функция WndProc\_OnTimer

Функция `WndProc_OnTimer` получает управление примерно один раз в секунду, обрабатывая сообщения `WM_TIMER`, поступающие от таймера с идентификатором `CLOCK_TIMER`.

Прежде всего эта функция формирует строку символов с текущим временем в формате "ЧЧ:ММ:СС", вызывая библиотечные функции системы разработки `time` и `localtime`:

```
time(&t);
ltime = localtime(&t);
wsprintf(szBuf, "%02d:%02d:%02d",
    ltime->tm_hour, ltime->tm_min, ltime->tm_sec);
```

Затем выполняется инициализация полей структуры `cd` типа `COPYDATASTRUCT`. В процессе инициализации мы записываем адрес буфера, содержащего строку символов, в поле `lpData`, а размер этого буфера (с учетом двоичного нуля, закрывающего строку) - в поле `cbData`:

```
cd.lpData = szBuf;
cd.cbData = strlen(szBuf) + 1;
```

Поле `dwData` не используется.

Заметим, что хотя серверное приложение RCLOCK при копировании полученных данных не использует поле `cbData` (так как мы передаем строку символов, закрытую двоичным нулем), при подготовке сообщения `WM_COPYDATA` к отправке необходимо заполнить оба поля: и `lpData`, и `cbData`.

Посылка сообщения `WM_COPYDATA` выполняется очень просто:

```
SendMessage(hWndServer, WM_COPYDATA,
    (WPARAM)hWnd, (LPARAM)&cd);
```

Сообщение посылается в окно с идентификатором `hWndServer`, который был определен в функции `WinMain`. В качестве параметра `wParam` вместе с этим сообщением необходимо передать идентификатор окна посылающего приложения, то есть идентификатор окна приложения STIME. Через параметр `lParam` передается адрес заполненной структуры `cd` типа `COPYDATASTRUCT`.

### Функция WndProc\_OnDestroy

Перед завершением своей работы приложение STIME посылает приложению RCLOCK строку `<Terminated>`, которая будет отображена в окне приложения RCLOCK. Для отправки используется только что описанная нами методика:

```
cd.lpData = szTerminated;
cd.cbData = strlen(szTerminated) + 1;
SendMessage(hWndServer, WM_COPYDATA,
    (WPARAM)hWnd, (LPARAM)&cd);
```

Далее функция WndProc\_OnDestroy удаляет таймер и завершает цикл обработки сообщений, вызывая для этого функцию PostQuitMessage.

#### Файл stime.h

В файле stime.h (листинг 2.8) определен идентификатор таймера CLOCK\_TIMER, а также прототипы функций.

Листинг 2.8. Файл rclock/stime/stime.h

```
#define CLOCK_TIMER 100

LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
void WndProc_OnDestroy(HWND hWnd);
BOOL WndProc_OnCreate(HWND hWnd,
    LPCREATESTRUCT lpCreateStruct);
void WndProc_OnPaint(HWND hWnd);
void WndProc_OnTimer(HWND hWnd, UINT id);
```

#### Файл resource.h

Файл resource.h (листинг 2.9) создается автоматически и содержит определения для файла описания ресурсов приложения STIME.

Листинг 2.9. Файл rclock/stime/resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by STIME.RC
//
#define IDR_APPMENU 102
#define IDI_APPICON 103
#define IDI_APPICONSM 104

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 106
#define _APS_NEXT_COMMAND_VALUE 40006
#define _APS_NEXT_CONTROL_VALUE 1010
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
```

#### Файл stime.rc

В файле stime.rc (листинг 2.10) определены ресурсы приложения STIME.

Листинг 2.10. Файл rclock/stime/stime.rc

```
//Microsoft Developer Studio generated resource script.
```

```
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

#ifdef APSTUDIO_INVOKED
////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include \"afxres.h\"\\r\\n"
    "\\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\\r\\n"
    "\\0"
END

#endif // APSTUDIO_INVOKED

////////////////////
//
// Icon
//
```



```
// Icon with lowest ID value placed first to ensure
// application icon
// remains consistent on all systems.
IDI_APPICON          ICON      DISCARDABLE     "TIME.ICO"
IDI_APPICONSM        ICON      DISCARDABLE     "TIMESM.ICO"

////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 162
        TOPMARGIN, 7
        BOTTOMMARGIN, 52
    END
END
#endif // APSTUDIO_INVOKED

#endif // English (U.S.) resources
////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
////////////////////////////////////
#endif // not APSTUDIO_INVOKED
```

## Каналы передачи данных Pipe

В среде операционной системы Microsoft Windows NT вам доступно такое удобное средство передачи данных между параллельно работающими процессами, как каналы типа Pipe. Это средство позволяет организовать передачу данных между локальными процессами, а также между процессами, запущенными на различных рабочих станциях в сети.

Каналы типа Pipe больше всего похожи на файлы, поэтому они достаточно просты в использовании.

Через канал можно передавать данные только между двумя процессами. Один из процессов создает канал, другой открывает его. После этого оба процесса могут передавать данные через канал в одну или обе стороны, используя для этого хорошо знакомые вам функции, предназначенные для

работы с файлами, такие как ReadFile и WriteFile. Заметим, что приложения могут выполнять над каналами Pipe синхронные или асинхронные операции, аналогично тому, как это можно делать с файлами. В случае использования асинхронных операций необходимо отдельно побеспокоиться об организации синхронизации.

## Именованные и анонимные каналы

Существуют две разновидности каналов Pipe - именованные (Named Pipes) и анонимные (Anonymous Pipes).

Как видно из названия, именованным каналам при создании присваивается имя, которое доступно для других процессов. Зная имя какой-либо рабочей станции в сети, процесс может получить доступ к каналу, созданному на этой рабочей станции.

Анонимные каналы обычно используются для организации передачи данных между родительскими и дочерними процессами, запущенными на одной рабочей станции или на "отдельно стоящем" компьютере.

## Имена каналов

Имена каналов в общем случае имеют следующий вид:

```
\\ИмяСервера\pipe\ИмяКанала
```

Если процесс открывает канал, созданный на другой рабочей станции, он должен указать имя сервера. Если же процесс создает канал или открывает канал на своей рабочей станции, вместо имени указывается символ точки:

```
\\.pipe\ИмяКанала
```

В любом случае процесс может создать канал только на той рабочей станции, где он запущен, поэтому при создании канала имя сервера никогда не указывается.

## Реализации каналов

В простейшем случае один серверный процесс создает один канал (точнее говоря, одну реализацию канала) для работы с одним клиентским процессом.

Однако часто требуется организовать взаимодействие одного серверного процесса с несколькими клиентскими. Например, сервер базы данных может принимать от клиентов запросы и рассылать ответы на них.

В случае такой необходимости серверный процесс может создать несколько реализаций канала, по одной реализации для каждого клиентского процесса.



## Функции для работы с каналами

В этом разделе мы опишем наиболее важные функции программного интерфейса Microsoft Windows NT, предназначенные для работы с каналами Pipes. Более подробную информацию вы найдете в документации, которая поставляется в составе библиотеки разработчика Microsoft Development Library (MSDN).

### Создание канала

Для создания именованных и анонимных каналов Pipes используются функции CreatePipe и CreateNamedPipe.

#### Функция CreatePipe

Анонимный канал создается функцией CreatePipe, имеющей следующий прототип:

```
BOOL CreatePipe(
    PHANDLE hReadPipe, // адрес переменной, в которую будет
                       // записан идентификатор канала для
                       // чтения данных
    PHANDLE hWritePipe, // адрес переменной, в которую будет
                       // записан идентификатор канала для
                       // записи данных
    LPSECURITY_ATTRIBUTES lpPipeAttributes, // адрес переменной
                                           // для атрибутов защиты
    DWORD nSize); // количество байт памяти,
                // зарезервированной для канала
```

Канал может использоваться как для записи в него данных, так и для чтения. Поэтому при создании канала функция CreatePipe возвращает два идентификатора, записывая их по адресу, заданному в параметрах hReadPipe и hWritePipe.

Идентификатор, записанный по адресу hReadPipe, можно передавать в качестве параметра функции ReadFile или ReadFileEx для выполнения операции чтения. Идентификатор, записанный по адресу hWritePipe, передается функции WriteFile или WriteFileEx для выполнения операции записи.

Через параметр lpPipeAttributes передается адрес переменной, содержащей атрибуты защиты для создаваемого канала. В наших приложениях мы будем указывать этот параметр как NULL. В результате канал будет иметь атрибуты защиты, принятые по умолчанию.

И, наконец, параметр nSize определяет размер буфера для создаваемого канала. Если этот размер указан как нуль, будет создан буфер с размером, принятым по умолчанию. Заметим, что при необходимости система может изменить указанный вами размер буфера.

В случае успеха функция CreatePipe возвращает значение TRUE, при ошибке - FALSE. В последнем случае для уточнения причины возникновения ошибки вы можете воспользоваться функцией GetLastError.

#### Функция CreateNamedPipe

Для создания именованного канала Pipe вы должны использовать функцию CreateNamedPipe. Вот прототип этой функции:

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName, // адрес строки имени канала
    DWORD dwOpenMode, // режим открытия канала
    DWORD dwPipeMode, // режим работы канала
    DWORD nMaxInstances, // максимальное количество
                       // реализаций канала
    DWORD nOutBufferSize, // размер выходного буфера в байтах
    DWORD nInBufferSize, // размер входного буфера в байтах
    DWORD nDefaultTimeout, // время ожидания в миллисекундах
    LPSECURITY_ATTRIBUTES lpSecurityAttributes); // адрес
                                                // переменной для атрибутов защиты
```

Через параметр lpName передается адрес строки имени канала в форме \\.\pipe\ИмяКанала (напомним, что при создании канала имя сервера не указывается, так как канал можно создать только на той рабочей станции, где запущен процесс, создающий канал).

Параметр dwOpenMode задает режим, в котором открывается канал. Остановимся на этом параметре подробнее.

Канал Pipe может быть ориентирован либо на передачу потока байт, либо на передачу сообщений. В первом случае данные через канал передаются по байтам, во втором - отдельными блоками заданной длины.

Режим работы канала (ориентированный на передачу байт или сообщений) задается, соответственно, константами PIPE\_TYPE\_BYTE или PIPE\_TYPE\_MESSAGE, которые указываются в параметре dwOpenMode. По умолчанию используется режим PIPE\_TYPE\_BYTE.

Помимо способа передачи данных через канал, с помощью параметра dwOpenMode можно указать, будет ли данный канал использован только для чтения данных, только для записи или одновременно для чтения и записи. Способ использования канала задается указанием одной из следующих констант:

Константа	Использование канала
PIPE_ACCESS_INBOUND	Только для чтения
PIPE_ACCESS_OUTBOUND	Только для записи
PIPE_ACCESS_DUPLEX	Для чтения и записи

Перечисленные выше параметры должны быть одинаковы для всех реализаций канала (о реализациях канала мы говорили выше). Далее мы перечислим параметры, которые могут отличаться для разных реализаций канала:

Константа	Использование канала
PIPE_READMODE_BYTE	Канал открывается на чтение в режиме последовательной передачи отдельных байт
PIPE_READMODE_MESSAGE	Канал открывается на чтение в режиме передачи отдельных сообщений указанной длины
PIPE_WAIT	Канал будет работать в блокирующем режиме, когда процесс переводится в состояние ожидания до завершения операций в канале
PIPE_NOWAIT	Неблокирующий режим работы канала. Если операция не может быть выполнена немедленно, в неблокирующем режиме функция завершается с ошибкой
FILE_FLAG_OVERLAPPED	Использование асинхронных операций (ввод и вывод с перекрытием). Данный режим позволяет процессу выполнять полезную работу параллельно с проведением операций в канале
FILE_FLAG_WRITE_THROUGH	В этом режиме функции, работающие с каналом, не возвращают управление до тех пор, пока не будет полностью завершена операция на удаленном компьютере. Используется только с каналом, ориентированном на передачу отдельных байт и только в том случае, когда канал создан между процессами, запущенными на различных станциях сети

Дополнительно к перечисленным выше флагам, через параметр `dwOpenMode` можно передавать следующие флаги защиты:

Флаг	Описание
WRITE_DAC	Вызывающий процесс должен иметь права доступа на запись к произвольному управляющему списку доступа именованного канала access control list (ACL)
WRITE_OWNER	Вызывающий процесс должен иметь права доступа на запись к процессу,

ACCESS_SYSTEM_SECURITY	владеющему именованным каналом Pipe Вызывающий процесс должен иметь права доступа на запись к управляющему списку доступа именованного канала access control list (ACL)
------------------------	--

Подробное описание этих флагов выходит за рамки нашей книги. При необходимости обратитесь к документации, входящей в состав SDK.

Теперь перейдем к параметру `dwPipeMode`, определяющему режим работы канала. В этом параметре вы можете указать перечисленные выше константы `PIPE_TYPE_BYTE`, `PIPE_TYPE_MESSAGE`, `PIPE_READMODE_BYTE`, `PIPE_READMODE_MESSAGE`, `PIPE_WAIT` и `PIPE_NOWAIT`. Для всех реализаций канала необходимо указывать один и тот же набор констант.

Параметр `nMaxInstances` определяет максимальное количество реализаций, которые могут быть созданы для канала. Вы можете указывать здесь значения от 1 до `PIPE_UNLIMITED_INSTANCES`. В последнем случае максимальное количество реализаций ограничивается только наличием свободных системных ресурсов.

Заметим, что если один серверный процесс использует несколько реализаций канала для связи с несколькими клиентскими, то общее количество реализаций может быть меньше, чем потенциальное максимальное количество клиентов. Это связано с тем, что клиенты могут использовать реализации по очереди, если только они не пожелают связаться с серверным процессом все одновременно.

Параметры `nOutBufferSize` и `nInBufferSize` определяют, соответственно, размер буферов, используемых для записи в канал и чтения из канала. При необходимости система может использовать буферы других, по сравнению с указанными, размеров.

Параметр `nDefaultTimeout` определяет время ожидания для реализации канала. Для всех реализаций необходимо указывать одинаковое значение этого параметра.

Через параметр `lpPipeAttributes` передается адрес переменной, содержащей атрибуты защиты для создаваемого канала. В наших приложениях мы будем указывать этот параметр как `NULL`. В результате канал будет иметь атрибуты защиты, принятые по умолчанию.

В случае успеха функция `CreateNamedPipe` возвращает идентификатор созданной реализации канала, который можно использовать в операциях чтения и записи, выполняемых с помощью таких функций, как `ReadFile` и `WriteFile`.

При ошибке функция `CreateNamedPipe` возвращает значение `INVALID_HANDLE_VALUE`. Код ошибки вы можете уточнить, вызвав функцию `GetLastError`.

### Использование функции CreateFile

Функция CreateFile, предназначенная для работы с файлами, может также быть использована для создания новых каналов и открытия существующих. При этом вместо имени файла вы должны указать этой функции имя канала Pipe.

### Пример использования функции CreateNamedPipe

Приведем пример использования функции CreateNamedPipe для создания именованного канала Pipe с именем \$MyPipe\$, предназначенным для чтения и записи данных, работающем в блокирующем режиме и допускающем создание неограниченного количества реализаций:

```
HANDLE hNamedPipe;
LPSTR lpszPipeName = "\\\\.\\pipe\\$MyPipe$";
hNamedPipe = CreateNamedPipe(
    lpszPipeName,
    PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
    PIPE_UNLIMITED_INSTANCES,
    512, 512, 5000, NULL);
```

Через создаваемый канал передаются сообщения (так как указана константа PIPE\_TYPE\_MESSAGE). Данная реализация предназначена только для чтения (константа PIPE\_READMODE\_MESSAGE).

При создании канала мы указали размер буферов ввода и вывода, равный 512 байт. Время ожидания операций выбрано равным 5 секунд. Атрибуты защиты не указаны, поэтому используются значения, принятые по умолчанию.

### Установка соединения с каналом со стороны сервера

После того как серверный процесс создал канал, он может перейти в режим соединения с клиентским процессом. Соединение со стороны сервера выполняется с помощью функции ConnectNamedPipe.

Прототип функции ConnectNamedPipe представлен ниже:

```
BOOL ConnectNamedPipe(
    HANDLE hNamedPipe, // идентификатор именованного канала
    LPOVERLAPPED lpOverlapped); // адрес структуры OVERLAPPED
```

Через первый параметр серверный процесс передает этой функции идентификатор канала, полученный от функции CreateNamedPipe.

Второй параметр используется только для организации асинхронного обмена данными через канал. Если вы используете только синхронные операции, в качестве значения для этого параметра можно указать NULL.

В случае успеха функция ConnectNamedPipe возвращает значение TRUE, а при ошибке - FALSE. Код ошибки можно получить с помощью функции GetLastError.

В зависимости от различных условий функция ConnectNamedPipe может вести себя по-разному.

Если параметр lpOverlapped указан как NULL, функция выполняется в синхронном режиме. В противном случае используется асинхронный режим.

Для канала, созданного в синхронном блокирующем режиме (с использованием константы PIPE\_WAIT), функция ConnectNamedPipe переходит в состояние ожидания соединения с клиентским процессом. Именно этот режим мы будем использовать в наших примерах программ, исходные тексты которых вы найдете ниже.

Если канал создан в синхронном неблокирующем режиме, функция ConnectNamedPipe немедленно возвращает управление с кодом TRUE, если только клиент был отключен от данной реализации канала и возможно подключение этого клиента. В противном случае возвращается значение FALSE. Дальнейший анализ необходимо выполнять с помощью функции GetLastError. Эта функция может вернуть значение ERROR\_PIPE\_LISTENING (если к серверу еще не подключен ни один клиент), ERROR\_PIPE\_CONNECTED (если клиент уже подключен) или ERROR\_NO\_DATA (если предыдущий клиент отключился от сервера, но клиент еще не завершил соединение).

Ниже мы приведем пример использования функции ConnectNamedPipe:

```
HANDLE hNamedPipe;
LPSTR lpszPipeName = "\\\\.\\pipe\\$MyPipe$";
hNamedPipe = CreateNamedPipe(
    lpszPipeName,
    PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
    PIPE_UNLIMITED_INSTANCES,
    512, 512, 5000, NULL);
fConnected = ConnectNamedPipe(hNamedPipe, NULL);
```

В данном случае функция ConnectNamedPipe перейдет в состояние ожидания, так как канал был создан для работы в синхронном блокирующем режиме.

### Установка соединения с каналом со стороны клиента

Для создания канала клиентский процесс может воспользоваться функцией CreateFile. Как вы помните, эта функция предназначена для работы с файлами, однако с ее помощью можно также открыть канал, указав его имя вместо имени файла. Забегая вперед, скажем, что функция CreateFile позволяет открывать не только файлы или каналы Pipe, но и другие системные ресурсы, например, устройства и каналы Mailslot.

Функция CreateFile была нами описана в предыдущем томе "Библиотеки системного программиста", однако для удобства мы повторим прототип этой функции и ее краткое описание:

Итак, прототип функции CreateFile:

```
HANDLE CreateFile(
    LPCTSTR lpFileName, // адрес строки имени файла
    DWORD dwDesiredAccess, // режим доступа
```

```

DWORD dwShareMode, // режим совместного использования файла
LPSECURITY_ATTRIBUTES lpSecurityAttributes, // дескриптор
// защиты
DWORD dwCreationDistribution, // параметры создания
DWORD dwFlagsAndAttributes, // атрибуты файла
HANDLE hTemplateFile); // идентификатор файла с атрибутами

```

Раньше при работе с файлами через параметр lpFileName вы передавали этой функции адрес строки, содержащей имя файла, который вы собираетесь создать или открыть. Строка должна быть закрыта двоичным нулем. Если функция CreateFile работает с каналом Pipe, параметр lpFileName определяет имя канала.

Параметр dwDesiredAccess определяет тип доступа, который должен быть предоставлен к открываемому файлу. В нашем случае этот тип доступа будет относиться к каналу Pipe. Здесь вы можете использовать логическую комбинацию следующих констант:

Константа	Описание
0	Доступ запрещен, однако приложение может определять атрибуты файла, канала или устройства, открываемого при помощи функции CreateFile
GENERIC_READ	Разрешен доступ на чтение из файла или канала Pipe
GENERIC_WRITE	Разрешен доступ на запись в файл или канал Pipe

Тип доступа, указанный при помощи параметра dwDesiredAccess, не должен противоречить типу доступа для канала, заданного при его создании функцией CreateNamedPipe.

С помощью параметра dwShareMode задаются режимы совместного использования открываемого или создаваемого файла. Для этого параметра вы можете указать комбинацию следующих констант:

Константа	Описание
0	Совместное использование файла запрещено
FILE_SHARE_READ	Другие приложения могут открывать файл с помощью функции CreateFile для чтения
FILE_SHARE_WRITE	Аналогично предыдущему, но на запись

Через параметр lpSecurityAttributes необходимо передать указатель на дескриптор защиты или значение NULL, если этот дескриптор не используется. В наших приложениях мы не работаем с дескриптором защиты.

Параметр dwCreationDistribution определяет действия, выполняемые функцией CreateFile, если приложение пытается создать файл, который уже существует. Для этого параметра вы можете указать одну из следующих констант:

Константа	Описание
CREATE_NEW	Если создаваемый файл уже существует, функция CreateFile возвращает код ошибки
CREATE_ALWAYS	Существующий файл перезаписывается, при этом содержимое старого файла теряется
OPEN_EXISTING	Открывается существующий файл. Если файл с указанным именем не существует, функция CreateFile возвращает код ошибки
OPEN_ALWAYS	Если указанный файл существует, он открывается. Если файл не существует, он будет создан
TRUNCATE_EXISTING	Если файл существует, он открывается, после чего длина файла устанавливается равной нулю. Содержимое старого файла теряется. Если же файл не существует, функция CreateFile возвращает код ошибки

Параметр dwFlagsAndAttributes задает атрибуты и флаги для файла.

При этом можно использовать любые логические комбинации следующих атрибутов (кроме атрибута FILE\_ATTRIBUTE\_NORMAL, который можно использовать только отдельно):

Атрибут	Описание
FILE_ATTRIBUTE_ARCHIVE	Файл был архивирован (выгружен)
FILE_ATTRIBUTE_COMPRESSED	Файл, имеющий этот атрибут, динамически сжимается при записи и восстанавливается при чтении. Если этот атрибут имеет каталог, то для всех расположенных в нем файлов и каталогов также выполняется динамическое сжатие данных
FILE_ATTRIBUTE_NORMAL	Остальные перечисленные в этом списке атрибуты не установлены
FILE_ATTRIBUTE_HIDDEN	Скрытый файл
FILE_ATTRIBUTE_READONLY	Файл можно только читать
FILE_ATTRIBUTE_SYSTEM	Файл является частью операционной

## СИСТЕМЫ

В дополнение к перечисленным выше атрибутам, через параметр `dwFlagsAndAttributes` вы можете передать любую логическую комбинацию флагов, перечисленных ниже:

Флаг	Описание
<code>FILE_FLAG_WRITE_THROUGH</code>	Отмена промежуточного кэширования данных для уменьшения вероятности потери данных при аварии
<code>FILE_FLAG_NO_BUFFERING</code>	Отмена промежуточной буферизации или кэширования. При использовании этого флага необходимо выполнять чтение и запись порциями, кратными размеру сектора (обычно 512 байт)
<code>FILE_FLAG_OVERLAPPED</code>	Асинхронное выполнение чтения и записи. Во время асинхронного чтения или записи приложение может продолжать обработку данных
<code>FILE_FLAG_RANDOM_ACCESS</code>	Указывает, что к файлу будет выполняться произвольный доступ. Флаг предназначен для оптимизации кэширования
<code>FILE_FLAG_SEQUENTIAL_SCAN</code>	Указывает, что к файлу будет выполняться последовательный доступ от начала файла к его концу. Флаг предназначен для оптимизации кэширования
<code>FILE_FLAG_DELETE_ON_CLOSE</code>	Файл будет удален сразу после того как приложение закроет его идентификатор. Этот флаг удобно использовать для временных файлов
<code>FILE_FLAG_BACKUP_SEMANTICS</code>	Файл будет использован для выполнения операции выгрузки или восстановления. При этом выполняется проверка прав доступа
<code>FILE_FLAG_POSIX_SEMANTICS</code>	Доступ к файлу будет выполняться в соответствии со спецификацией POSIX

И, наконец, последний параметр `hTemplateFile` предназначен для доступа к файлу шаблона с расширенными атрибутами создаваемого файла.

В случае успешного завершения функция `CreateFile` возвращает идентификатор созданного или открытого файла (или каталога), а при работе с каналом Pipe - идентификатор реализации канала.

При ошибке возвращается значение `INVALID_HANDLE_VALUE` (а не `NULL`, как можно было бы предположить). Код ошибки можно определить при помощи функции `GetLastError`.

В том случае, если файл уже существует и были указаны константы `CREATE_ALWAYS` или `OPEN_ALWAYS`, функция `CreateFile` не возвращает код ошибки. В то же время в этой ситуации функция `GetLastError` возвращает значение `ERROR_ALREADY_EXISTS`.

Приведем фрагмент исходного текста клиентского приложения, открывающего канал с именем `$MyPipe$` при помощи функции `CreateFile`:

```
char    szPipeName[256];
HANDLE hNamedPipe;
strcpy(szPipeName, "\\.\pipe\\$MyPipe$");
hNamedPipe = CreateFile(
    szPipeName, GENERIC_READ | GENERIC_WRITE,
    0, NULL, OPEN_EXISTING, 0, NULL);
```

Здесь канал открывается как для записи, так и для чтения.

### Отключение серверного процесса от клиентского процесса

Если сервер работает с несколькими клиентскими процессами, то он может использовать для этого несколько реализаций канала, причем одни и те же реализации могут применяться по очереди.

Установив канал с клиентским процессом при помощи функции `ConnectNamedPipe`, серверный процесс может затем разорвать канал, вызвав для этого функцию `DisconnectNamedPipe`. После этого реализация канала может быть вновь использована для соединения с другим клиентским процессом.

Прототип функции `DisconnectNamedPipe` мы привели ниже:

```
BOOL DisconnectNamedPipe(HANDLE hNamedPipe);
```

Через параметр `hNamedPipe` функции передается идентификатор реализации канала Pipe, полученный от функции `CreateNamedPipe`.

В случае успеха функция возвращает значение `TRUE`, а при ошибке - `FALSE`. Код ошибки можно получить от функции `GetLastError`.

### Закрывание идентификатора канала

Если канал больше не нужен, после отключения от клиентского процесса серверный и клиентский процессы должны закрыть его идентификатор функцией `CloseHandle`:

```
CloseHandle(hNamedPipe);
```



### Запись данных в канал

Запись данных в открытый канал выполняется с помощью функции WriteFile, аналогично записи в обычный файл:

```
HANDLE hNamedPipe;
DWORD  cbWritten;
char    szBuf[256];
WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1,
          &cbWritten, NULL);
```

Через первый параметр функции WriteFile передается идентификатор реализации канала. Через второй параметр передается адрес буфера, данные из которого будут записаны в канал. Размер этого буфера указывается при помощи третьего параметра. Предпоследний параметр используется для определения количества байт данных, действительно записанных в канал. И, наконец, последний параметр задан как NULL, поэтому запись будет выполняться в синхронном режиме.

Учтите, что если канал был создан для работы в блокирующем режиме, и функция WriteFile работает синхронно (без использования вывода с перекрытием), то эта функция не вернет управление до тех пор, пока данные не будут записаны в канал.

### Чтение данных из канала

Как и следовало ожидать, для чтения данных из канала можно воспользоваться функцией ReadFile, например, так:

```
HANDLE hNamedPipe;
DWORD  cbRead;
char    szBuf[256];
ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL);
```

Данные, прочитанные из канала hNamedPipe, будут записаны в буфер szBuf, имеющий размер 512 байт. Количество действительно прочитанных байт данных будет сохранено функцией ReadFile в переменной cbRead. Так как последний параметр функции указан как NULL, используется синхронный режим работы без перекрытия.

### Другие функции

Среди других функций, предназначенных для работы с каналами Pipe, мы рассмотрим функции CallNamedPipe, TransactNamedPipe, PeekNamedPipe, WaitNamedPipe, SetNamedPipeHandleState, GetNamedPipeInfo, GetNamedPipeHandleState.

#### Функция CallNamedPipe

Обычно сценарий взаимодействия клиентского процесса с серверным заключается в выполнении следующих операций:

- подключение к каналу с помощью функции CreateFile;

- выполнение операций чтения или записи такими функциями как ReadFile или WriteFile;
- отключение от канала функцией CloseHandle.

Функция CallNamedPipe позволяет выполнить эти операции за один прием, при условии что канал открыт в режиме передачи сообщений и что клиент посылает одно сообщение серверу и в ответ также получает от сервера одно сообщение.

Ниже мы приведем прототип функции CallNamedPipe:

```
BOOL CallNamedPipe(
    LPCTSTR lpNamedPipeName, // адрес имени канала
    LPVOID lpOutBuffer,      // адрес буфера для записи
    DWORD nOutBufferSize,   // размер буфера для записи
    LPVOID lpInBuffer,      // адрес буфера для чтения
    DWORD nInBufferSize,    // размер буфера для чтения
    LPDWORD lpBytesRead,     // адрес переменной для записи
                             // количества прочитанных байт данных
    DWORD nTimeout);        // время ожидания в миллисекундах
```

Перед вызовом функции CallNamedPipe вы должны записать в параметр lpNamedPipeName указатель на текстовую строку, содержащую имя канала Pipe. При формировании этой строки вы должны руководствоваться теми же правилами, что и при использовании функции CreateFile.

Кроме имени канала, вы также должны подготовить буфер, содержащий передаваемые через канал данные. Адрес и размер этого буфера следует указать функции CallNamedPipe, соответственно, через параметры lpOutBuffer и nOutBufferSize.

Данные, полученные от сервера в ответ на посланное ему сообщение, будут записаны в буфер, который вы тоже должны подготовить заранее. Адрес этого буфера необходимо указать через параметр lpInBuffer, а размер буфера - через параметр nInBufferSize.

В переменную, адрес которой указан через параметр lpBytesRead, записывается количество байт, полученных через канал от сервера.

Параметр nTimeout определяет, в течении какого времени функция CallNamedPipe будет ожидать доступности канала Pipe, прежде чем она вернет код ошибки. Помимо численного значения в миллисекундах, вы можете указать в этом параметре одну из следующих констант:

Константа	Описание
NMPWAIT_NOWAIT	Ожидание канала Pipe не выполняется. Если канал не доступен, функция CallNamedPipe сразу возвращает код ошибки
NMPWAIT_WAIT_FOREVER	Ожидание выполняется бесконечно долго

---

NMPWAIT_USE_DEFAULT_WAIT	Ожидание выполняется в течении периода времени, указанного при вызове функции CreateNamedPipe
--------------------------	---

---

В случае успешного завершения функция CallNamedPipe возвращает значение TRUE, а при ошибке - FALSE. Код ошибки можно получить, вызвав функцию GetLastError.

Заметим, что может возникнуть ситуация, при которой длина сообщения, полученного от сервера, превосходит размер буфера, предусмотренного процессом. В этом случае функция CallNamedPipe завершится с ошибкой, а функция GetLastError вернет значение ERROR\_MORE\_DATA. Не существует никакого способа получить через канал оставшуюся часть сообщения, так как перед возвращением управления функция CallNamedPipe закрывает канал с сервером.

### Функция TransactNamedPipe

Функция TransactNamedPipe, также как и функция CallNamedPipe, предназначена для выполнения передачи и приема данных от клиентского процесса серверному. Однако эта функция более гибкая в использовании, чем функция CallNamedPipe.

Прежде всего, перед использованием функции TransactNamedPipe клиентский процесс должен открыть канал с сервером, воспользовавшись для этого, например, функцией CreateFile.

Кроме того, клиентский процесс может выполнять обмен данными с сервером, вызывая функцию TransactNamedPipe много раз. При этом не будет происходить многократного открытия и закрытия канала.

Прототип функции TransactNamedPipe представлен ниже:

```
BOOL TransactNamedPipe(
    HANDLE hNamedPipe, // идентификатор канала Pipe
    LPVOID lpvWriteBuf, // адрес буфера для записи
    DWORD cbWriteBuf, // размер буфера для записи
    LPVOID lpvReadBuf, // адрес буфера для чтения
    DWORD cbReadBuf, // размер буфера для чтения
    LPDWORD lpcbRead, // адрес переменной, в которую будет
    // записано количество действительно прочитанных байт
    LPOVERLAPPED lpov); // адрес структуры типа OVERLAPPED
```

Через параметр hNamedPipe вы должны передать функции TransactNamedPipe идентификатор предварительно открытого канала. Канал следует открыть в режиме передачи сообщений.

Параметры lpvWriteBuf и cbWriteBuf задают, соответственно, адрес и размер буфера, содержимое которого будет передано через канал.

Ответное сообщение, полученное от сервера, будет записано в буфер с адресом lpvReadBuf. Размер этого буфера следует передать функции TransactNamedPipe через параметр cbReadBuf.

После того как функция TransactNamedPipe вернет управление, в переменную, адрес которой передавался через параметр lpcbRead, будет записан размер принятого от сервера сообщения в байтах.

Если операция передачи данных через канал должна выполняться с перекрытием (асинхронно), вы должны подготовить структуру типа OVERLAPPED и передать ее адрес через параметр lpov. В противном случае параметр lpov должен быть задан как NULL.

В случае успешного завершения функция TransactNamedPipe возвращает значение TRUE, а при ошибке - FALSE. Код ошибки можно получить, вызвав функцию GetLastError.

Если возникает ситуация, при которой длина сообщения, полученного от сервера, превосходит размер буфера, предусмотренного процессом, функция TransactNamedPipe завершится с ошибкой, а функция GetLastError вернет значение ERROR\_MORE\_DATA. Оставшуюся часть сообщения можно прочитать с помощью такой функции, как ReadFile.

### Функция PeekNamedPipe

Чтение данных из канала функцией ReadFile вызывает удаление прочитанных данных. В противоположность этому, функция PeekNamedPipe позволяет получить данные из именованного или анонимного канала без удаления, так что при последующих вызовах этой функции или функции ReadFile будут получены все те же данные, что и при первом вызове функции PeekNamedPipe.

Еще одно отличие заключается в том, что функция PeekNamedPipe никогда не переходит в состояние ожидания, сразу возвращая управление вне зависимости от того, есть данные в канале или нет.

Прототип функции PeekNamedPipe представлен ниже:

```
BOOL PeekNamedPipe(
    HANDLE hPipe, // идентификатор канала Pipe
    LPVOID lpvBuffer, // адрес буфера для прочитанных данных
    DWORD cbBuffer, // размер буфера прочитанных данных
    LPDWORD lpcbRead, // адрес переменной, в которую будет
    // записано количество действительно
    // прочитанных байт данных
    LPDWORD lpcbAvail, // адрес переменной, в которую будет
    // записано общее количество байт данных,
    // доступных в канале для чтения
    LPDWORD lpcbMessage); // адрес переменной, в которую будет
    // записано количество непрочитанных
    // байт в данном сообщении
```

Через параметр hPipe функции PeekNamedPipe нужно передать идентификатор открытого анонимного или именованного канала Pipe.

Данные, полученные из канала, будут записаны в буфер lpvBuffer, имеющий размер cbBuffer байт. При этом количество действительно прочитанных байт будет сохранено в переменной, адрес которой передается функции PeekNamedPipe через параметр lpcbRead.

В случае успешного завершения функция PeekNamedPipe возвращает значение TRUE, а при ошибке - FALSE. Код ошибки можно получить, вызвав функцию GetLastError.

#### Функция WaitNamedPipe

С помощью функции WaitNamedPipe процесс может выполнять ожидание момента, когда канал Pipe будет доступен для соединения:

```
BOOL WaitNamedPipe(
    LPCTSTR lpszPipeName, // адрес имени канала Pipe
    DWORD dwTimeout);    // время ожидания в миллисекундах
```

Через параметр lpszPipeName задается имя канала, для которого выполняется ожидание готовности к соединению. Время ожидания в миллисекундах задается через параметр dwTimeout.

Помимо численного значения в миллисекундах, вы можете указать в этом параметре одну из следующих констант:

Константа	Описание
NMPWAIT_WAIT_FOREVER	Ожидание выполняется бесконечно долго
NMPWAIT_USE_DEFAULT_WAIT	Ожидание выполняется в течении периода времени, указанного при вызове функции CreateNamedPipe

Если канал стал доступен до истечения периода времени, заданного параметром dwTimeout, функция WaitNamedPipe возвращает значение TRUE. В противном случае возвращается значение FALSE и вы можете воспользоваться функцией GetLastError.

#### Функция SetNamedPipeHandleState

При необходимости вы можете изменить режимы работы для уже созданного канала Pipe. Для этого предназначена функция SetNamedPipeHandleState, прототип которой мы привели ниже:

```
BOOL SetNamedPipeHandleState(
    HANDLE hNamedPipe, // идентификатор канала Pipe
    LPDWORD lpdwMode,  // адрес переменной, в которой указан
                      // новый режим канала
    LPDWORD lpcbMaxCollect, // адрес переменной, в которой
                      // указывается максимальный размер
                      // пакета, передаваемого в канал
    LPDWORD lpdwCollectDataTimeout); // адрес максимальной
                      // задержки перед передачей данных
```

Параметр hNamedPipe задает идентификатор канала Pipe, режим работы которого будет изменен.

Новый режим работы записывается в переменную, адрес которой задан через параметр lpdwMode. Вы можете указать одну из следующих констант, определяющих режим работы канала:

Константа	Использование канала
PIPE_READMODE_BYTE	Канал открывается на чтение в режиме последовательной передачи отдельных байт
PIPE_READMODE_MESSAGE	Канал открывается на чтение в режиме передачи отдельных сообщений указанной длины
PIPE_WAIT	Канал будет работать в блокирующем режиме, когда процесс переводится в состояние ожидания до завершения операций в канале
PIPE_NOWAIT	Неблокирующий режим работы канала. Если операция не может быть выполнена немедленно, в неблокирующем режиме функция завершается с ошибкой

Константы PIPE\_WAIT и PIPE\_NOWAIT, задающие блокирующий и неблокирующий режим соответственно, можно комбинировать при помощи логической операции ИЛИ с константами PIPE\_READMODE\_BYTE и PIPE\_READMODE\_MESSAGE.

Если текущий режим работы канала изменять не нужно, для параметра lpdwMode следует указать значение NULL.

Теперь рассмотрим назначение параметра lpcbMaxCollect.

Если при открытии канала клиентским процессом функцией CreateFile не была указана константа FILE\_FLAG\_WRITE\_THROUGH, то данные передаются пакетами, которые собираются из отдельных сообщений. Размер такого пакета как раз и определяет параметр lpcbMaxCollect.

В том случае, когда вы не собираетесь изменять размер пакета, укажите для параметра lpcbMaxCollect значение NULL.

Параметр lpdwCollectDataTimeout задает максимальный интервал между передачами данных по сети. Если функция SetNamedPipeHandleState изменяет параметры канала со стороны сервера, или если сервер и клиент работают на одном и том же компьютере, параметр lpdwCollectDataTimeout должен быть задан как NULL.

В случае успешного завершения функция SetNamedPipeHandleState возвращает значение TRUE, а при ошибке - FALSE. Код ошибки можно получить, вызвав функцию GetLastError.



### Функция GetNamedPipeHandleState

С помощью функции GetNamedPipeHandleState процесс может определить состояние канала Pipe, зная его идентификатор.

Прототип функции GetNamedPipeHandleState мы привели ниже:

```
BOOL GetNamedPipeHandleState(
    HANDLE hNamedPipe, // идентификатор именованного канала
    LPDWORD lpState, // адрес флагов состояния канала
    LPDWORD lpCurInstances, // адрес количества реализаций
    LPDWORD lpMaxCollectionCount, // адрес размера пакета
    // передаваемых данных
    LPDWORD lpCollectDataTimeout, // адрес максимального
    // времени ожидания
    LPTSTR lpUserName, // адрес имени пользователя
    // клиентского процесса
    DWORD nMaxUserNameSize); // размер буфера для
    // имени пользователя клиентского процесса
```

Идентификатор канала, для которого нужно определить состояние, передается функции GetNamedPipeHandleState через параметр hNamedPipe.

Через параметр lpState нужно передать указатель на переменную типа DWORD, в которую будет записан один из флагов состояния канала:

Флаги состояния	Описание
PIPE_WAIT	Канал будет работать в блокирующем режиме, когда процесс переводится в состояние ожидания до завершения операций в канале
PIPE_NOWAIT	Неблокирующий режим работы канала. Если операция не может быть выполнена немедленно, в неблокирующем режиме функция завершается с ошибкой

Если информация о состоянии канала не требуется, в качестве значения для параметра lpState следует использовать константу NULL.

В переменную, адрес которой передается через параметр lpCurInstances, записывается текущее количество реализаций канала. Если эта информация вам не нужна, передайте через параметр lpCurInstances значение NULL.

Параметры lpMaxCollectionCount и lpCollectDataTimeout позволяют определить, соответственно, размер пакета передаваемых данных и максимальное время ожидания между передачами.

Через параметр lpUserName вы должны передать указатель на буфер, в который функция GetNamedPipeHandleState запишет имя пользователя клиентского процесса. Размер этого буфера задается в параметре nMaxUserNameSize.

При необходимости вы можете задать значения параметров lpMaxCollectionCount, lpCollectDataTimeout и lpUserName как NULL. В этом случае соответствующая информация не будет извлекаться.

В случае успешного завершения функция GetNamedPipeHandleState возвращает значение TRUE, а при ошибке - FALSE. Код ошибки можно получить, вызвав функцию GetLastError.

### Функция GetNamedPipeInfo

Еще одна функция, позволяющая получить информацию об именованном канале по его идентификатору, называется GetNamedPipeInfo:

```
BOOL GetNamedPipeInfo(
    HANDLE hNamedPipe, // идентификатор канала
    LPDWORD lpFlags, // адрес флагов типа канала
    LPDWORD lpOutBufferSize, // адрес размера выходного буфера
    LPDWORD lpInBufferSize, // адрес размера входного буфера
    LPDWORD lpMaxInstances); // адрес максимального количества
    // реализаций канала
```

Параметр hNamedPipe задает идентификатор именованного канала Pipe, для которого требуется получить информацию.

Через параметр lpFlags функции GetNamedPipeInfo необходимо передать адрес переменной типа DWORD или NULL, если флаги определять не требуется. Ниже мы привели возможные значения флагов:

Флаг	Описание
PIPE_CLIENT_END	Идентификатор ссылается на клиентскую часть канала
PIPE_SERVER_END	Идентификатор ссылается на серверную часть канала
PIPE_TYPE_MESSAGE	Канал работает в режиме передачи сообщений

В переменные, адреса которых задаются через параметры lpOutBufferSize и lpInBufferSize, функция GetNamedPipeInfo заносит размеры входного и выходного буфера, соответственно. Если эта информация не нужна, передайте через параметры lpOutBufferSize и lpInBufferSize значение NULL.

И, наконец, через параметр lpMaxInstances передается адрес переменной, в которую будет записано максимальное значение реализаций, которое можно создать для данного канала. Если после вызова функции GetNamedPipeInfo в этой переменной записано значение PIPE\_UNLIMITED\_INSTANCES, количество реализаций ограничивается только свободными системными ресурсами.

В случае успешного завершения функции GetNamedPipeInfo возвращает значение TRUE, а при ошибке - FALSE. Код ошибки можно получить, вызвав функцию GetLastError.

## Примеры приложений

В этом разделе мы приведем исходные тексты двух приложений, которые передают друг другу данные через именованный канал Pipes. Заметим, что наши приложения способны установить канал связи между различными рабочими станциями через сеть.

Первое из этих приложений называется PIPES. Оно выполняет роль сервера, который получает команды от клиентского приложения PIPEC и отображает их в консольном окне (рис. 2.4).

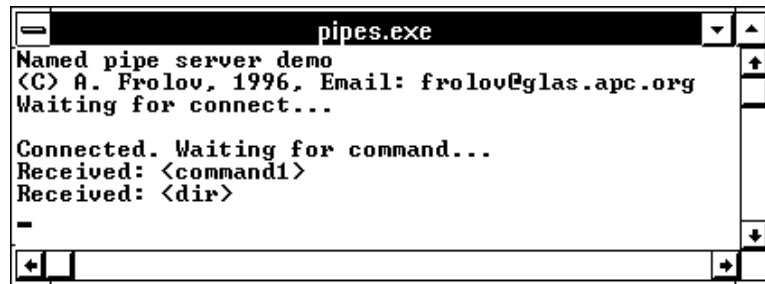


Рис. 2.4. Консольное окно серверного приложения PIPES

Сразу после запуска приложение PIPES переходит в состояние ожидания соединения с клиентским приложением. При этом в его окне отображается строка `Waiting for connect`.

Как только клиентское приложение PIPEC установит канал связи с приложением PIPES, в окно приложения PIPES выводятся строки `Connected` и `Waiting for command`. Команды, посылаемые серверу, выводятся в окне в режиме свертки. Никакой другой обработки команд не выполняется, однако принятые команды посылаются сервером обратно клиентскому приложению.

Консольное окно клиентского приложения PIPEC, запущенного в среде операционной системы Microsoft Windows 95, показано на рис. 2.5.

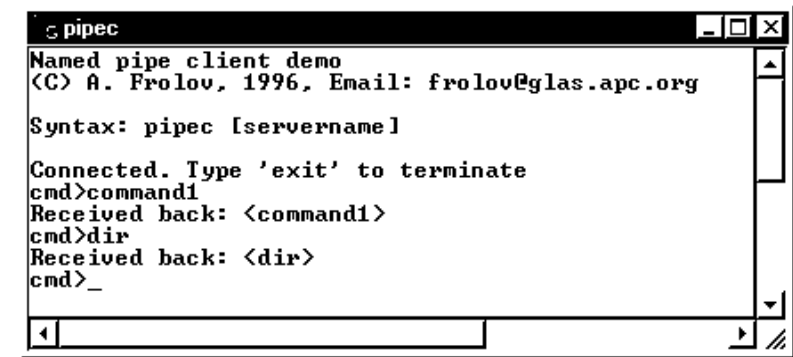


Рис. 2.5. Консольное окно клиентского приложения PIPEC

При запуске клиентского приложения PIPEC вы дополнительно можете указать параметр - имя рабочей станции, например:  
`pipec netlab`

Если имя рабочей станции не указано, приложение PIPEC будет пытаться установить канал с серверным приложением PIPES, запущенным на том же компьютере, что и PIPEC.

После успешного создания канала клиентское приложение выводит сообщение `Connected` и предлагает вводить команды в приглашении `cmd>`. В качестве команд вы можете вводить произвольные последовательности символов, кроме `exit`. Команда `exit` используется для завершения работы приложений PIPEC и PIPES.

После того как команда посылается серверу, она возвращается обратно и отображается в окне клиентского приложения для контроля в строке `Received back`.

## Приложение PIPES

Исходные тексты приложения PIPES приведены в листинге 2.11.

Листинг 2.11. Файл `pipe/pipes/pipes.c`

```
// =====
// Приложение PIPES (серверное приложение)
// Демонстрация использования каналов Pipe
// для передачи данных между процессами
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====

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

```

#include <conio.h>

int main()
{
    // Флаг успешного создания канала
    BOOL    fConnected;

    // Идентификатор канала Pipe
    HANDLE hNamedPipe;

    // Имя создаваемого канала Pipe
    LPSTR  lpszPipeName = "\\.\pipe\\$MyPipe$";

    // Буфер для передачи данных через канал
    char    szBuf[512];

    // Количество байт данных, принятых через канал
    DWORD   cbRead;

    // Количество байт данных, переданных через канал
    DWORD   cbWritten;

    printf("Named pipe server demo\n"
        "(C) A. Frolov, 1996, Email: frolov@glas.apc.org\n");

    // Создаем канал Pipe, имеющий имя lpszPipeName
    hNamedPipe = CreateNamedPipe(
        lpszPipeName,
        PIPE_ACCESS_DUPLEX,
        PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
        PIPE_UNLIMITED_INSTANCES,
        512, 512, 5000, NULL);

    // Если возникла ошибка, выводим ее код и завершаем
    // работу приложения
    if(hNamedPipe == INVALID_HANDLE_VALUE)
    {
        fprintf(stdout, "CreateNamedPipe: Error %ld\n",
            GetLastError());
        getch();
        return 0;
    }

    // Выводим сообщение о начале процесса создания канала
    fprintf(stdout, "Waiting for connect...\n");

    // Ожидаем соединения со стороны клиента
    fConnected = ConnectNamedPipe(hNamedPipe, NULL);

    // При возникновении ошибки выводим ее код
    if(!fConnected)
    {

```

```

switch(GetLastError())
{
    case ERROR_NO_DATA:
        fprintf(stdout, "ConnectNamedPipe: ERROR_NO_DATA");
        getch();
        CloseHandle(hNamedPipe);
        return 0;
        break;

    case ERROR_PIPE_CONNECTED:
        fprintf(stdout,
            "ConnectNamedPipe: ERROR_PIPE_CONNECTED");
        getch();
        CloseHandle(hNamedPipe);
        return 0;
        break;

    case ERROR_PIPE_LISTENING:
        fprintf(stdout,
            "ConnectNamedPipe: ERROR_PIPE_LISTENING");
        getch();
        CloseHandle(hNamedPipe);
        return 0;
        break;

    case ERROR_CALL_NOT_IMPLEMENTED:
        fprintf(stdout,
            "ConnectNamedPipe: ERROR_CALL_NOT_IMPLEMENTED");
        getch();
        CloseHandle(hNamedPipe);
        return 0;
        break;

    default:
        fprintf(stdout, "ConnectNamedPipe: Error %ld\n",
            GetLastError());
        getch();
        CloseHandle(hNamedPipe);
        return 0;
        break;
}

CloseHandle(hNamedPipe);
getch();
return 0;
}

// Выводим сообщение об успешном создании канала
fprintf(stdout, "\nConnected. Waiting for command...\n");

// Цикл получения команд через канал

```

```

while(1)
{
    // Получаем очередную команду через канал Pipe
    if(ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL))
    {
        // Пошляем эту команду обратно клиентскому
        // приложению
        if(!WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1,
            &cbWritten, NULL))
            break;

        // Выводим принятую команду на консоль
        printf("Received: <%s>\n", szBuf);

        // Если пришла команда "exit",
        // завершаем работу приложения
        if(!strcmp(szBuf, "exit"))
            break;
    }
    else
    {
        fprintf(stdout, "ReadFile: Error %ld\n",
            GetLastError());
        getch();
        break;
    }
}

CloseHandle(hNamedPipe);
return 0;
}

```

В области локальных переменных функции main определена строка lpszPipeName, в которой хранится имя канала:

```
LPSTR lpszPipeName = "\\\\.\\pipe\\$MyPipe$";
```

Так как канал создается локально, в качестве имени компьютера указан символ точки. Канал называется \$MyPipe\$.

Буфер szBuf размером 512 байт нужен для хранения данных, передаваемых через канал.

В переменные cbRead и cbWritten при выполнении операций чтения и записи через канал записывается, соответственно, количество принятых и переданных байт данных.

После вывода "рекламной" строки приложение PIPES создает канал, вызывая для этого функцию CreateNamedPipe:

```

hNamedPipe = CreateNamedPipe(
    lpszPipeName,
    PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
    PIPE_UNLIMITED_INSTANCES,

```

```

    512, 512, 5000, NULL);

```

В качестве первого параметра мы передаем этой функции имя канала. Во втором канале указана константа PIPE\_ACCESS\_DUPLEX, поэтому канал работает и на прием информации, и на передачу (в дуплексном режиме).

Константа PIPE\_TYPE\_MESSAGE определяет, что через канал будут передаваться сообщения заданной длины (а не просто последовательность байт данных).

Если в процессе создания канала произошла ошибка, ее код определяется с помощью функции GetLastError. Вслед за этим в консольное окно приложения выводится сообщение с кодом ошибки и приложение переводится в состояние ожидания до тех пор, пока пользователь не нажмет какую-нибудь клавишу. После этого работа приложения завершается.

После создания канала, который будет работать в блокирующем режиме, вызывается функция ConnectNamedPipe:

```
fConnected = ConnectNamedPipe(hNamedPipe, NULL);
```

Из-за блокирующего режима работы и из-за того, что канал работает без перекрытия в синхронном режиме, после вызова функции ConnectNamedPipe сервер перейдет в состояние ожидания. Он будет находиться в этом состоянии до тех пор, пока клиентское приложение PIPEC не установит с ним канал Pipe.

При возникновении ошибки наше приложение получает ее код и анализирует его, выводя в консольное окно соответствующее сообщение. Затем приложение закрывает идентификатор канала и завершает свою работу.

После успешного создания канала приложение PIPES входит в цикл получения команд от клиентского приложения PIPEC:

```

while(1)
{
    if(ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL))
    {
        if(!WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1,
            &cbWritten, NULL))
            break;
        printf("Received: <%s>\n", szBuf);
        if(!strcmp(szBuf, "exit"))
            break;
    }
    else
    {
        fprintf(stdout, "ReadFile: Error %ld\n",
            GetLastError());
        getch();
        break;
    }
}

```



```

// Получаем эту же команду обратно от сервера
if(ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL))
    printf("Received back: <%s>\n", szBuf);

// Если произошла ошибка, выводим ее код и
// завершаем работу приложения
else
{
    fprintf(stdout, "ReadFile: Error %ld\n",
        GetLastError());
    getch();
    break;
}

// В ответ на команду "exit" завершаем цикл
// обмена данными с серверным процессом
if(!strcmp(szBuf, "exit"))
    break;
}

// Закрываем идентификатор канала
CloseHandle(hNamedPipe);
return 0;
}

```

Сразу после запуска приложение PIPEC проверяет количество аргументов командной строки. Если при запуске пользователь задал в качестве параметра имя рабочей станции, то это имя вставляется внутрь строки, задающей имя канала Pipe, как это показано ниже:

```

if(argc > 1)
    sprintf(szPipeName, "\\\\"%s\\pipe\\$MyPipe$", argv[1]);

```

Если же при запуске имя рабочей станции не указано, клиентское приложение будет открывать локальный канал с локальным именем:

```

else
    strcpy(szPipeName, "\\\\.\\pipe\\$MyPipe$");

```

При использовании любого варианта для открытия канала используется функция CreateFile:

```

hNamedPipe = CreateFile(
    szPipeName, GENERIC_READ | GENERIC_WRITE,
    0, NULL, OPEN_EXISTING, 0, NULL);

```

Так как мы указали константы GENERIC\_READ и GENERIC\_WRITE, канал открывается и на чтение, и на запись.

После открытия канала запускается цикл ввода и передачи команд:

```

while(1)
{
    printf("cmd>");
    gets(szBuf);
    if(!WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1,

```

```

        &cbWritten, NULL))
        break;
    if(ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL))
        printf("Received back: <%s>\n", szBuf);
    else
    {
        fprintf(stdout, "ReadFile: Error %ld\n",
            GetLastError());
        getch();
        break;
    }
    if(!strcmp(szBuf, "exit"))
        break;
}

```

Для ввода команд используется известная вам функция gets. Введенная команда посылается серверу функцией WriteFile. Сразу после этого наше приложение вызывает функцию ReadFile чтобы получить от сервера эту же команду обратно.

В случае успеха команда сравнивается со строкой exit. Если пользователь ввел эту строку, клиентское приложение завершает свою работу. Аналогично поступает и серверное приглашение PIPES.

### Каналы передачи данных Mailslot

В завершении этой главы мы рассмотрим еще один простой способ организации передачи данных между различными процессами, основанный на использовании датаграммных каналов Mailslot.

Каналы Mailslot позволяют выполнять одностороннюю передачу данных от одного или нескольких клиентов к одному или нескольким серверам. Главная особенность каналов Mailslot заключается в том, что они, в отличие от других средств, рассмотренных нами в этой главе, позволяют передавать данные в широковещательном режиме.

Это означает, что на компьютере или в сети могут работать несколько серверных процессов, способных получать сообщения через каналы Mailslot. При этом один клиентский процесс может посылать сообщения сразу всем этим серверным процессам.

С помощью каналов Pipe вы не сможете передавать данные в широковещательном режиме, так как только два процесса могут создать канал типа Pipe.

### Создание канала Mailslot

Канал Mailslot создается серверным процессом с помощью специально предназначенной для этого функции CreateMailslot, которую мы рассмотрим немного позже. После создания серверный процесс получает идентификатор канала Mailslot. Пользуясь этим идентификатором, сервер может читать



сообщения, посылаемые в канал клиентскими процессами. Однако сервер не может выполнять над каналом Mailslot операцию записи, так как этот канал предназначен только для односторонней передачи данных - от клиента к серверу.

Приведем прототип функции CreateMailslot:

```
HANDLE CreateMailslot(
    LPCTSTR lpName,          // адрес имени канала Mailslot
    DWORD   nMaxMsgSize,     // максимальный размер сообщения
    DWORD   lReadTimeout,    // время ожидания для чтения
    LPSECURITY_ATTRIBUTES lpSecurityAttributes); // адрес
                                // структуры защиты
```

Через параметр lpName вы должны передать функции CreateMailslot адрес строки символов с именем канала Mailslot. Эта строка имеет следующий вид:

```
\\.\mailslot\[Путь]ИмяКанала
```

В этом имени путь является необязательной компонентой. Тем не менее, вы можете указать его аналогично тому, как это делается для файлов. Что же касается имени канала Mailslot, то оно задается аналогично имени канала Pipes.

Параметр nMaxMsgSize определяет максимальный размер сообщений, передаваемых через создаваемый канал Mailslot. Вы можете указать здесь нулевое значение, при этом размер сообщений не будет ограничен. Есть, однако, одно исключение - размер широковещательных сообщений, передаваемых всем рабочим станциям и серверам домена не должен превышать 400 байт.

С помощью параметра lReadTimeout серверное приложение может задать время ожидания для операции чтения в миллисекундах, по истечении которого функция чтения вернет код ошибки. Если вы укажете в этом параметре значение MAILSLT\_WAIT\_FOREVER, ожидание будет бесконечным.

Параметр lpSecurityAttributes задает адрес структуры защиты, который мы в наших приложениях будем указывать как NULL.

При ошибке функцией CreateMailslot возвращается значение INVALID\_HANDLE\_VALUE. Код ошибки можно определить при помощи функции GetLastError.

Ниже мы привели пример использования функции CreateMailslot в серверном приложении:

```
LPSTR lpszMailslotName = "\\.\mailslot\\$MailslotName$";
hMailslot = CreateMailslot(lpszMailslotName, 0,
    MAILSLT_WAIT_FOREVER, NULL);
```

В этом примере мы задали максимальный размер сообщения, поэтому на эту величину нет ограничений (кроме ограничения в 400 байт для сообщений, передаваемых всем компьютерам домена в широковещательном режиме).

Время ожидания указано как MAILSLT\_WAIT\_FOREVER, поэтому функции, работающие с данным каналом Mailslot, будут работать в блокирующем режиме.

### Открытие канала Mailslot

Прежде чем приступить к работе с каналом Mailslot, клиентский процесс должен его открыть. Для выполнения этой операции следует использовать функцию CreateFile, например, так:

```
LPSTR lpszMailslotName = "\\.\mailslot\\$MailslotName$";
hMailslot = CreateFile(
    lpszMailslotName, GENERIC_WRITE,
    FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
```

Здесь в качестве первого параметра функции CreateFile передается имя канала Mailslot. Заметим, что вы можете открыть канал Mailslot, созданный на другой рабочей станции в сети. Для этого строка имени канала, передаваемая функции CreateFile, должна иметь следующий вид:

```
\\ИмяРабочейСтанции\mailslot\[Путь]ИмяКанала
```

Можно открыть канал для передачи сообщений всем рабочим станциям заданного домена. Для этого необходимо задать имя по следующему образцу:

```
\\ИмяДомена\mailslot\[Путь]ИмяКанала
```

Для передачи сообщений одновременно всем рабочим станциям сети первичного домена имя задается следующим образом:

```
\\*\mailslot\[Путь]ИмяКанала
```

В качестве второго параметра функции CreateFile мы передаем константу GENERIC\_WRITE. Эта константа определяет, что над открываемым каналом будет выполняться операция записи. Напомним, что клиентский процесс может только посылать сообщения в канал Mailslot, но не читать их оттуда. Чтение сообщений из канала Mailslot - задача для серверного процесса.

Третий параметр указан как FILE\_SHARE\_READ, и это тоже необходимо, так как сервер может читать сообщения, посылаемые одновременно несколькими клиентскими процессами.

Обратите также внимание на константу OPEN\_EXISTING. Она используется потому, что функция CreateFile открывает существующий канал, а не создает новый.

### Запись сообщений в канал Mailslot

Запись сообщений в канал Mailslot выполняет клиентский процесс, вызывая для этого функцию WriteFile. С этой функцией вы уже имели дело:

```
HANDLE hMailslot;
char    szBuf[512];
DWORD   cbWritten;
WriteFile(hMailslot, szBuf, strlen(szBuf) + 1,
```

```
&cbWritten, NULL);
```

В качестве первого параметра этой функции необходимо передать идентификатор канала Mailslot, полученный от функции CreateFile.

Второй параметр определяет адрес буфера с сообщением, третий - размер сообщения. В нашем случае сообщения передаются в виде текстовой строки, закрытой двоичным нулем, поэтому для определения длины сообщения мы воспользовались функцией strlen.

### Чтение сообщений из канала Mailslot

Серверный процесс может читать сообщения из созданного им канала Mailslot при помощи функции ReadFile, как это показано ниже:

```
HANDLE hMailslot;
char szBuf[512];
DWORD cbRead;
ReadFile(hMailslot, szBuf, 512, &cbRead, NULL);
```

Через первый параметр функции ReadFile передается идентификатор созданного ранее канала Mailslot, полученный от функции CreateMailslot. Второй и третий параметры задают, соответственно, адрес буфера для сообщения и его размер.

Заметим, что перед выполнением операции чтения следует проверить состояние канала Mailslot. Если в нем нет сообщений, то функцию ReadFile вызывать не следует. Для проверки состояния канала вы должны воспользоваться функцией GetMailslotInfo, описанной ниже.

### Определение состояния канала Mailslot

Серверный процесс может определить текущее состояние канала Mailslot по его идентификатору с помощью функции GetMailslotInfo. Прототип этой функции мы привели ниже:

```
BOOL GetMailslotInfo(
    HANDLE hMailslot,           // идентификатор канала Mailslot
    LPDWORD lpMaxMessageSize,   // адрес максимального размера
                                // сообщения
    LPDWORD lpNextSize,         // адрес размера следующего сообщения
    LPDWORD lpMessageCount,     // адрес количества сообщений
    LPDWORD lpReadTimeout);     // адрес времени ожидания
```

Через параметр hMailslot функции передается идентификатор канала Mailslot, состояние которого необходимо определить.

Остальные параметры задаются как указатели на переменные типа DWORD, в которые будут записаны параметры состояния канала Mailslot.

В переменную, адрес которой передается через параметр lpMaxMessageSize, после возвращения из функции GetMailslotInfo будет записан максимальный размер сообщения. Вы можете использовать это

значение для динамического получения буфера памяти, в который это сообщение будет прочитано функцией ReadFile.

В переменную, адрес которой указан через параметр lpNextSize, записывается размер следующего сообщения, если оно есть в канале. Если же в канале больше нет сообщений, в эту переменную будет записана константа MAILSLOT\_NO\_MESSAGE.

С помощью параметра lpMessageCount вы можете определить количество сообщений, записанных в канал клиентскими процессами. Если это количество равно нулю, вам не следует вызывать функцию ReadFile для чтения несуществующего сообщения.

И, наконец, в переменную, адрес которой задается в параметре lpReadTimeout, записывается текущее время ожидания, установленное для канала (в миллисекундах).

Если вам не нужна вся информация, которую можно получить с помощью функции GetMailslotInfo, некоторые из ее параметров (кроме, разумеется, первого) можно указать как NULL.

В случае успешного завершения функция GetMailslotInfo возвращает значение TRUE, а при ошибке - FALSE. Код ошибки можно получить, вызвав функцию GetLastError.

Ниже мы привели пример использования функции GetMailslotInfo:

```
BOOL fReturnCode;
DWORD cbMessages;
DWORD cbMsgNumber;
fReturnCode = GetMailslotInfo(hMailslot, NULL, &cbMessages,
                              &cbMsgNumber, NULL);
```

### Изменение состояния канала Mailslot

С помощью функции SetMailslotInfo серверный процесс может изменить время ожидания для канала Mailslot уже после его создания.

Прототип функции SetMailslotInfo приведен ниже:

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,           // идентификатор канала Mailslot
    DWORD dwReadTimeout);       // время ожидания
```

Через параметр hMailslot функции SetMailslotInfo передается идентификатор канала Mailslot, для которого нужно изменить время ожидания.

Новое значение времени ожидания в миллисекундах задается через параметр dwReadTimeout. Вы также можете указать здесь константы 0 или MAILSLOT\_WAIT\_FOREVER. В первом случае функции, работающие с каналом, вернут управление немедленно, во втором - будут находиться в состоянии ожидания до тех пор, пока не завершится выполняемая операция.



## Примеры приложений

Для примера мы подготовили исходные тексты двух приложений - MSLOTS и MSLOTС, которые обмениваются информацией через канал Mailslot (рис. 2.6).

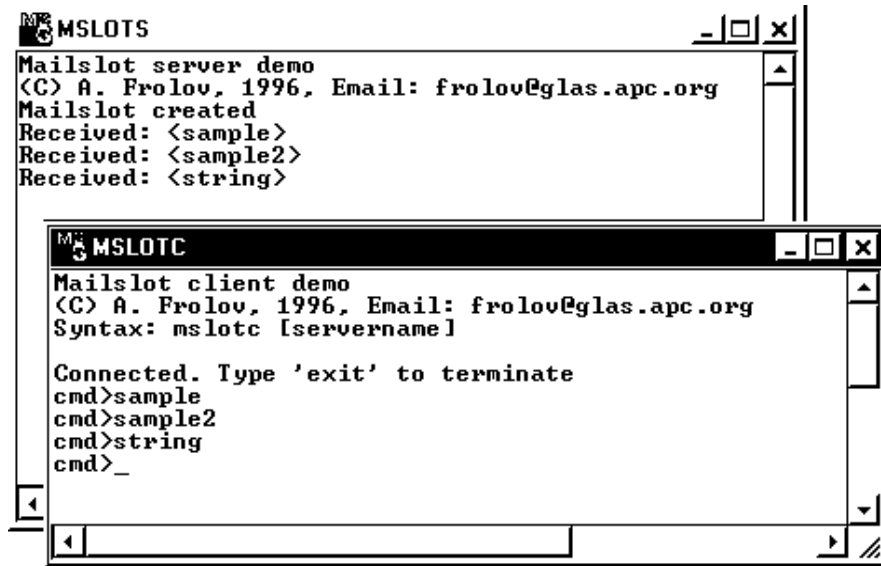


Рис. 2.6. Обмен сообщениями между приложениями MSLOTS и MSLOTС

Приложение MSLOTS выполняет роль сервера, создавая канал Mailslot. Периодически с интервалом 0,5 с это приложение проверяет, не появилось ли в канале сообщение. Если появилось, это сообщение отображается в консольном окне.

Клиентское приложение MSLOTС устанавливает связь с приложением MSLOTS. Если при запуске указать имя компьютера или домена, возможно подключение к серверу MSLOTS, запущенному на другой рабочей станции в сети.

Если вводить текстовые строки в приглашении cmd>, которое выводится в консольном окне клиентского приложения MSLOTС, они будут передаваться серверу через канал Mailslot и отображаться в его окне. Для завершения работы обоих приложений вы должны ввести в приглашении клиента команду exit.

Перейдем теперь к описанию исходных текстов наших приложений.

## Приложение MSLOTS

Исходный текст серверного приложения MSLOTS представлен в листинге 2.13.

Листинг 2.13. Файл mailslot/mslots/mslots.c

```
// =====
// Приложение MSLOTS (серверное приложение)
// Демонстрация использования каналов Mailslot
// для передачи данных между процессами
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====

#include <windows.h>
#include <stdio.h>
#include <conio.h>

int main()
{
    // Код возврата из функций
    BOOL    fReturnCode;

    // Размер сообщения в байтах
    DWORD   cbMessages;

    // Количество сообщений в канале Mailslot
    DWORD   cbMsgNumber;

    // Идентификатор канала Mailslot
    HANDLE  hMailslot;

    // Имя создаваемого канала Mailslot
    LPSTR   lpzMailslotName = "\\.\mailslot\\$Channel$";

    // Буфер для передачи данных через канал
    char    szBuf[512];

    // Количество байт данных, принятых через канал
    DWORD   cbRead;

    printf("Mailslot server demo\n"
           "(C) A. Frolov, 1996, Email: frolov@glas.apc.org\n");

    // Создаем канал Mailslot, имеющий имя lpzMailslotName
    hMailslot = CreateMailslot(
        lpzMailslotName, 0,
        MAILSLT_WAIT_FOREVER, NULL);

    // Если возникла ошибка, выводим ее код и завершаем
    // работу приложения
```

```

if(hMailslot == INVALID_HANDLE_VALUE)
{
    fprintf(stdout, "CreateMailslot: Error %ld\n",
        GetLastError());
    getch();
    return 0;
}

// Выводим сообщение о создании канала
fprintf(stdout, "Mailslot created\n");

// Цикл получения команд через канал
while(1)
{
    // Определяем состояние канала Mailslot
    fReturnCode = GetMailslotInfo(
        hMailslot, NULL, &cbMessages,
        &cbMsgNumber, NULL);

    if(!fReturnCode)
    {
        fprintf(stdout, "GetMailslotInfo: Error %ld\n",
            GetLastError());
        getch();
        break;
    }

    // Если в канале есть Mailslot сообщения,
    // читаем первое из них и выводим на экран
    if(cbMsgNumber != 0)
    {
        if(ReadFile(hMailslot, szBuf, 512, &cbRead, NULL))
        {
            // Выводим принятую строку на консоль
            printf("Received: <%s>\n", szBuf);

            // Если пришла команда "exit",
            // завершаем работу приложения
            if(!strcmp(szBuf, "exit"))
                break;
        }
        else
        {
            fprintf(stdout, "ReadFile: Error %ld\n",
                GetLastError());
            getch();
            break;
        }
    }

    // Выполняем задержку на 500 миллисекунд
    Sleep(500);
}

```

```

}

// Перед завершением приложения закрываем
// идентификатор канала Mailslot
CloseHandle(hMailslot);
return 0;
}

```

Прежде всего, серверное приложение создает канал Mailslot, пользуясь для этого функцией CreateMailslot:

```

hMailslot = CreateMailslot(lpszMailslotName, 0,
    MAILSLT_WAIT_FOREVER, NULL);

```

Далее запускается цикл, в котором после определения состояния канала выполняется чтение сообщений из него (при условии, что в канале есть сообщения). Для проверки состояния канала мы используем функцию GetMailslotInfo.

Сообщение читается функцией ReadFile:

```

ReadFile(hMailslot, szBuf, 512, &cbRead, NULL);

```

После чтения перед выполнением очередной проверки состояния приложение выполняет задержку, вызывая для этого функцию Sleep:

```

Sleep(500);

```

Задержка необходима для того, чтобы ожидание сообщения в цикле не отнимало слишком много системных ресурсов у других приложений.

Перед завершением работы приложения мы закрываем идентификатор канала Mailslot с помощью функции CloseHandle:

```

CloseHandle(hMailslot);

```

### Приложение MSLOTС

Исходный текст клиентского приложения MSLOTС представлен в листинге 2.14.

Листинг 2.14. Файл mailslot/mslotc/mslotc.c

```

// =====
// Приложение MSLOTС (клиентское приложение)
// Демонстрация использования каналов Mailslot
// для передачи данных между процессами
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====

#include <windows.h>
#include <stdio.h>
#include <conio.h>

DWORD main(int argc, char *argv[])
{

```

```

// Идентификатор канала Mailslot
HANDLE hMailslot;

// Буфер для имени канала Mailslot
char szMailslotName[256];

// Буфер для передачи данных через канал
char szBuf[512];

// Количество байт, переданных через канал
DWORD cbWritten;

printf("Mailslot client demo\n"
      "(C) A. Frolov, 1996, Email: frolov@glas.apc.org\n");

printf("Syntax: mslotc [servername]\n");

// Если при запуске было указано имя сервера,
// указываем его в имени канала Mailslot
if(argc > 1)
    sprintf(szMailslotName, "\\\\"s\\mailslot\\$Channel$",
            argv[1]);

// Если имя сервера задано не было, создаем канал
// с локальным процессом
else
    strcpy(szMailslotName, "\\\\.\\mailslot\\$Channel$");

// Создаем канал с процессом MSLOTS
hMailslot = CreateFile(
    szMailslotName, GENERIC_WRITE,
    FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);

// Если возникла ошибка, выводим ее код и
// завершаем работу приложения
if(hMailslot == INVALID_HANDLE_VALUE)
{
    fprintf(stdout, "CreateFile: Error %ld\n",
            GetLastError());
    getch();
    return 0;
}

// Выводим сообщение о создании канала
fprintf(stdout, "\nConnected. Type 'exit' to terminate\n");

// Цикл отправки команд через канал
while(1)
{
    // Выводим приглашение для ввода команды
    printf("cmd>");

```

```

// Вводим текстовую строку
gets(szBuf);

// Передаем введенную строку серверному процессу
// в качестве команды
if(!WriteFile(hMailslot, szBuf, strlen(szBuf) + 1,
    &cbWritten, NULL))
    break;

// В ответ на команду "exit" завершаем цикл
// обмена данными с серверным процессом
if(!strcmp(szBuf, "exit"))
    break;
}

// Закрываем идентификатор канала
CloseHandle(hMailslot);
return 0;
}

```

Сразу после запуска приложение проверяет параметры. Если вы указали имя компьютера или домена, оно будет вставлено в строку, передаваемую функции `CreateFile`, открывающей канал Mailslot.

Канал открывается следующим образом:

```

hMailslot = CreateFile( szMailslotName, GENERIC_WRITE,
    FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);

```

В результате будет открыт канал, предназначенный для записи в синхронном режиме.

Ввод и передача текстовых строк через канал выполняется в цикле, не имеющем никаких особенностей. Для записи введенной строки в канал мы вызываем функцию `WriteFile`:

```

WriteFile(hMailslot, szBuf, strlen(szBuf) + 1,
    &cbWritten, NULL);

```

Перед завершением своей работы приложение MSLOTС закрывает канал, вызывая для этого функцию `CloseHandle`.

## 3 БИБЛИОТЕКИ ДИНАМИЧЕСКОЙ КОМПОНОВКИ

Библиотеки динамической компоновки DLL (Dynamic Link Libraries) являются стержневым компонентом операционной системы Windows NT и многих приложений Windows. Без преувеличения можно сказать, что вся операционная система Windows, все ее драйверы, а также другие расширения есть ни что иное, как набор библиотек динамической компоновки. Редкое крупное приложение Windows не имеет собственных библиотек динамической компоновки, и ни одно приложение не может обойтись без вызова функций, расположенных в таких библиотеках. В частности, все функции программного интерфейса Windows NT находятся именно в библиотеках динамической компоновки DLL.

В 13 томе “Библиотеки системного программиста”, который называется “Операционная система Microsoft Windows 3.1 для программиста. Часть третья” мы уже рассказывали о создании и использовании библиотек DLL в среде операционной системы Microsoft Windows версии 3.1. Что же касается операционных систем Microsoft Windows NT и Microsoft Windows 95, то в них библиотеки DLL создаются и работают по-другому.

### Статическая и динамическая компоновка

Прежде чем приступить к изучению особенностей использования библиотек динамической компоновки в операционной системе Microsoft Windows NT, напомним кратко, чем отличаются друг от друга статическая и динамическая компоновка.

При использовании статической компоновки вы готовили исходный текст приложения, затем транслировали его для получения объектного модуля. После этого редактор связей компоновал объектные модули, полученные в результате трансляции исходных текстов и модули из библиотек объектных модулей, в один исполнимый ехе-файл. В процессе запуска файл программы загружался полностью в оперативную память и ему передавалось управление.

Таким образом, при использовании статической компоновки редактор связей записывает в файл программы все модули, необходимые для работы. В любой момент времени в оперативной памяти компьютера находится весь код, необходимый для работы запущенной программы.

В среде мультзадачной операционной системы статическая компоновка неэффективна, так как приводит к неэкономному использованию очень дефицитного ресурса - оперативной памяти. Представьте себе, что в системе одновременно работают 5 приложений, и все они вызывают такие функции, как `sprintf`, `memcpy`, `strcmp` и т. д. Если приложения были собраны с использованием статической компоновки, в памяти будут находиться одновременно 5 копий функции `sprintf`, 5 копий функции `memcpy`, и т. д. (рис. 3.1).

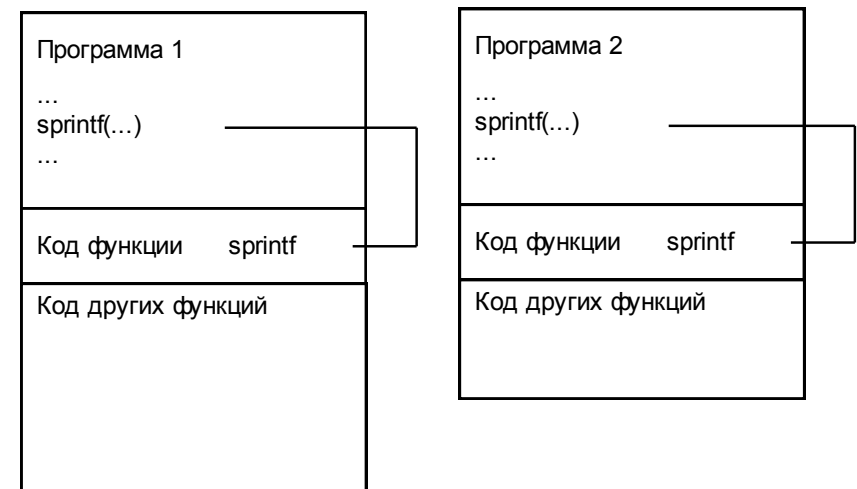


Рис. 3.1. Вызов функций при использовании статической компоновки

Очевидно, использование оперативной памяти было бы намного эффективнее, если бы в памяти находилось только по одной копии функций, а все работающие параллельно программы могли бы их вызывать.

Практически в любой многозадачной операционной системе для любого компьютера используется именно такой способ обращения к функциям, нужным одновременно большому количеству работающих параллельно программ.

При использовании динамической компоновки загрузочный код нескольких (или нескольких десятков) функций объединяется в отдельные файлы, загружаемые в оперативную память в единственном экземпляре. Программы, работающие параллельно, вызывают функции, загруженные в память из файлов библиотек динамической компоновки, а не из файлов программ.

Таким образом, используя механизм динамической компоновки, в загрузочном файле программы можно расположить только те функции,

которые являются специфическими для данной программы. Те же функции, которые нужны всем (или многим) программам, работающим параллельно, можно вынести в отдельные файлы - библиотеки динамической компоновки, и хранить в памяти в единственном экземпляре (рис. 3.2). Эти файлы можно загружать в память только при необходимости, например, когда какая-нибудь программа захочет вызвать функцию, код которой расположен в библиотеке.

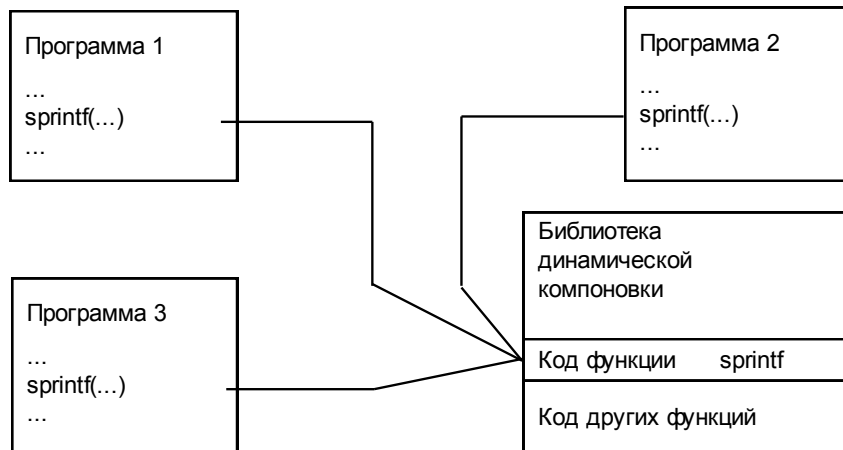


Рис. 3.2. Вызов функции при использовании динамической компоновки

В операционной системе Windows NT файлы библиотек динамической компоновки имеют расширение имени `dll`, хотя можно использовать любое другое, например, `exe`. В первых версиях Windows DLL-библиотеки располагались в файлах с расширением имени `exe`. Возможно поэтому файлы `kernel286.exe`, `kernel386.exe`, `gdi.exe` и `user.exe` имели расширение имени `exe`, а не `dll`, несмотря на то, что перечисленные выше файлы, составляющие ядро операционной системы Windows версии 3.1, есть ни что иное, как DLL-библиотеки. Наиболее важные компоненты операционной системы Microsoft Windows NT расположены в библиотеках с именами `kernel32.dll` (ядро операционной системы), `user32.dll` (функции пользовательского интерфейса), `gdi32.dll` (функции для рисования изображений и текста).

Механизм динамической компоновки был изобретен задолго до появления операционных систем Windows и OS/2 (которая также активно использует механизм динамической компоновки). Например, в мультизадачных многопользовательских операционных системах VS1, VS2, MVS, VM, созданных для компьютеров IBM-370 и аналогичных, код функций, нужных параллельно работающим программам, располагается в отдельных

библиотеках и может загружаться при необходимости в специально выделенную общую область памяти.

### DLL-библиотеки в операционной системе Windows NT

В операционной системе Microsoft Windows версии 3.1 после загрузки DLL-библиотека становилась как бы частью операционной системы. DLL-библиотека является модулем и находится в памяти в единственном экземпляре, содержит сегменты кода и ресурсы, а так же один сегмент данных (рис. 3.3). Можно сказать, что для DLL-библиотеки создается одна копия (instance), состоящая только из сегмента данных, и один модуль, состоящий из кода и ресурсов.

DLL-библиотека



Рис. 3.3. Структура DLL-библиотеки в памяти

DLL-библиотека, в отличие от приложения, не имеет стека и очереди сообщения. Функции, расположенные в модуле DLL-библиотеки, выполняются в контексте вызвавшей их задачи. При этом они пользуются стеком копии приложения, так как собственного стека в DLL-библиотеке не предусмотрено. Тем не менее, в среде операционной системы Microsoft Windows версии 3.1 функции, расположенные в 16-разрядной DLL-библиотеке, пользуются сегментом данных, принадлежащей этой библиотеке, а не копии приложения.

Создавая приложения для операционной системы Microsoft Windows версии 3.1, вы делали DLL-библиотеки для коллективного использования ресурсов или данных, расположенных в сегменте данных библиотеки. Функции, входящие в состав 16-разрядной DLL-библиотеки, могут заказывать блоки памяти с атрибутом `GMEM_SHARE`. Такой блок памяти не принадлежит ни одному приложению и поэтому не освобождается автоматически при завершении работы приложения. Так как в Windows версии 3.1 все приложения используют общую глобальную память, блоки памяти с атрибутом `GMEM_SHARE` можно использовать для обмена

данными между приложениями. Управлять таким обменом могут, например, функции, расположенные в соответствующей DLL-библиотеке.

Когда разные приложения, запущенные в среде операционной системы Microsoft Windows версии 3.1, обращались к одной и той же функции DLL-библиотеки, то они все использовали для этого один и тот же адрес. Это и понятно - так как все приложения работали на одной виртуальной машине, то все они находились в одном адресном пространстве.

В операционной системе Microsoft Windows NT каждое приложение работает в рамках отдельного адресного пространства. Поэтому для того чтобы приложение могло вызывать функции из DLL-библиотеки, эта библиотека должна находиться в адресном пространстве приложения.

Здесь мы обращаем ваше внимание на первое отличие механизма динамической компоновки в среде Microsoft Windows NT от аналогичного механизма для Microsoft Windows версии 3.1.

Отображение страниц DLL-библиотеки

В среде Microsoft Windows NT DLL-библиотека загружается в страницы виртуальной памяти, которые отображаются в адресные пространства всех "заинтересованных" приложений, которым нужны функции из этой библиотеки. При этом используется механизм, аналогичный отображению файлов на память, рассмотренный в первой главе нашей книги.

На рис. 3.4 схематически показано отображение кода и данных DLL-библиотеки в адресные пространства двух приложений.

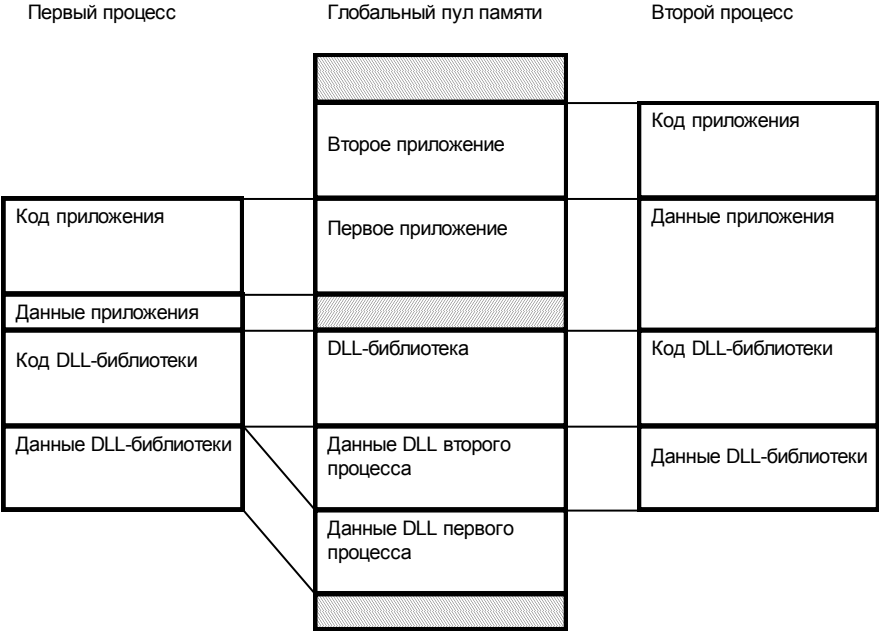


Рис. 3.4. Отображение DLL-библиотеки в адресные пространства двух процессов

На этом рисунке показано, что в глобальном пуле памяти находится один экземпляр кода DLL-библиотеки, который отображается в адресные пространства приложений (процессов). Что же касается данных DLL-библиотеки, то для каждого приложения в глобальном пуле создается отдельная область. Таким образом, различные приложения не могут в этом случае передавать друг другу данные через общую область данных DLL-библиотеки, как это можно было делать в среде операционной системы Microsoft Windows версии 3.1.

Тем не менее, принципиальная возможность создания глобальных областей памяти DLL-библиотеки, доступных разным процессам, существует. Для этого необходимо при редактировании описать область данных DLL-библиотеки как SHARED.

Заметим, что одна и та же функция DLL-библиотеки может отображаться на разные адреса в различные адресные пространства приложений. Это же относится к глобальным и статическим переменным DLL-библиотеки - они будут отображаться на разные адреса для различных приложений.

Когда первое приложение загрузит DLL-библиотеку в память (явно или неявно), эта библиотека (точнее говоря, страницы памяти, в которые она



загружена) будет отображена в адресное пространство этого приложения. Если теперь другое приложение попытается загрузить ту же самую библиотеку еще раз, то для него будет создано новое отображение тех же самых страниц. На этот раз страницы могут быть отображены уже на другие адреса.

Кроме того, для каждой DLL-библиотеки система ведет счетчик использования (usage count). Содержимое этого счетчика увеличивается при очередной загрузке библиотеки в память и уменьшается при освобождении библиотеки.

Когда содержимое счетчика использования DLL-библиотеки станет равным нулю, библиотека будет выгружена из памяти.

### Обмен данными между приложениями через DLL-библиотеку

В среде операционной системы Microsoft Windows версии 3.1 существовала возможность обмена данными между различными работающими параллельно приложениями через область локальной кучи (local heap) DLL-библиотеки. Эта область находилась в сегменте данных DLL-библиотеки и потому была доступна всем приложениям.

Что же касается операционной системы Microsoft Windows NT, то в ее среде DLL-библиотеки не имеют собственных областей данных, а отображаются в адресные пространства приложений, загружающих эти библиотеки. Как результат, приложения не могут получать адреса статических и глобальных переменных и использовать эти переменные для обмена данными - ведь адрес, верный в контексте одного приложения, не будет иметь никакого смысла для другого приложения.

Приложение также может сделать попытку изменить содержимое статической или глобальной переменной, с тем чтобы другие приложения могли прочитать новое значение. Однако этот способ передачи данных между приложениями не будет работать. Попытка изменения данных будет зафиксирована операционной системой, которая создаст для этого приложения копию страницы памяти, в которой находится изменившиеся данные, с использованием механизма копирования при записи (copy-on-write). Мы рассказывали об этом механизме в предыдущем томе "Библиотеки системного программиста".

Если по какой-либо причине вы не можете использовать для обмена данными между приложениями файлы, отображаемые на память, можно создать DLL-библиотеку с данными, имеющими атрибут SHARED.

Для этого прежде всего вы должны задать имя секции данных, которая будет использована для передачи данных между процессами. Это можно сделать с помощью прагмы транслятора data\_seg:

```
#pragma data_seg (".shar")
```

После этого в файле определения модуля для DLL-библиотеки необходимо указать атрибуты секции данных, как это показано ниже:

```
SECTIONS .shar READ WRITE SHARED
```

Можно также указать ключ редактору связей:  
-SECTION: .shar,RWS

Строка RWS определяет атрибуты секции: R - READ, W - WRITE, S - SHARED.

При обращении к глобальным переменным, расположенным в секции с атрибутом SHARED, процессы должны выполнять взаимную синхронизацию с использованием таких средств, как критические секции, объекты-события, семафоры.

### Как работает DLL-библиотека

В операционной системе Microsoft Windows версии 3.1 16-разрядная DLL-библиотека состоит из нескольких специфических функций и произвольного набора функций, выполняющих ту работу, для которой разрабатывалась данная библиотека. В заголовке загрузочного модуля DLL-библиотеки описаны экспортируемые точки входа, соответствующие всем или некоторым определенным в ней функциям. Приложения могут вызывать только те функции DLL-библиотеки, которые ей экспортируются.

В процессе инициализации после загрузки 16-разрядной DLL-библиотеки в память Windows версии 3.1 вызывает функцию LibEntry, которая должна быть определена в каждой DLL-библиотеке. Задачей функции LibEntry является инициализация локальной области памяти, если она определена для DLL-библиотеки.

Функция LibEntry должна быть дальней функцией, составленной на языке ассемблера, так как она получает параметры через регистры процессора. Мы подробно описали функцию LibEntry и ее параметры в 13 томе "Библиотеки системного программиста". Заметим, что использование языка ассемблера затрудняет создание мультиплатформных приложений, поэтому в мультиплатформной операционной системе Microsoft Windows NT используется другой способ инициализации.

Создавая 16-разрядную DLL-библиотеку, вам не надо определять функцию LibEntry самостоятельно, так как при создании файла DLL-библиотеки редактор связей, входящий в систему разработки, включает уже имеющийся в стандартной библиотеке модуль. Этот стандартный модуль выполняет всю необходимую работу по инициализации локальной области памяти DLL-библиотеки (с помощью функции LocalInit) и затем вызывает функцию LibMain.

Функция LibMain должна присутствовать в каждой 16-разрядной DLL-библиотеке. Эту функцию надо определить самостоятельно, причем вы можете воспользоваться языком программирования C.

По своему назначению функция LibMain напоминает функцию WinMain обычного приложения Windows. Функция WinMain получает управление при

запуске приложения, а функция LibMain - при первой загрузке DLL-библиотеки в память. Так же как и функция WinMain, функция LibMain имеет параметры, которые можно использовать для инициализации библиотеки.

Прототип функции LibMain и ее параметры были описаны в 13 томе “Библиотеки системного программиста”.

Другая функция, которая присутствует в каждой 16-разрядной DLL-библиотеке, это функция WEP.

В среде Microsoft Windows версии 3.1 DLL-библиотека в любой момент времени может быть выгружена из памяти. В этом случае Windows перед выгрузкой вызывает функцию WEP. Эта функция, как и функция LibMain, вызывается только один раз. Она может быть использована для уничтожения структур данных и освобождения блоков памяти, заказанных при инициализации DLL-библиотеки.

Вам не обязательно самостоятельно определять функцию WEP. Так же как и функция LibEntry, функция WEP добавляется в 16-разрядную DLL-библиотеку транслятором.

Что же касается 32-разрядных DLL-библиотек операционной системы Microsoft Windows NT, то их инициализация и выгрузка из памяти происходит иначе.

### Инициализация DLL-библиотеки в среде Microsoft Windows NT

В 32-разрядных DLL-библиотеках операционной системы Microsoft Windows NT вместо функций LibMain и WEP используется одна функция DLLEntryPoint, которая выполняет все необходимые задачи по инициализации библиотеки и при необходимости освобождает заказанные ранее ресурсы (имя функции инициализации может быть любым).

Функции LibMain и WEP вызываются только один раз при загрузке библиотеки в память и при ее выгрузке. В отличие от них, функция DLLEntryPoint вызывается всякий раз, когда выполняется инициализация процесса или задачи, обращающихся к функциям библиотеки, а также при явной загрузке и выгрузке библиотеки функциями LoadLibrary и FreeLibrary.

Ниже мы привели прототип функции DLLEntryPoint:

```
BOOL WINAPI DllEntryPoint(
    HINSTANCE hinstDLL, // идентификатор модуля DLL-библиотеки
    DWORD     fdwReason, // код причины вызова функции
    LPVOID     lpvReserved); // зарезервировано
```

Через параметр hinstDLL функции DLLEntryPoint передается идентификатор модуля DLL-библиотеки, который можно использовать при обращении к ресурсам, расположенным в файле этой библиотеки.

Что же касается параметра fdwReason, то он зависит от причины, по которой произошел вызов функции DLLEntryPoint. Этот параметр может принимать следующие значения:

Значение	Описание
DLL_PROCESS_ATTACH	Библиотека отображается в адресное пространство процесса в результате запуска процесса или вызова функции LoadLibrary
DLL_THREAD_ATTACH	Текущий процесс создал новую задачу, после чего система вызывает функции DLLEntryPoint всех DLL-библиотек, подключенных к процессу
DLL_THREAD_DETACH	Этот код причины передается функции DLLEntryPoint, когда задача завершает свою работу нормальным (не аварийным) способом
DLL_PROCESS_DETACH	Отображение DLL-библиотеки в адресное пространство отменяется в результате нормального завершения процесса или вызова функции FreeLibrary

Параметр lpvReserved зарезервирован. В SDK, тем не менее, сказано, что значение параметра lpvReserved равно NULL во всех случаях, кроме двух следующих:

- когда параметр fdwReason равен DLL\_PROCESS\_ATTACH и используется статическая загрузка DLL-библиотеки;
- когда параметр fdwReason равен DLL\_PROCESS\_DETACH и функция DLLEntryPoint вызвана в результате завершения процесса, а не вызова функции FreeLibrary

В процессе инициализации функция DLLEntryPoint может отменить загрузку DLL-библиотеки. Если код причины вызова равен DLL\_PROCESS\_ATTACH, функция DLLEntryPoint отменяет загрузку библиотеки, возвращая значение FALSE. Если же инициализация выполнена успешно, функция должна вернуть значение TRUE.

В том случае, когда приложение пыталось загрузить DLL-библиотеку функцией LoadLibrary, а функция DLLEntryPoint отменила загрузку, функция LoadLibrary возвратит значение NULL. Если же приложение выполняет инициализацию DLL-библиотеки неявно, при отмене загрузки библиотеки приложение также не будет загружено для выполнения.

Приведем пример функции инициализации DLL-библиотеки:

```
BOOL WINAPI DllEntryPoint(
    HMODULE hModule, // идентификатор модуля
    DWORD   fdwReason, // причина вызова функции DLLEntryPoint
    LPVOID   lpvReserved) // зарезервировано
{
    switch(fdwReason)
    {
        // Подключение нового процесса
        case DLL_PROCESS_ATTACH:
```



```

{
    // Обработка подключения процесса
    . . .
    break;
}

// Подключение новой задачи
case DLL_THREAD_ATTACH:
{
    // Обработка подключения новой задачи
    . . .
    break;
}

// Отключение процесса
case DLL_PROCESS_DETACH:
{
    // Обработка отключения процесса
    . . .
    break;
}

// Отключение задачи
case DLL_THREAD_DETACH:
{
    // Обработка отключения задачи
    . . .
    break;
}
}
return TRUE;
}

```

### Экспортирование функций и глобальных переменных

Хотя существуют DLL-библиотеки без исполнимого кода и предназначенные для хранения ресурсов, подавляющее большинство DLL-библиотек экспортируют функции для их совместного использования несколькими приложениями.

Кроме функций LibMain и WEP в 16-разрядных DLL-библиотеках операционной системы Microsoft Windows версии 3.1 и функции DLLEntryPoint в 32-разрядных библиотеках операционных систем Microsoft Windows NT и Microsoft Windows 95 могут быть определены *экспортируемые* и *неэкспортируемые* функции.

Экспортируемые функции доступны для вызова приложениям Windows. Неэкспортируемые являются локальными для DLL-библиотеки, они доступны только для функций библиотеки.

При необходимости вы можете экспортировать из 32-разрядных DLL-библиотек не только функции, но и глобальные переменные.

Самый простой способ сделать функцию экспортируемой - перечислить все экспортируемые функции в файле определения модуля при помощи оператора EXPORTS:

```

EXPORTS
    ИмяТочкиВхода [=ВнутрИмя] [@Номер] [NONAME] [CONSTANT]
    . . .

```

Здесь ИмяТочкиВхода задает имя, под которым экспортируемая из DLL-библиотеки функция будет доступна для вызова.

Внутри DLL-библиотеки эта функция может иметь другое имя. В этом случае необходимо указать ее внутреннее имя ВнутрИмя.

С помощью параметра @Номер вы можете задать порядковый номер экспортируемой функции.

Если вы не укажете порядковые номера экспортируемых функций, при компоновке загрузочного файла DLL-библиотеки редактор связи создаст свою собственную нумерацию, которая может изменяться при внесении изменений в исходные тексты функций и последующей повторной компоновке.

Заметим, что ссылка на экспортируемую функцию может выполняться двумя различными способами - по имени функции и по ее порядковому номеру. Если функция вызывается по имени, ее порядковый номер не имеет значения. Однако вызов функции по порядковому номеру выполняется быстрее, поэтому использование порядковых номеров предпочтительнее.

Если при помощи файла определения модуля DLL-библиотеки вы задаете фиксированное распределение порядковых номеров экспортируемых функций, при внесении изменений в исходные тексты DLL-библиотеки это распределение не изменится. В этом случае все приложения, ссылающиеся на функции из этой библиотеки по их порядковым номерам, будут работать правильно. Если же вы не определили порядковые номера функций, у приложений могут возникнуть проблемы с правильной адресацией функции из-за возможного изменения этих номеров.

Указав флаг NONAME и порядковый номер, вы сделаете имя экспортируемой функции невидимым. При этом экспортируемую функцию можно будет вызвать только по порядковому номеру, так как имя такой функции не попадет в таблицу экспортируемых имен DLL-библиотеки.

Флаг CONSTANT позволяет экспортировать из DLL-библиотеки не только функции, но и данные. При этом параметр ИмяТочкиВхода задает имя экспортируемой глобальной переменной, определенной в DLL-библиотеке.

Приведем пример экспортирования функций и глобальных переменных из DLL-библиотеки:

```

EXPORTS
    DrawBitmap=MyDraw @4
    ShowAll
    HideAll
    MyPoolPtr @5 CONSTANT
    GetMyPool @8 NONAME

```

### FreeMyPool @9 NONAME

В приведенном выше примере в разделе EXPORTS перечислены имена нескольких экспортируемых функций DrawBitmap, ShowAll, HideAll, GetMyPool, FreeMyPool и глобальной переменной MyPoolPtr.

Функция MyDraw, определенная в DLL-библиотеке, экспортируется под именем DrawBitmap. Она также доступна под номером 4.

Функции ShowAll и HideAll экспортируются под своими “настоящими” именами, с которыми они определены в DLL-библиотеке. Для них не заданы порядковые номера.

Функции GetMyPool и FreeMyPool экспортируются с флагом NONAME, поэтому к ним можно обращаться только по их порядковым номерам, которые равны, соответственно, 8 и 9.

Имя MyPoolPtr экспортируется с флагом CONSTANT, поэтому оно является именем глобальной переменной, определенной в DLL-библиотеке, и доступной для приложений, загружающих эту библиотеку.

### Импортирование функций

Когда вы используете статическую компоновку, то включаете в файл проекта приложения соответствующий lib-файл, содержащий нужную вам библиотеку объектных модулей. Такая библиотека содержит исполняемый код модулей, который на этапе статической компоновки включается в ехе-файл загрузочного модуля.

Если используется динамическая компоновка, в загрузочный ехе-файл приложения записывается не исполнимый код функций, а ссылка на соответствующую DLL-библиотеку и функцию внутри нее. Как мы уже говорили, эта ссылка может быть организована с использованием либо имени функции, либо ее порядкового номера в DLL-библиотеке.

Откуда при компоновке приложения редактор связей узнает имя DLL-библиотеки, имя или порядковый номер экспортируемой функции? Для динамической компоновки функции из DLL-библиотеки можно использовать различные способы.

### Библиотека импорта

Для того чтобы редактор связей мог создать ссылку, в файл проекта приложения вы должны включить так называемую *библиотеку импорта* (import library). Эта библиотека создается автоматически системой разработки Microsoft Visual C++.

Следует заметить, что стандартные библиотеки систем разработки приложений Windows содержат как обычные объектные модули, предназначенные для статической компоновки, так и ссылки на различные стандартные DLL-библиотеки, экспортирующие функции программного интерфейса операционной системы Windows.

### Динамический импорт функций во время выполнения приложения

В некоторых случаях невозможно выполнить динамическую компоновку на этапе редактирования. Вы можете, например, создать приложение, которое состоит из основного модуля и дополнительных, реализованных в виде DLL-библиотек. Состав этих дополнительных модулей и имена файлов, содержащих DLL-библиотеки, может изменяться, при этом в приложение могут добавляться новые возможности.

Если вы, например, разрабатываете систему распознавания речи, то можете сделать ее в виде основного приложения и набора DLL-библиотек, по одной библиотеке для каждого национального языка. В продажу система может поступить в комплекте с одной или двумя библиотеками, но в дальнейшем пользователь сможет купить дополнительные библиотеки и, просто переписав новые библиотеки на диск, получить возможность работы с другими языками. При этом основной модуль приложения не может “знать” заранее имена файлов дополнительных DLL-библиотек, поэтому статическая компоновка с использованием библиотеки импорта невозможна.

Однако приложение может в любой момент времени загрузить любую DLL-библиотеку, вызвав специально предназначенную для этого функцию программного интерфейса Windows с именем LoadLibrary. Приведем ее прототип:

```
HINSTANCE WINAPI LoadLibrary(LPCSTR lpzLibFileName);
```

Параметр функции является указателем на текстовую строку, закрытую двоичным нулем. В эту строку перед вызовом функции следует записать путь к файлу DLL-библиотеки или имя этого файла. Если путь к файлу не указан, при поиске выполняется последовательный просмотр следующих каталогов:

- каталог, из которого запущено приложение;
- текущий каталог;
- 32-разрядный системный каталог Microsoft Windows NT;
- 16-разрядный системный каталог;
- каталог в котором находится операционная система Windows NT;
- каталоги, перечисленные в переменной описания среды PATH

Если файл DLL-библиотеки найден, функция LoadLibrary возвращает идентификатор модуля библиотеки. В противном случае возвращается значение NULL. При этом код ошибки можно получить при помощи функции GetLastError.

Функция LoadLibrary может быть вызвана разными приложениями для одной и той же DLL-библиотеки несколько раз. В этом случае в среде операционной системы Microsoft Windows версии 3.1 загрузка DLL-библиотеки выполняется только один раз. Последующие вызовы функции LoadLibrary приводят только к увеличению счетчика использования DLL-библиотеки. Что же касается Microsoft Windows NT, то при многократном

вызове функции LoadLibrary различными процессами функция инициализации DLL-библиотеки получает несколько раз управление с кодом причины вызова, равным значению DLL\_PROCESS\_ATTACH.

В качестве примера приведем фрагмент исходного текста приложения, загружающего DLL-библиотеку из файла DLLDEMO.DLL:

```
typedef HWND (WINAPI *MYDLLPROC) (LPSTR);
MYDLLPROC GetAppWindow;
HANDLE hDLL;

hDLL = LoadLibrary("DLLDEMO.DLL");
if(hDLL != NULL)
{
    GetAppWindow = (MYDLLPROC)GetProcAddress(hDLL,
        "FindApplicationWindow");
    if(GetAppWindow != NULL)
    {
        if(GetAppWindow(szWindowTitle) != NULL)
            MessageBox(NULL, "Application window was found",
                szAppTitle, MB_OK | MB_ICONINFORMATION);
        else
            MessageBox(NULL, "Application window was not found",
                szAppTitle, MB_OK | MB_ICONINFORMATION);
    }
    FreeLibrary(hDLL);
}
```

Здесь вначале с помощью функции LoadLibrary выполняется попытка загрузки DLL-библиотеки DLLDEMO.DLL. В случае успеха приложение получает адрес точки входа для функции с именем FindApplicationWindow, для чего используется функция GetProcAddress. Этой функцией мы займемся немного позже.

Если точка входа получена, функция вызывается через указатель GetAppWindow.

После использования DLL-библиотека освобождается при помощи функции FreeLibrary, прототип который показан ниже:

```
void WINAPI FreeLibrary(HINSTANCE hLibrary);
```

В качестве параметра этой функции следует передать идентификатор освобождаемой библиотеки.

При освобождении DLL-библиотеки ее счетчик использования уменьшается. Если этот счетчик становится равным нулю (что происходит, когда все приложения, работавшие с библиотекой, освободили ее или завершили свою работу), DLL-библиотека выгружается из памяти.

Каждый раз при освобождении DLL-библиотеки вызывается функция DLLEntryPoint с параметрами DLL\_PROCESS\_DETACH или DLL\_THREAD\_DETACH, выполняющая все необходимые завершающие действия.

Теперь о функции GetProcAddress.

Для того чтобы вызвать функцию из библиотеки, зная ее идентификатор, необходимо получить значение дальнего указателя на эту функцию, вызвав функцию GetProcAddress:

```
FARPROC WINAPI GetProcAddress(HINSTANCE hLibrary,
    LPCSTR lpszProcName);
```

Через параметр hLibrary вы должны передать функции идентификатор DLL-библиотеки, полученный ранее от функции LoadLibrary.

Параметр lpszProcName является дальним указателем на строку, содержащую имя функции или ее порядковый номер, преобразованный макрокомандой MAKEINTRESOURCE.

Приведем фрагмент кода, в котором определяются адреса двух функций. В первом случае используется имя функции, а во втором - ее порядковый номер:

```
FARPROC lpMsg;
FARPROC lpTellMe;
lpMsg = GetProcAddress(hLib, "Msg");
lpTellMe = GetProcAddress(hLib, MAKEINTRESOURCE(8));
```

Перед тем как передать управление функции по полученному адресу, следует убедиться в том, что этот адрес не равен NULL:

```
if(lpMsg != (FARPROC) NULL)
{
    (*lpMsg)((LPSTR) "My message");
}
```

Для того чтобы включить механизм проверки типов передаваемых параметров, вы можете определить свой тип - указатель на функцию, и затем использовать его для преобразования типа адреса, полученного от функции GetProcAddress:

```
typedef int (PASCAL *LPGETZ)(int x, int y);
LPGETZ lpGetZ;
lpGetZ = (LPGETZ)GetProcAddress(hLib, "GetZ");
```

А что произойдет, если приложение при помощи функции LoadLibrary попытается загрузить DLL-библиотеку, которой нет на диске?

В этом случае операционная система Microsoft Windows NT выведет на экран диалоговую панель с сообщением о том, что она не может найти нужную DLL-библиотеку. В некоторых случаях появление такого сообщения нежелательно, так как либо вас не устраивает внешний вид этой диалоговой панели, либо по логике работы вашего приложения описанная ситуация является нормальной.

Для того чтобы отключить режим вывода диалоговой панели с сообщением о невозможности загрузки DLL-библиотеки, вы можете использовать функцию SetErrorMode, передав ей в качестве параметра значение SEM\_FAILCRITICALERRORS:

```
UINT nPrevErrorMode;
nPrevErrorMode = SetErrorMode(SEM_FAILCRITICALERRORS);
hDLL = LoadLibrary("DLLDEMO.DLL");
```

```
if(hDLL != NULL)
{
    // Работа с DLL-библиотекой
    . . .
}
SetErrorMode(nPrevErrorMode);
```

Приведем прототип функции SetErrorMode:  
 UINT WINAPI SetErrorMode(UINT fuErrorMode);

Эта функция позволяет отключать встроенный в Windows обработчик критических ошибок. В качестве параметра этой функции можно указывать комбинацию следующих значений:

Значение	Описание
SEM_FAILCRITICALERRORS	Операционная система Microsoft Windows NT не выводит на экран сообщения обработчика критических ошибок, возвращая приложению соответствующий код ошибки
SEM_NOGPFAULTERRORBOX	На экран не выводится сообщение об ошибке защиты памяти. Этот флаг может использоваться только при отладке приложений, если они имеют собственный обработчик такой ошибки
SEM_NOOPENFILEERRORBOX	Если Microsoft Windows NT не может открыть файл, на экран не выводится диалоговая панель с сообщением об ошибке

Функция SetErrorMode возвращает предыдущий режим обработки ошибки.

### Файл определения модуля для DLL-библиотеки

Файл определения модуля для DLL-библиотеки отличается от соответствующего файла обычного приложения Windows. В качестве примера приведем образец такого файла:

```
LIBRARY     DLLNAME
DESCRIPTION 'DLL-библиотека DLLNAME'
EXPORTS
DrawBitmap=MyDraw    @4
ShowAll
HideAll
MyPoolPtr    @5 CONSTANT
GetMyPool    @8 NONAME
FreeMyPool   @9 NONAME
```

В файле определения модуля DLL-библиотеки вместо оператора NAME должен находиться оператор LIBRARY, определяющий имя модуля DLL-библиотеки, под которым она будет известна Windows. Однако формат строк файла описания модуля зависит от используемой системы разработки. Например, если вы создаете DLL-библиотеку при помощи Microsoft Visual C++ версии 4.0, оператор LIBRARY можно не указывать.

### Анализ DLL-библиотек при помощи программы dumpbin.exe

В комплекте системы разработки Microsoft Visual C++ входит программа dumpbin.exe, предназначенная для запуска из командной строки. С помощью этой утилиты вы сможете проанализировать содержимое любого загрузочного файла в формате COFF, в том числе DLL-библиотеки, определив имена экспортируемых функций, их порядковые номера, имена DLL-библиотек и номера функций, импортируемых из этих библиотек и т. д. Можно даже дизассемблировать секции кода с использованием таблицы символов, если такая имеется в файле.

Выберем для исследования DLL-библиотеку comdlg32.dll, в которой находятся функции для работы со стандартными диалоговыми панелями.

Вначале запустим программу dumpbin.exe, передав ей в качестве параметра имя DLL-библиотеки:

```
c:\msdev\bin>dumpbin comdlg32.dll > 1st.txt
```

Перед запуском программы dumpbin.exe мы скопировали файл comdlg32.dll в каталог c:\msdev\bin.

Программа запишет в файл 1st.txt информацию о типе файла (DLL-библиотека) и перечислит названия секций и их размер, как это показано ниже:

```
Microsoft (R) COFF Binary File Dumper Version 3.10.6038
Copyright (C) Microsoft Corp 1992-1996. All rights reserved.
Dump of file comdlg32.dll
File Type: DLL
Summary
 4000 .data
 1000 .edata
 2000 .rdata
 2000 .reloc
 9000 .rsrc
17000 .text
```

Ниже мы перечислили названия некоторых стандартных секций (полное описание вы найдете в документации, которая поставляется в составе использованного вами средства разработки приложений для Microsoft Windows NT):

Название	Описание
.data	Секция инициализированных данных

.text	Секция кода
.rdata	Данные, которые можно только читать во время выполнения
.edata	Таблица экспортируемых имен
.reloc	Таблица перемещений
.rsrc	Ресурсы
.bss	Секция неинициализированных данных
.xdata	Таблица обработки исключений
.CRT	Данные библиотеки C, которые можно только читать во время выполнения
.debug	Отладочная информация
.tls	Локальная память задач

Для просмотра более подробной информации о секции следует воспользоваться параметром /SECTION:

```
c:\msdev\bin>dumpbin comdlg32.dll /SECTION:.data > lst.txt
```

Результат выполнения этой команды показан ниже:

```
Microsoft (R) COFF Binary File Dumper Version 3.10.6038
Copyright (C) Microsoft Corp 1992-1996. All rights reserved.
Dump of file comdlg32.dll
File Type: DLL
SECTION HEADER #3
  .data name
    35E4 virtual size
    1A000 virtual address
    E00 size of raw data
    18C00 file pointer to raw data
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
C8000040 flags
  Initialized Data
  Not Paged
  (no align specified)
  Read Write
  Summary
    4000 .data
```

Аналогичная информация о секции .text приведена ниже:

```
Microsoft (R) COFF Binary File Dumper Version 3.10.6038
Copyright (C) Microsoft Corp 1992-1996. All rights reserved.
Dump of file comdlg32.dll
File Type: DLL
SECTION HEADER #1
  .text name
```

```
16FF9 virtual size
1000 virtual address
17000 size of raw data
  400 file pointer to raw data
  0 file pointer to relocation table
  0 file pointer to line numbers
  0 number of relocations
  0 number of line numbers
68000020 flags
  Code
  Not Paged
  (no align specified)
  Execute Read
  Summary
    17000 .text
```

Для просмотра списка имен экспортируемых функций и глобальных переменных запустите программу dumpbin.exe с параметром /EXPORTS:

```
c:\msdev\bin>dumpbin comdlg32.dll /EXPORTS > lst.txt
```

Список появится в следующем виде:

```
Microsoft (R) COFF Binary File Dumper Version 3.10.6038
Copyright (C) Microsoft Corp 1992-1996. All rights reserved.
Dump of file comdlg32.dll
File Type: DLL
  Section contains the following Exports for comdlg32.dll
    0 characteristics
    30D0D8ED time date stamp Fri Dec 15 05:09:49 1995
    0.00 version
    1 ordinal base
    23 number of functions
    23 number of names

ordinal hint  name
1 0 ChooseColorA (00012442)
2 1 ChooseColorW (0001164F)
3 2 ChooseFontA (00009235)
4 3 ChooseFontW (00014028)
5 4 CommDlgExtendedError (0000F92A)
6 5 FindTextA (0001373A)
7 6 FindTextW (0001372A)
8 7 GetFileTitleA (0000FB57)
9 8 GetFileTitleW (0000FAF1)
10 9 GetOpenFileNameA (00004952)
11 A GetOpenFileNameW (0000F9EE)
12 B GetSaveFileNameA (0000493B)
13 C GetSaveFileNameW (0000FA37)
14 D LoadAlterBitmap (0000A450)
15 E PageSetupDlgA (00014277)
16 F PageSetupDlgW (00014373)
17 10 PrintDlgA (0000738E)
18 11 PrintDlgW (00014238)
```



```

19 12 ReplaceTextA (0001375A)
20 13 ReplaceTextW (0001374A)
21 14 WantArrows (000122D5)
22 15 dwLBS subclass (000018F7)
23 16 dwOKSubclass (000018C4)

```

```

Summary
4000 .data
1000 .edata
2000 .rdata
2000 .reloc
9000 .rsrc
17000 .text

```

Наряду с именами функций здесь отображаются порядковые номера функций и их адреса (в скобках).

Указав параметр /DISASM, вы можете дизассемблировать секцию кода, однако полученный в результате листинг может оказаться слишком большим и трудным для анализа. Вот фрагмент такого листинга:

```

Microsoft (R) COFF Binary File Dumper Version 3.10.6038
Copyright (C) Microsoft Corp 1992-1996. All rights reserved.
Dump of file comdlg32.dll
File Type: DLL
77DF1000: 83 3D EC C4 E0 77 cmp dword ptr ds:[77E0C4ECh],0
00
77DF1007: 56 push esi
77DF1008: 75 2D jne 77DF1037
77DF100A: 83 3D 8C C7 E0 77 cmp dword ptr ds:[77E0C78Ch],0
00
77DF1011: 8B 74 24 08 mov esi,dword ptr [esp+8]
77DF1015: 0F 84 93 97 00 00 je 77DFA7AE
77DF101B: 83 FE 01 cmp esi,1
77DF101E: 1B C0 sbb eax,eax
77DF1020: 24 FE and al,0FEh
77DF1022: 05 02 7F 00 00 add eax,7F02h
77DF1027: 50 push eax
77DF1028: 6A 00 push 0

```

В заключение этого раздела приведем список параметров программы dumpbin.exe.

Параметр	Описание
/ALL	Просмотр всей доступной информации, исключая листинг дизассемблирования
/ARCHIVEMEMBERS	Просмотр минимальной информации об объектах библиотеки
/DISASM	Дизассемблирование секции кода
/EXPORTS	Просмотр экспортируемых имен
/FPO	Просмотр записи FPO (Frame Pointer Optimization)

/HEADERS	Просмотр заголовка файла и заголовков каждой секции, либо заголовка каждого объекта, расположенного в библиотеке
/IMPORTS	Просмотр импортированных имен
/LINENUMBERS	Просмотр номеров строк формата COFF
/LINKERMEMBER	Просмотр доступных символов, опередленных в библиотеке как public
/OUT:ИмяФайла	Запись выходной информации не на стандартное устройство вывода (консоль), а в файл с именем ИмяФайла
/RAWDATA	Просмотр дампа каждой секции
/RELOCATIONS	Просмотр таблицы перемещений
/SECTION:Секция	Просмотр информации только о секции, имеющей имя Секция
/SUMMARY	Просмотр минимальной информации о секциях
/SYMBOLS	Просмотр таблицы символов формата COFF

### Исходные тексты DLL-библиотеки DLLDEMO

В качестве примера приведем исходные тексты простейшей DLL-библиотеки DLLDemo.DLL, в которой определены всего две функции. Первая из них - это функция инициализации DLLEntryPoint, а вторая - функция FindApplicationWindow.

Функция инициализации DLLEntryPoint в нашем случае не выполняет никакой работы, однако когда она получает управление, на экране появляется одно из четырех сообщений (в зависимости от значения кода причины вызова). Таким образом, вы сможете проследить ход инициализации DLL-библиотеки при ее отображении в адресное пространство процессов, а также при отключении DLL-библиотеки от процессов.

В задачу функции FindApplicationWindow входит поиск главного окна приложения по заголовку этого окна. В случае успеха функция FindApplicationWindow возвращает идентификатор первого найденного окна с подходящим заголовком, а при неудаче - значение NULL. Вы можете использовать эту функцию для проверки, запущено ли указанное вами приложение, или нет.

Исходный текст DLL-библиотеки представлен в листинге 3.1.

```

Листинг 3.1. Файл dlldemo\dlldemo.c
// =====
// DLL-библиотека DLLDemo.DLL
// Поиск окна по заданному заголовку
//
// (C) Фролов А.В., 1996

```

```
// Email: frolov@glas.apc.org
// =====
```

```
#include <windows.h>
#include <windowsx.h>
#include "dlldemo.h"
```

```
// Глобальная переменная, в которую записывается
// идентификатор найденного окна или значение NULL,
// если окно с заданным заголовком не найдено
HWND hwndFound;
```

```
// -----
// Функция DllEntryPoint
// Точка входа DLL-библиотеки
// -----
```

```
BOOL WINAPI DllEntryPoint(
    HMODULE hModule,    // идентификатор модуля
    DWORD   fdwReason,  // причина вызова функции DllEntryPoint
    LPVOID  lpvReserved) // зарезервировано
{
    switch(fdwReason)
    {
        // Подключение нового процесса
        case DLL_PROCESS_ATTACH:
        {
            MessageBox(NULL, "Process attached", "DLL Demo", MB_OK);
            break;
        }

        // Подключение новой задачи
        case DLL_THREAD_ATTACH:
        {
            MessageBox(NULL, "Thread attached", "DLL Demo", MB_OK);
            break;
        }

        // Отключение процесса
        case DLL_PROCESS_DETACH:
        {
            MessageBox(NULL, "Process detached", "DLL Demo", MB_OK);
            break;
        }

        // Отключение задачи
        case DLL_THREAD_DETACH:
        {
            MessageBox(NULL, "Thread detached", "DLL Demo", MB_OK);
            break;
        }
    }
    return TRUE;
}
```

```
}
```

```
// -----
// Функция FindApplicationWindow
// Поиск главного окна приложения по его заголовку
// -----
```

```
HWND FindApplicationWindow(LPSTR lpszWindowTitle)
{
    // Запускаем цикл поиска окна с заголовком,
    // адрес которого передан функции через
    // параметр lpszWindowTitle
    EnumWindows(EnumWindowsProc, (LPARAM)lpszWindowTitle);

    // Возвращаем значение глобальной переменной hwndFound,
    // которое устанавливается функцией обратного вызова
    // EnumWindowsProc в зависимости от результата поиска
    return hwndFound;
}
```

```
// -----
// Функция EnumWindowsProc
// -----
```

```
BOOL CALLBACK EnumWindowsProc(
    HWND  hwnd,    // идентификатор родительского окна
    LPARAM lParam) // адрес строки заголовка окна
{
    // Буфер для хранения заголовка окна
    char szBuf[512];

    // Получаем заголовок окна
    GetWindowText(hwnd, szBuf, 512);

    // Сравниваем заголовок со строкой, адрес которой
    // передан в функцию EnumWindowsProc через параметр lParam
    if(!strcmp((LPSTR)lParam, szBuf))
    {
        // Если заголовок совпал, сохраняем идентификатор
        // текущего окна в глобальной переменной hwndFound
        hwndFound = hwnd;

        // Завершаем цикл просмотра окон
        return FALSE;
    }

    // Если заголовок не совпал, продолжаем поиск
    else
    {
        // Записываем в глобальную переменную hwndFound
        // значение NULL. Это признак того, что окно
        // с заданным заголовком не было найдено
        hwndFound = NULL;
    }
}
```



```

    // Для продолжения поиска возвращаем значение TRUE
    return TRUE;
}
}

```

В файле `dlldemo.h` (листинг 3.2) находятся прототипы функций, определенных в нашей DLL-библиотеке.

Листинг 3.2. Файл `dlldemo\dlldemo.h`

```

BOOL WINAPI DllEntryPoint(HMODULE hModule,
    DWORD fdwReason, LPVOID lpvReserved);

HWND FindApplicationWindow(LPSTR lpszWindowTitle);

BOOL CALLBACK EnumWindowsProc(
    HWND hwnd, // идентификатор родительского окна
    LPARAM lParam); // произвольное значение

```

Файл определения модуля `dlldemo.def` DLL-библиотеки представлен в листинге 3.3.

Листинг 3.3. Файл `dlldemo\dlldemo.def`

```

EXPORTS
    FindApplicationWindow @1

```

Итак, займемся исходными текстами DLL-библиотеки.

В области глобальных переменных определена переменная `hwndFound`. В эту переменную будет записан идентификатор найденного окна или значение `NULL`, если поиск окончился неудачно.

Функция `DllEntryPoint` предназначена для инициализации библиотеки. В нашем случае функция инициализации просто выводит на экран различные сообщения в зависимости от кода причины вызова.

Функция `FindApplicationWindow` ищет главное окно приложения по заголовку этого окна, адрес которого передается ей через параметр `lpszWindowTitle`.

Для поиска окна мы использовали функцию `EnumWindows`. В качестве первого параметра этой функции передается адрес функции обратного вызова, которая будет использована для сравнения заголовков всех окон с заданным, а в качестве второго - адрес искомого заголовка:

```
EnumWindows(EnumWindowsProc, (LPARAM)lpszWindowTitle);
```

Если функция `EnumWindowsProc` найдет окно, она запишет его идентификатор в глобальную переменную `hwndFound`. Если же приложение с таким заголовком не запущено, в эту переменную будет записано значение `NULL`.

Функции обратного вызова `EnumWindowsProc` (имя функции может быть любым) передаются два параметра: идентификатор окна и 32-разрядное значение, которое передавалось функции `EnumWindows` в качестве второго параметра. В нашем случае это адрес заголовка искомого окна:

```

BOOL CALLBACK EnumWindowsProc(
    HWND hwnd, // идентификатор родительского окна
    LPARAM lParam) // адрес строки заголовка окна
{
    char szBuf[512];
    GetWindowText(hwnd, szBuf, 512);
    if(!strcmp((LPSTR)lParam, szBuf))
    {
        hwndFound = hwnd;
        return FALSE;
    }
    else
    {
        hwndFound = NULL;
        return TRUE;
    }
}

```

После вызова функции `EnumWindows` функция `EnumWindowsProc` будет вызываться в цикле для окна верного уровня каждого запущенного приложения.

Зная идентификатор окна (который передается функции `EnumWindowsProc` через первый параметр), мы с помощью функции `GetWindowText` получаем заголовок окна и записываем его в буфер `szBuf`. Затем этот заголовок сравнивается с заданным при помощи функции `strcmp`. Адрес заданного заголовка мы получаем через параметр `lParam`.

Функция обратного вызова, адрес которой указан в первом параметре функции `EnumWindows`, может вернуть значение `FALSE` или `TRUE`.

В первом случае цикл просмотра окон заканчивается и функция `EnumWindows` возвращает управление вызвавшему ее приложению. Во втором случае просмотр окон будет продолжен до тех пор, пока функции обратного вызова не будут переданы идентификаторы главных окон всех запущенных приложений.

Если окно найдено, функция обратного вызова записывает его идентификатор в глобальную переменную `hwndFound` и возвращает значение `FALSE`, после чего поиск продолжается. Если же заголовки не совпадают, в эту переменную записывается значение `NULL`, после чего для продолжения поиска функция `EnumWindowsProc` возвращает значение `TRUE`.

Несколько слов о настройке проекта DLL-библиотеки `DLLDemo.DLL`.

При создании проекта DLL-библиотеки "с нуля" вы должны указать тип рабочего пространства проекта (Project Workspace) как `Dynamic-Link Library` (рис. 3.5).

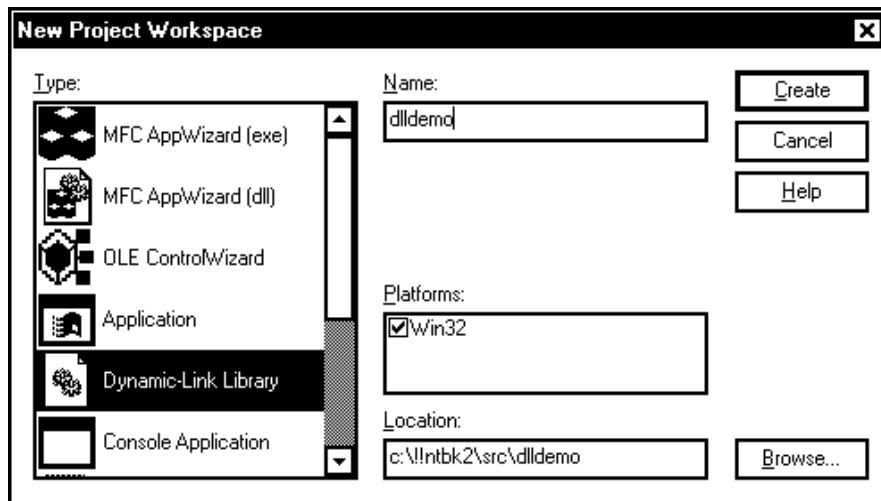


Рис. 3.5. Создание нового проекта для DLL-библиотеки

Если в вашей DLL-библиотеке определена точка входа (функция инициализации), то в параметрах проекта вы должны указать ее имя. Для этого в системе Microsoft Visual C++ версии 4.0 выберите из меню Build строку Settings. На экране появится диалоговая панель Project Settings, показанная на рис. 3.6.

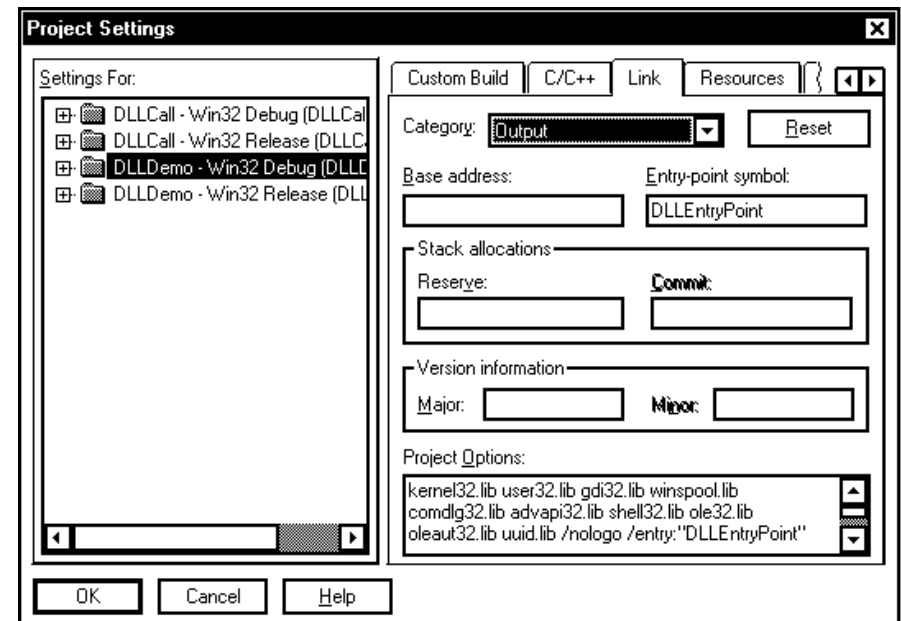


Рис. 3.6. Диалоговая панель Project Settings

Пользуясь кнопками в правом верхнем углу этой диалоговой панели, откройте страницу Link. Затем выберите из списка Category строку Output. В поле Entry-point symbol введите имя вашей функции инициализации и нажмите кнопку OK.

Если вы выполняете редактирование загрузочного модуля DLL-библиотеки в пакетном режиме, при запуске редактора связей укажите параметр /entry:"ИмяФункцииИнициализации".

Напомним, что для функции инициализации DLL-библиотеки вы можете выбрать любое имя. Нужно только указать его в проекте или в параметрах редактора связей.

### Приложение DLLCALL

Приложение DLLCALL работает совместно с DLL-библиотекой DLLDemo.DLL, описанной в предыдущем разделе.

Главное окно приложения DLLCALL и меню File показано на рис. 3.7.

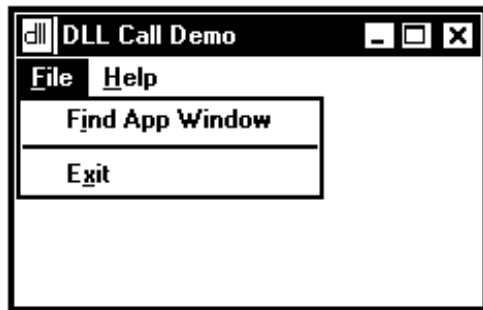


Рис. 3.7. Главное окно приложения DLLCALL

Если из меню File выбрать строку Find App Window, на экране появится диалоговая панель Find Application Window, показанная на рис. 3.8.

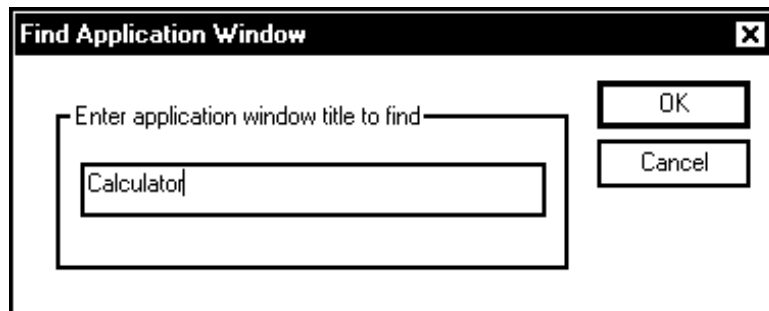


Рис. 3.8. Диалоговая панель Find Application Window

Здесь в поле Enter application window title to find вы должны ввести заголовок окна приложения. Если приложение с таким заголовком запущено, оно будет найдено, после чего на экране появится соответствующее сообщение (рис. 3.9).



Рис. 3.9. Сообщение о том, что заданное окно найдено

При инициализации DLL-библиотеки, вызванной подключением процесса DLLCALL, на экране возникает сообщение, показанное на рис. 3.10.

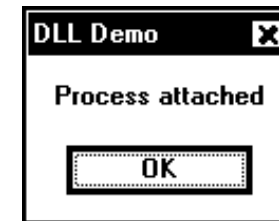


Рис. 3.10. Сообщение о подключении процесса к DLL-библиотеке

Когда приложение DLLCALL отключается от DLL-библиотеки, на экран выводится сообщение, показанное на рис. 3.11.

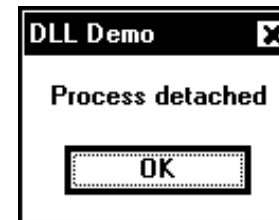


Рис. 3.11. Сообщение об отключении процесса от DLL-библиотеки

Главный файл исходных текстов приложения DLLCALL представлен в листинге 3.4.

```
Листинг 3.4. Файл dlldemo\dlldemo\dlldemo.c
// =====
// Приложение DLLCall
```

```
// Вызов функции из DLL-библиотеки
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====
```

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "resource.h"
#include "afxres.h"
```

```
#include "dllcall.h"
```

```
// Определяем тип: указатель на функцию
typedef HWND (WINAPI *MYDLLPROC) (LPSTR);
```

```
// Указатель на функцию, расположенную в
// DLL-библиотеке
MYDLLPROC GetAppWindow;
```

```
// Буфер для заголовка окна, поиск которого
// будет выполняться
char szWindowTitle[512];
```

```
// Идентификатор DLL-библиотеки
HANDLE hDLL;
```

```
HINSTANCE hInst;
char szAppName[] = "DLLCallApp";
char szAppTitle[] = "DLL Call Demo";
```

```
// -----
// Функция WinMain
// -----
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
```

```
{
    WNDCLASSEX wc;
    HWND hWnd;
    MSG msg;
```

```
// Сохраняем идентификатор приложения
hInst = hInstance;
```

```
// Проверяем, не было ли это приложение запущено ранее
hWnd = FindWindow(szAppName, NULL);
if (hWnd)
```

```
{
    // Если было, выдвигаем окно приложения на
    // передний план
```

```
    if (IsIconic(hWnd))
        ShowWindow(hWnd, SW_RESTORE);
    SetForegroundWindow(hWnd);
    return FALSE;
}
```

```
// Регистрируем класс окна
memset(&wc, 0, sizeof(wc));
wc.cbSize = sizeof(WNDCLASSEX);
wc.hIconSm = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICONSM),
    IMAGE_ICON, 16, 16, 0);
wc.style = 0;
wc.lpfnWndProc = (WNDPROC) WndProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInst;
wc.hIcon = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICON),
    IMAGE_ICON, 32, 32, 0);
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
wc.lpszClassName = szAppName;
if (!RegisterClassEx(&wc))
    if (!RegisterClass((LPWNDCLASS) &wc.style))
        return FALSE;
```

```
// Создаем главное окно приложения
hWnd = CreateWindow(szAppName, szAppTitle,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInst, NULL);
if (!hWnd) return (FALSE);
```

```
// Отображаем окно и запускаем цикл
// обработки сообщений
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;
```

```
// -----
// Функция WndProc
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
```

```

        LPARAM lParam)
{
    switch(msg)
    {
        HANDLE_MSG(hWnd, WM_COMMAND, WndProc_OnCommand);
        HANDLE_MSG(hWnd, WM_DESTROY, WndProc_OnDestroy);

        default:
            return(DefWindowProc(hWnd, msg, wParam, lParam));
    }
}

// -----
// Функция WndProc_OnDestroy
// -----
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{
    PostQuitMessage(0);
    return 0L;
}

// -----
// Функция WndProc_OnCommand
// -----
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id,
    HWND hwndCtl, UINT codeNotify)
{
    switch (id)
    {
        case ID_FILE_EXIT:
        {
            // Завершаем работу приложения
            PostQuitMessage(0);
            return 0L;
            break;
        }

        case ID_FILE_FINDWINDOW:
        {
            // Отображаем диалоговую панель для ввода
            // заголовка главного окна приложения,
            // поиск которого будет выполняться
            if(DialogBox(hInst, MAKEINTRESOURCE(IDD_DLG_FIND),
                hWnd, DlgProc))
            {
                // Первый способ вызова функции из DLL-библиотеки:
                // прямой вызов с использованием библиотеки экспорта
            }
        }
    }
}

/*
    // Выполняем поиск окна с заголовком, заданным

```

```

    // при помощи диалоговой панели
    if(FindApplicationWindow(szWindowTitle) != NULL)
        MessageBox(NULL, "Application window was found",
            szAppTitle, MB_OK | MB_ICONINFORMATION);

    else
        MessageBox(NULL, "Application window was not found",
            szAppTitle, MB_OK | MB_ICONINFORMATION);
}

// Второй способ вызова функции из DLL-библиотеки:
// загрузка DLL-библиотеки функцией LoadLibrary

// Загружаем DLL-библиотеку
hDLL = LoadLibrary("DLLDEMO.DLL");

// Если библиотека загружена успешно, выполняем
// вызов функции
if(hDLL != NULL)
{
    // Получаем адрес нужной нам функции
    GetAppWindow =
        (MYDLLPROC)GetProcAddress(hDLL,
            "FindApplicationWindow");

    // Если адрес получен, вызываем функцию
    if(GetAppWindow != NULL)
    {
        // Выполняем поиск окна с заголовком, заданным
        // при помощи диалоговой панели
        if(GetAppWindow(szWindowTitle) != NULL)
            MessageBox(NULL, "Application window was found",
                szAppTitle, MB_OK | MB_ICONINFORMATION);
        else
            MessageBox(NULL,
                "Application window was not found",
                szAppTitle, MB_OK | MB_ICONINFORMATION);
    }

    // Освобождаем DLL-библиотеку
    FreeLibrary(hDLL);
}

break;
}

case ID_HELP_ABOUT:
{
    MessageBox(hWnd,
        "DLL Call Demo\n"
        "(C) Alexandr Frolov, 1996\n"

```

```

        "Email: frolov@glas.apc.org",
        szAppTitle, MB_OK | MB_ICONINFORMATION);
        return 0L;
        break;
    }

    default:
        break;
}
return FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify,
    DefWindowProc);
}

// -----
// Функция DlgProc
// -----
LRESULT WINAPI
DlgProc(HWND hdlg, UINT msg, WPARAM wParam,
    LPARAM lParam)
{
    switch(msg)
    {
        HANDLE_MSG(hdlg, WM_INITDIALOG, DlgProc_OnInitDialog);
        HANDLE_MSG(hdlg, WM_COMMAND, DlgProc_OnCommand);

        default:
            return FALSE;
    }
}

// -----
// Функция DlgProc_OnInitDialog
// -----
BOOL DlgProc_OnInitDialog(HWND hdlg, HWND hwndFocus,
    LPARAM lParam)
{
    return TRUE;
}

// -----
// Функция DlgProc_OnCommand
// -----
#pragma warning(disable: 4098)
void DlgProc_OnCommand(HWND hdlg, int id,
    HWND hwndCtl, UINT codeNotify)
{
    switch (id)
    {
        case IDOK:
        {
            // Если пользователь нажал кнопку OK,

```

```

        // получаем текст из поля редактирования и
        // сохраняем его в глобальном буфере szWindowTitle
        GetDlgItemText(hdlg, IDC_EDIT1, szWindowTitle, 512);

        // Завершаем работу диалоговой панели
        EndDialog(hdlg, 1);
        return TRUE;
    }

    // Если пользователь нажимает кнопку Cancel,
    // завершаем работу диалоговой панели
    case IDCANCEL:
    {
        // Завершаем работу диалоговой панели
        EndDialog(hdlg, 0);
        return TRUE;
    }
    default:
        break;
}
return FALSE;
}

```

Файл `dllcall.h` (листинг 3.5) содержит прототипы функций, определенных в приложении `DLLCALL`.

Листинг 3.5. Файл `dldemo\dllcall\dllcall.h`

```

// -----
// Описание функций
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

void WndProc_OnCommand(HWND hWnd, int id,
    HWND hwndCtl, UINT codeNotify);

void WndProc_OnDestroy(HWND hWnd);

HWND FindApplicationWindow(LPSTR lpszWindowTitle);

LRESULT WINAPI
DlgProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);

BOOL DlgProc_OnInitDialog(HWND hdlg, HWND hwndFocus,
    LPARAM lParam);

void DlgProc_OnCommand(HWND hdlg, int id,
    HWND hwndCtl, UINT codeNotify);

```





```

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DLGFIND, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 240
        TOPMARGIN, 7
        BOTTOMMARGIN, 67
    END
END
#endif // APSTUDIO_INVOKED

#endif // Russian resources
////////////////////////////////////
#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////
#endif // not APSTUDIO_INVOKED

```

Файл resource.h (листинг 3.7) содержит определения констант для файла описания ресурсов приложения.

Листинг 3.7. Файл dldemo\dllcall\resource.h

```

//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by DLLCall.rc
//
#define IDR_MENU1 101
#define IDR_APPMENU 101
#define IDI_APPICON 102
#define IDI_APPICONSM 103
#define IDD_DLGFIND 104
#define IDC_EDIT1 1000
#define ID_FILE_EXIT 40001
#define ID_HELP_ABOUT 40002
#define ID_FILE_FINDWINDOW 40003

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 105
#define _APS_NEXT_COMMAND_VALUE 40004
#define _APS_NEXT_CONTROL_VALUE 1001
#define _APS_NEXT_SYMED_VALUE 101

```

```

#endif
#endif

```

Рассмотрим исходные тексты приложения DLLCALL.

### Глобальные переменные и определения

В начале файла dllcall.c (листинг 3.4) мы определили тип MYDLLPROC как указатель на функцию, возвращающую значение HWND и принимающую один параметр типа LPSTR.

Далее в области глобальных переменных мы определили переменную GetAppWindow с типом MYDLLPROC:

```
MYDLLPROC GetAppWindow;
```

Эта переменная будет использована для хранения указателя на функцию из динамически загружаемой DLL-библиотеки.

Глобальный буфер szWindowTitle предназначен для хранения заголовка окна, поиск которого будет выполнять наше приложение.

В глобальной переменной hDLL хранится идентификатор динамически загруженной DLL-библиотеки.

### Функция WinMain

Функция WinMain сохраняет идентификатор приложения в глобальной переменной hInst а затем проверяет, не было ли это приложение уже запущено. Если было, главное окно приложения выдвигается на передний план.

Далее функция WinMain регистрирует класс главного окна приложения, создает и отображает это окно. Затем запускается обычный цикл обработки сообщений.

### Функция WndProc

Функция WndProc обрабатывает сообщения WM\_COMMAND и WM\_DESTROY, для чего вызываются, соответственно, функции WndProc\_OnCommand и WndProc\_OnDestroy.

### Функция WndProc\_OnDestroy

Эта функция вызывается при уничтожении главного окна приложения. Ее задачей является завершение цикла обработки сообщений, для чего она вызывает функцию PostQuitMessage.

## Функция WndProc\_OnCommand

Эта функция обрабатывает сообщение WM\_COMMAND, которое приходит от главного меню приложения.

Когда пользователь выбирает из меню File строку Find App Window, с помощью функции DialogBox на экран выводится диалоговая панель, предназначенная для ввода заголовка окна, которое нужно найти.

Если пользователь ввел заголовок и нажал в диалоговой панели кнопку OK, функция WndProc\_OnCommand выполняет поиск окна, вызывая соответствующую функцию из DLL-библиотеки DLLDemo.DLL, исходные тексты которой мы только что рассмотрели.

В листинге мы подготовили два способа подключения DLL-библиотеки - прямой с использованием библиотеки экспорта и динамический.

Первый способ достаточно прост, однако предполагает, что в проект приложения DLLCALL будет включен файл библиотеки экспорта DLLDemo.LIB. Этот файл создается автоматически системой Microsoft Visual C++ при сборке проекта DLL-библиотеки.

Фрагмент кода, использующий прямое подключение, закрыт в листинге 3.4 символами комментария:

```
if(FindApplicationWindow(szWindowTitle) != NULL)
    MessageBox(NULL, "Application window was found",
        szAppTitle, MB_OK | MB_ICONINFORMATION);
else
    MessageBox(NULL, "Application window was not found",
        szAppTitle, MB_OK | MB_ICONINFORMATION);
```

В этом фрагменте мы выполняем простой вызов функции FindApplicationWindow, определенной в DLL-библиотеке DLLDemo.DLL. Прототип функции FindApplicationWindow мы поместили в файл dllcall.h.

Второй фрагмент загружает DLL-библиотеку при помощи функции LoadLibrary, а в случае успеха затем получает указатель на функцию FindApplicationWindow. Для получения указателя здесь применяется функция GetProcAddress:

```
hDLL = LoadLibrary("DLLDEMO.DLL");
if(hDLL != NULL)
{
    GetAppWindow =
        (MYDLLPROC)GetProcAddress(hDLL, "FindApplicationWindow");
    if(GetAppWindow != NULL)
    {
        if(GetAppWindow(szWindowTitle) != NULL)
            MessageBox(NULL, "Application window was found",
                szAppTitle, MB_OK | MB_ICONINFORMATION);
        else
            MessageBox(NULL, "Application window was not found",
                szAppTitle, MB_OK | MB_ICONINFORMATION);
    }
}
FreeLibrary(hDLL);
```

```
}
```

Проверяя работу этих двух фрагментов кода, обратите внимание на последовательность появления сообщений о подключении процесса к DLL-библиотеке и отключении от нее.

При использовании прямого подключения DLL-библиотеки сообщение о подключении процесса появляется сразу после запуска приложения DLLCALL, даже еще до появления на экране главного окна этого приложения. Это и понятно - DLL-библиотека отображается в адресное пространство процесса при его запуске. Если нужная DLL-библиотека не будет найдена, процесс так и не сможет запуститься. При этом на экране появится соответствующее системное сообщение.

Когда DLL-библиотека загружается динамически, сообщение о подключении процесса появляется только после того, как пользователь выберет строку Find App Window из меню File, так как только после этого произойдет подключение. Сообщение об отключении процесса появится после отображения результатов поиска окна, так как в этот момент будет вызвана функция FreeLibrary.

## Функция DlgProc

Функция DlgProc обрабатывает сообщения WM\_INITDIALOG и WM\_COMMAND, поступающие в функцию диалога диалоговой панели, предназначенной для ввода заголовка окна. Эти сообщения обрабатываются, соответственно, функциями DlgProc\_OnInitDialog и DlgProc\_OnCommand.

## Функция DlgProc\_OnInitDialog

Эта функция возвращает значение TRUE, разрешая отображение диалоговой панели. При создании собственного приложения вы можете добавить в эту функцию код инициализации органов управления диалоговой панели или связанных с этой панелью данных.

## Функция DlgProc\_OnCommand

Функция DlgProc\_OnCommand обрабатывает сообщение WM\_COMMAND, поступающее в функцию диалога от органов управления, расположенных в диалоговой панели.

Если пользователь нажимает кнопку OK, функция DlgProc\_OnCommand извлекает содержимое однострочного текстового редактора (введенное имя заголовка окна), вызывая для этого макрокоманду GetDlgItemText, и сохраняет это содержимое в глобальном буфере szWindowTitle. Затем функция завершает работу диалоговой панели с кодом 1, в результате чего приложение приступит к поиску окна с заданным заголовком.

---

В том случае, когда пользователь отказался от поиска, нажав в диалоговой панели кнопку Cancel, работа диалоговой панели завершается с нулевым кодом. Поиск окна в этом случае не выполняется.

## 4 НАЦИОНАЛЬНЫЕ ПАРАМЕТРЫ

Операционная система Microsoft Windows NT разрабатывалась таким образом, чтобы максимально облегчить разработку приложений, адаптированных для использования в различных странах (как сейчас модно говорить, локализованных приложений).

В чем здесь проблема?

В каждой стране используется свой национальный язык, свои символы, своя система обозначения времени и даты и так далее. Если бы разработчики пытались учесть в своих приложениях все национальные особенности, им бы пришлось затратить немало времени на создание приложений. К счастью, в операционной системе Microsoft Windows NT имеется специальный программный интерфейс, значительно облегчающий процедуру создания “интернациональных” приложений.

### Наборы национальных параметров

При установке Microsoft Windows NT пользователь может указать, какие наборы национальных параметров должны быть установлены (для каких стран). Российские пользователи в своем большинстве пожелают установить по крайней мере английский и русский набор параметров, для того чтобы было можно работать в английской и русской среде. Некоторым дополнительно могут потребоваться национальные параметры для работы с французским, немецким или каким-либо другим языком.

С точки зрения разработчика приложений наборы параметров обозначаются при помощи так называемого идентификатора наборов национальных параметров LCID (в документации к SDK этот термин звучит как locale identifier).

Многозадачная операционная система Microsoft Windows NT допускает установку отдельного набора национальных параметров для каждой задачи. Для этого предназначена функция с именем SetThreadLocale. Что же касается Microsoft Windows 95, то в среде этой операционной системы соответствующих средств нет - пользователь должен выбрать только один набор при помощи приложения Control Panel (пиктограмма Regional Settings). Новый набор параметров будет использован после перезагрузки.

Рассмотрим кратко средства, предназначенные для работы с наборами национальных параметров.

### Установка набора национальных параметров

Как мы только что сказали, в операционной системе Microsoft Windows NT для установки набора национальных параметров, используемых текущей задачей, эта задача должна вызвать функцию SetThreadLocale:

```
BOOL SetThreadLocale(
    LCID Locale); // идентификатор национального набора
```

Работа с функцией SetThreadLocale проста - вам достаточно передать ей нужный идентификатор в качестве параметра. При успехе функция возвратит значение TRUE, а при ошибке - FALSE. В последнем случае вы можете определить код ошибки с помощью функции GetLastError.

Как задавать идентификатор национального набора параметров?

Вы можете передать функции SetThreadLocale либо одну из констант, либо значение, полученное от макрокоманды MAKELCID. Список констант приведен ниже.

Константа	Описание
LOCALE_SYSTEM_DEFAULT	Идентификатор, который используется операционной системой по умолчанию
LOCALE_USER_DEFAULT	Идентификатор, который используется по умолчанию для текущего пользователя, работающего в среде Microsoft Windows NT

Макрокоманда MAKELCID позволяет составить идентификатор национального набора параметров LCID из двух значений: идентификатора национального языка и идентификатора метода сортировки. Ниже мы привели прототип этой макрокоманды и ее определение:

```
DWORD MAKELCID(
    WORD wLanguageID, // идентификатор национального языка
    WORD wSortID);    // идентификатор метода сортировки
```

```
#define MAKELCID(lgid, srtid) \
    (((DWORD)((DWORD)(WORD)(srtid)) << 16) \
    | ((DWORD)((WORD)(lgid))))
```

Что касается идентификатора сортировки, то здесь вы должны указывать значение SORT\_DEFAULT. А для создания идентификатора национального языка вам придется воспользоваться еще одной макрокомандой с именем MAKELANGID:

```
WORD MAKELANGID(
    USHORT usPrimaryLanguage, // первичный идентификатор языка
    USHORT usSubLanguage);    // вторичный идентификатор языка
```

```
#define MAKELANGID(usPrimaryLanguage, usSubLanguage) \
    (((WORD)(usSubLanguage) << 10) \
    | (WORD)(usPrimaryLanguage))
```

Первичный идентификатор задает национальный язык, а вторичный - его диалект или разновидность.

Для первичного идентификатора вы можете указать одно из следующих значений:

<i>Первичный идентификатор</i>	<i>Национальный язык</i>
LANG_AFRICAANS	Африканский
LANG_ALBANIAN	Албанский
LANG_ARABIC	Арабский
LANG_BASQUE	Баский
LANG_BULGARIAN	Болгарский
LANG_BYELORUSSIAN	Белорусский
LANG_CATALAN	Каталанский
LANG_CHINESE	Китайский
LANG_CROATIAN	Хорватский
LANG_CZECH	Чехословацкий
LANG_DANISH	Датский
LANG_DUTCH	Нидерландский
LANG_ENGLISH	Английский
LANG_ESTONIAN	Эстонский
LANG_FINNISH	Финнский
LANG_FRENCH	Французский
LANG_GERMAN	Немецкий
LANG_GREEK	Греческий
LANG_HEBREW	Еврейский
LANG_HUNGARIAN	Венгерский
LANG_ICELANDIC	Исландский
LANG_INDONESIAN	Индонезийский
LANG_ITALIAN	Итальянский
LANG_JAPANESE	Японский
LANG_KOREAN	Корейский
LANG_LATVIAN	Латвийский
LANG_LITHUANIAN	Литовский
LANG_NEUTRAL	Нейтральный
LANG_NORWEGIAN	Норвежский
LANG_POLISH	Польский

LANG_PORTUGUESE	Португальский
LANG_ROMANIAN	Румынский
LANG_RUSSIAN	Русский
LANG_SLOVAK	Словацкий
LANG_SLOVENIAN	Словенский
LANG_SORBIAN	Сербский
LANG_SPANISH	Испанский
LANG_SWEDISH	Шведский
LANG_THAI	Таиландский
LANG_TURKISH	Турецкий
LANG_UKRANIAN	Украинский

Ниже мы привели список допустимых вторичных идентификаторов:

<i>Вторичный идентификатор</i>	<i>Диалект</i>
SUBLANG_CHINESE_HONGKONG	Гонконгский диалект китайского
SUBLANG_CHINESE_SIMPLIFIED	Упрощенный диалект китайского
SUBLANG_CHINESE_SINGAPORE	Сингапурский диалект китайского
SUBLANG_CHINESE_TRADITIONAL	Традиционный китайский
SUBLANG_DEFAULT	Диалект, который используется по умолчанию
SUBLANG_DUTCH	Нидерландский
SUBLANG_DUTCH_BELGIAN	Бельгийский диалект нидерландского
SUBLANG_ENGLISH_AUS	Австрийский диалект английского
SUBLANG_ENGLISH_CAN	Канадский диалект английского
SUBLANG_ENGLISH_EIRE	Ирландский диалект английского
SUBLANG_ENGLISH_NZ	Новозеландский диалект английского
SUBLANG_ENGLISH_UK	Британский диалект английского
SUBLANG_ENGLISH_US	Американский диалект английского
SUBLANG_FRENCH	Французский
SUBLANG_FRENCH_BELGIAN	Бельгийский диалект французского
SUBLANG_FRENCH_CANADIAN	Канадский диалект французского
SUBLANG_FRENCH_SWISS	Шведский диалект французского
SUBLANG_GERMAN	Немецкий

SUBLANG_GERMAN_AUSTRIAN	Австрийский диалект немецкого
SUBLANG_GERMAN_SWISS	Швейцарский диалект немецкого
SUBLANG_ITALIAN	Итальянский
SUBLANG_ITALIAN_SWISS	Швейцарский диалект итальянского
SUBLANG_NEUTRAL	Нейтральный
SUBLANG_PORTUGUESE	Португальский
SUBLANG_PORTUGUESE_BRAZILIAN	Бразильский диалект португальского
SUBLANG_SPANISH	Испанский
SUBLANG_SPANISH_MEXICAN	Мексиканский диалект испанского
SUBLANG_SPANISH_MODERN	Современный испанский
SUBLANG_SYS_DEFAULT	Диалект, который используется операционной системой по умолчанию

Заметим, что хотя приложение может указывать любые из перечисленных выше идентификаторов национальных языков, функция `SetThreadLocale` сможет установить только те, что были выбраны при установке операционной системы Microsoft Windows NT.

И еще одно замечание.

Если в качестве первичного идентификатора языка указать константу `LANG_NEUTRAL`, то комбинации с идентификаторами `SUBLANG_NEUTRAL`, `SUBLANG_DEFAULT` и `SUBLANG_SYS_DEFAULT` будут иметь специальное значение, как это показано ниже:

<i>Вторичный идентификатор в комбинации с <code>LANG_NEUTRAL</code></i>	<i>Национальный язык</i>
<code>SUBLANG_NEUTRAL</code>	Нейтральный язык
<code>SUBLANG_DEFAULT</code>	Язык, который установлен по умолчанию для текущего пользователя, работающего с Microsoft Windows NT
<code>SUBLANG_SYS_DEFAULT</code>	Язык, который используется операционной системой по умолчанию

Ниже мы привели пример использования функции `SetThreadLocale` для установки английского и русского наборов национальных параметров:

```
// Установка английского набора параметров
fRc = SetThreadLocale(MAKELCID(
    MAKELANGID(LANG_ENGLISH, SUBLANG_NEUTRAL), SORT_DEFAULT));
```

```
// Установка русского набора параметров
fRc = SetThreadLocale(MAKELCID(
    MAKELANGID(LANG_RUSSIAN, SUBLANG_NEUTRAL), SORT_DEFAULT));
```

### Определение национального набора параметров

Помимо установки нового набора национальных параметров, часто возникает и обратная задача - определение текущего набора национальных параметров, или определение набора, который используется по умолчанию операционной системой или текущим пользователем. Для решения этой задачи в программном интерфейсе операционной системы Microsoft Windows NT предусмотрен набор функций и макрокоманд, к рассмотрению которых мы сейчас переходим.

### Определение текущего набора национальных параметров для задачи

В любой момент времени задача может получить установленный для нее идентификатор текущего набора установленных параметров, если воспользуется функцией `GetThreadLocals`:

```
LCID GetThreadLocale(VOID);
```

Эта функция не имеет параметров. Она возвращает 32-разрядный идентификатор национального набора параметров, из которого при помощи различных макрокоманд можно выделить различные компоненты.

С помощью макрокоманды `LANGIDFROMLCID` вы можете выделить из идентификатора набора национальных параметров идентификатор национального языка:

```
WORD LANGIDFROMLCID(
    LCID lcid); // идентификатор набора национальных параметров
#define LANGIDFROMLCID(lcid) ((WORD)(lcid))
```

Далее, с помощью макрокоманды `PRIMARYLANGID` нетрудно выделить из идентификатора национального языка первичный идентификатор языка:

```
WORD PRIMARYLANGID(
    WORD lgid); // идентификатор национального языка
#define PRIMARYLANGID(lgid) ((WORD)(lgid) & 0x3ff)
```

Аналогично, макрокоманда `SUBLANGID` позволяет выделить из идентификатора национального языка вторичный идентификатор языка (диалект):

```
WORD SUBLANGID(
    WORD lgid); // идентификатор национального языка
#define SUBLANGID(lgid) ((WORD)(lgid) >> 10)
```

### Определение набора национальных параметров по умолчанию

Если приложению нужно использовать наборы национальных параметров, которые используются операционной системой по умолчанию или которые

установлены для текущего пользователя по умолчанию, оно также может использовать специальные функции.

Функция `GetSystemDefaultLCID` не имеет параметров и возвращает идентификатор набора национальных параметров, которые используются операционной системой по умолчанию:

```
LCID GetSystemDefaultLCID(VOID);
```

Для определения идентификатора набора национальных параметров, установленных по умолчанию для текущего пользователя Microsoft Windows NT вы должны вызвать функцию `GetUserDefaultLCID`, которая также не имеет параметров:

```
LCID GetUserDefaultLCID(VOID);
```

Функция `GetSystemDefaultLangID` позволяет определить идентификатор национального языка, который используется операционной системой по умолчанию:

```
LANGID GetSystemDefaultLangID(VOID);
```

С помощью функции `GetUserDefaultLangID` приложение может определить идентификатор национального языка, который установлен по умолчанию для текущего пользователя Microsoft Windows NT:

```
LANGID GetUserDefaultLangID(VOID);
```

### Определение отдельных национальных параметров

Теперь, когда мы научились устанавливать и определять идентификатор национальных параметров, пора перейти к самим национальным параметрам.

Для определения значений отдельных параметров для заданного идентификатора национальных параметров вы должны использовать функцию `GetLocaleInfo`:

```
int GetLocaleInfo(
    LCID Locale, // идентификатор набора параметров
    LCTYPE LType, // тип информации
    LPTSTR lpLCData, // адрес буфера для информации
    int cchData); // размер буфера для информации
```

Параметр `Locale` этой функции задает идентификатор национальных параметров, для которого нужно определить один из конкретных параметров.

Нужный национальный параметр задается параметром `LType` функции `GetLocaleInfo`. Немного позже мы приведем сокращенный список допустимых значений для этого параметра.

Полученная информация будет записана в буфер, адрес которого задается параметром `lpLCData`, а размер - параметром `cchData`. Информация будет записана в буфер в виде текстовой строки.

Обычно буфер заказывается динамически, причем для определения требуемого размера буфера достаточно указать значение параметра

`lpLCData`, равное `NULL`, - в этом случае функция `GetLocaleInfo` вернет нужный размер буфера в байтах.

В случае успешного выполнения функция `GetLocaleInfo` возвращает размер текстовой строки с информацией, записанной в буфер `lpLCData`. При ошибке возвращается нулевое значение.

Для типа информации `LCType` можно задавать очень много значений. Все допустимые значения описаны в документации SDK. Из-за ограниченного объема книги мы не имеем возможности все их перечислить, поэтому ограничимся только самыми интересными:

- **LOCALE\_ILANGUAGE**  
Идентификатор национального языка (длиной не более 5 символов)
- **LOCALE\_SLANGUAGE**  
Полное название национального языка
- **LOCALE\_SENGLANGUAGE**  
Полное английское название языка
- **LOCALE\_SABBREVLANGNAME**  
Сокращенное трехсимвольное название языка
- **LOCALE\_SNATIVELANGNAME**  
Естественное названия языка
- **LOCALE\_ICOUNTRY**  
Код страны (длиной не более 6 символов)
- **LOCALE\_SCOUNTRY**  
Полное локализованное название страны
- **LOCALE\_SENGCOUNTRY**  
Полное английское название страны
- **LOCALE\_SABBREVCTRYNAME**  
Сокращенное название страны
- **LOCALE\_SNATIVECTRYNAME**  
Естественное название страны
- **LOCALE\_IDEFAULTLANGUAGE**  
Идентификатор основного языка, который используется в данной стране
- **LOCALE\_IDEFAULTCOUNTRY**  
Основной код страны
- **LOCALE\_IDEFAULTCODEPAGE**  
Номер кодовой страницы OEM



- **LOCALE\_IDEFAULTANSICODEPAGE**  
Номер кодовой страницы ANSI
- **LOCALE\_SLIST**  
Символ, который используется для разделения элементов списка
- **LOCALE\_IMEASURE**  
Система измерений (0 - метрическая, 1 - американская)
- **LOCALE\_SDECIMAL**  
Символ, который используется в качестве десятичного разделителя в числах
- **LOCALE\_STHOUSAND**  
Символ, который используется в качестве разделителя групп цифр в многозначных числах
- **LOCALE\_SDATE**  
Символ-разделитель в строке даты
- **LOCALE\_STIME**  
Символ-разделитель в строке времени
- **LOCALE\_IDATE**  
Порядок, в котором располагаются компоненты даты:  
0: Месяц-День-Год,  
1: День-Месяц-Год,  
2: Год-Месяц-День
- **LOCALE\_SDAYNAME1**  
Естественное длинное название для понедельника
- **LOCALE\_SDAYNAME2 - LOCALE\_SDAYNAME7**  
Естественное длинное название для дней недели от вторника до воскресения
- **LOCALE\_SABBREVDAYNAME1**  
Естественное сокращенное название для понедельника
- **LOCALE\_SABBREVDAYNAME2 - LOCALE\_SABBREVDAYNAME7**  
Естественное сокращенное название для дней недели от вторника до воскресения
- **LOCALE\_SMONTHNAME1**  
Естественное длинное название для января
- **LOCALE\_SMONTHNAME2 - LOCALE\_SMONTHNAME12**

Естественное длинное название для месяцев от февраля до декабря  
Помимо перечисленных, предусмотрены многочисленные константы для определения формата отображения даты, времени и денежных единиц, положительных и отрицательных чисел и так далее.

В качестве примера использования функции `GetLocaleInfo` приведем следующий фрагмент кода, в котором мы определяем полное название национального языка для текущей задачи:

```
GetLocaleInfo(
    GetThreadLocale(), LOCALE_SLANGUAGE, szBuf, 512);
```

Здесь полученное название языка будет записано в виде текстовой строки в буфер `szBuf`.

### Форматное преобразование даты и времени

В этом разделе мы рассмотрим две функции, предназначенные для получения текстовых строк времени и даты. Формат этих строк зависит от указанного идентификатора набора национальных параметров.

#### Преобразование времени

С помощью функции `GetTimeFormat` вы можете получить текстовую строку времени:

```
int GetTimeFormat(
    LCID Locale, // идентификатор набора параметров
    DWORD dwFlags, // флаги режимов работы функции
    CONST SYSTEMTIME *lpTime, // время
    LPCTSTR lpFormat, // строка формата времени
    LPTSTR lpTimeStr, // буфер для полученной строки времени
    int cchTime); // размер буфера в байтах
```

Через параметр `Locale` вы должны передать функции `GetTimeFormat` идентификатор набора национальных параметров, для которого необходимо выполнить форматирование строки времени.

Параметр `dwFlags` определяет режимы работы функции. Для этого параметра вы можете указать следующие значения:

Константа	Описание
LOCALE_NOUSEROVERRIDE	Строка времени будет получена в формате, который используется операционной системой по умолчанию для данного идентификатора набора национальных параметров
TIME_NOMINUTESORSECONDS	Не использовать минуты или секунды
TIME_NOSECONDS	Не использовать секунды
TIME_NOTIMEMARKER	Не использовать маркер

TIME_FORCE24HOURFORMAT	Всегда использовать 24-часовой формат времени
------------------------	---

Параметр lpTime может принимать значение NULL или содержать указатель на заполненную структуру типа SYSTEMTIME. В первом случае после завершения работы функции в буфере lpTimeStr будет получена строка для текущего времени. Во втором случае строка будет соответствовать времени, записанному в структуре SYSTEMTIME.

Приведем формат структуры SYSTEMTIME:

```
typedef struct _SYSTEMTIME
{
    WORD wYear;           // год
    WORD wMonth;          // месяц (1 - январь, 2 - февраль
                          // и так далее)
    WORD wDayOfWeek;      // день недели (0 - воскресенье.
                          // 1 - понедельник, и так далее)
    WORD wDay;            // день месяца
    WORD wHour;           // часы
    WORD wMinute;         // минуты
    WORD wSecond;         // секунды
    WORD wMilliseconds;   // миллисекунды
} SYSTEMTIME;
```

Параметр lpFormat задает строку формата, в соответствии с которым будет отформатирована выходная строка.

Если этот параметр указан как NULL, будет использован стандартный формат, принятый для данного идентификатора набора национальных параметров, указанных функции GetTimeFormat в параметре Locale. В противном случае строка формата должна быть сформирована приложением.

Строка формата времени может содержать специальные символы, пробелы и произвольные символы, заключенные в кавычки. Пробелы и произвольные символы будут появляться в выходной строке в указанном месте. Вместо специальных символов будут вставлены отдельные компоненты времени:

Символ	Компонента времени
h	Часы без ведущего нуля в 12-часовом формате
hh	Часы с ведущим нулем в 12-часовом формате
H	Часы без ведущего нуля в 24-часовом формате
HH	Часы с ведущим нулем в 24-часовом формате
m	Минуты без ведущего нуля
mm	Минуты с ведущим нулем
s	Секунды без ведущего нуля

ss	Секунды с ведущим нулем
t	Маркер (такой как A или P)
tt	Многосимвольный маркер (такой как AM или PM)

Параметры lpTimeStr и cchTime указывают, соответственно, адрес и размер буфера, в который будет записана отформатированная строка. Если параметр cchTime равен нулю, функция GetTimeFormat вернет размер буфера, достаточный для записи полной выходной строки.

Ниже мы привели пример использования функции GetTimeFormat для получения текущего времени. При этом мы указали формат, принятый по умолчанию для идентификатора набора национальных параметров, установленного в текущей задаче:

```
GetTimeFormat(GetThreadLocale(),
    LOCALE_NOUSEROVERRIDE, NULL, NULL, szBuf, 512);
```

### Преобразование даты

Отформатированную в соответствии с указанным набором национальных параметров текстовую строку даты вы можете получить от функции GetDateFormat, во многом аналогичной только что рассмотренной функции GetTimeFormat.

Приведем прототип функции GetDateFormat:

```
int GetDateFormat(
    LCID Locale,           // идентификатор набора параметров
    DWORD dwFlags,         // флаги режима работы функции
    CONST SYSTEMTIME *lpDate, // дата
    LPCTSTR lpFormat,      // строка формата даты
    LPTSTR lpDateStr,      // буфер для записи выходной строки
    int cchDate);          // размер выходного буфера в байтах
```

Рассмотрим отличия этой функции от функции GetTimeFormat.

Ниже мы привели набор констант, которые можно использовать для параметра dwFlags. Эти константы отличаются от тех, что можно использовать с функцией GetTimeFormat:

Константа	Описание
LOCALE_NOUSEROVERRIDE	Строка даты будет получена в формате, который используется операционной системой по умолчанию для данного идентификатора набора национальных параметров
DATE_SHORTDATE	Сокращенный формат даты
DATE_LONGDATE	Полный формат даты
DATE_USE_ALT_CALENDAR	Использование альтернативного

календаря, если таковой определен для данного идентификатора набора национальных параметров

Отличаются также специальные символы, которые можно использовать в строке формата даты:

Символ	Компонента даты
d	День месяца без ведущего нуля
dd	День месяца с ведущим нулем
ddd	Трехбуквенное сокращение дня недели
dddd	Полное название дня недели
M	Номер месяца без ведущего нуля
MM	Номер месяца с ведущим нулем
MMM	Трехбуквенное сокращение названия месяца
MMMM	Полное название месяца
y	Двухзначное обозначение года без ведущего нуля (последние две цифры года)
yy	Двухзначное обозначение года с ведущим нулем
yyyy	Полный номер года
gg	Название периода или эры

Пример использования функции `GetDateFormat` для получения строки текущей даты приведен ниже:

```
GetDateFormat(GetThreadLocale(),
    LOCALE_NOUSEROVERRIDE | DATE_LONGDATE,
    NULL, NULL, szBuf1, 512);
```

### Изменение раскладки клавиатуры

Помимо изменения текущего набора национальных параметров приложение может изменить раскладку клавиатуры. При этом изменится расположение символов на клавиатуре и генерируемые при нажатии на клавиши символьные коды.

Для изменения раскладки клавиатуры в программном интерфейсе операционной системы Microsoft Windows NT предусмотрено несколько специальных функций.

### Получение списка установленных раскладок

При установке операционной системы Microsoft Windows NT или Microsoft Windows 95 пользователь может выбрать для работы одну или несколько раскладок клавиатуры. Для определения списка установленных раскладок вы можете использовать функцию `GetKeyboardLayoutList`:

```
UINT GetKeyboardLayoutList(
    int nBuff, // количество элементов в буфере
    HKL *lpList); // указатель на буфер
```

Функция `GetKeyboardLayoutList` записывает в буфер, адрес которого задан параметром `lpList`, массив идентификаторов установленных раскладок клавиатуры, имеющих тип `HKL`. Через параметр `nBuff` вы должны передать функции размер буфера, указанный в количестве идентификаторов типа `HKL`. Как определить этот размер?

Для этого достаточно указать функции `GetKeyboardLayoutList` параметр `nBuff`, имеющий нулевое значение. При этом функция вернет размер массива, необходимый для записи в него всех идентификаторов.

Вот фрагмент кода, в котором приложение заказывает динамически память для идентификаторов установленных раскладок клавиатуры, а затем заполняет полученный буфер с помощью функции `GetKeyboardLayoutList`:

```
UINT uLayouts;
HKL *lpList;
uLayouts = GetKeyboardLayoutList(0, NULL);
lpList = malloc(uLayouts * sizeof(HKL));
uLayouts = GetKeyboardLayoutList(uLayouts, lpList);
```

Заметим, что младшее слово идентификатора раскладки клавиатуры содержит идентификатор набора национальных параметров. Этот факт можно использовать для определения названия национального языка или страны, соответствующего данной раскладке клавиатуры.

Операция определения названия национального языка выполняется в приведенном ниже фрагменте кода:

```
GetLocaleInfo(MAKELCID((UINT)hklCurrent & 0xffffffff),
    SORT_DEFAULT), LOCALE_SLANGUAGE, szBuf, 512);
```

### Определение названия текущей раскладки клавиатуры

Для определения названия текущей (активной) раскладки клавиатуры вы можете воспользоваться функцией `GetKeyboardLayoutName`:

```
BOOL GetKeyboardLayoutName(
    LPTSTR pwszKLID); // адрес буфера для имени раскладки
```

Через единственный параметр вы должны передать этой функции адрес буфера, в который будет записано название раскладки. Размер буфера должен быть не меньше чем `KL_NAMELENGTH` байт.

### Определение идентификатора раскладки клавиатуры для задачи

С помощью функции `GetKeyboardLayout` приложение может определить идентификатор раскладки клавиатуры, назначенный для данной задачи:

```
HKL GetKeyboardLayout(
    DWORD dwLayout); // идентификатор задачи
```

Единственный параметр этой функции определяет идентификатор задачи, для которой нужно определить идентификатор раскладки клавиатуры. Если вы укажете нулевое значение, функция возвратит идентификатор раскладки для текущей задачи.

### Загрузка раскладки клавиатуры

С помощью функции `LoadKeyboardLayout` вы можете загрузить новую раскладку клавиатуры:

```
HKL LoadKeyboardLayout(
    LPCWSTR pwszKLID, // адрес буфера названия раскладки
    UINT Flags); // флаги режима работы функции
```

При помощи параметра `pwszKLID` задается имя загружаемой раскладки. Это имя должно задаваться в виде текстовой строки, содержащей значение идентификатора национального языка в текстовом виде. Для загрузки, например, американской раскладки клавиатуры необходимо задать строку 00000409, а для загрузки русской раскладки - строку 00000419.

Параметр `Flags` задает режимы работы функции `LoadKeyboardLayout` и может иметь следующие значения:

Константа	Описание
KLF_ACTIVATE	Если указанная раскладка клавиатуры не была загружена ранее, она загружается и становится активной
KLF_REORDER	В этом случае раскладка циклически сдвигается в списке загруженных раскладок
KLF_SUBSTITUTE_OK	Использование альтернативной раскладки клавиатуры, указанной в регистрационной базе данных (ключ <code>HKEY_CURRENT_USER\Keyboard Layout\Substitutes</code> )
KLF_UNLOADPREVIOUS	Используется вместе с флагом <code>KLF_ACTIVATE</code> и только тогда, когда указанная раскладка уже загружена. В этом случае загруженная ранее раскладка клавиатуры выгружается

### Выгрузка раскладки клавиатуры

Зная идентификатор загруженной ранее раскладки клавиатуры, вы можете выгрузить эту раскладку из памяти. Для этого следует вызвать функцию `UnloadKeyboardLayout`:

```
BOOL UnloadKeyboardLayout(
    HKL hkl); // идентификатор выгружаемой раскладки
```

В качестве единственного параметра этой функции следует передать идентификатор выгружаемой раскладки клавиатуры `hkl`.

### Переключение раскладки клавиатуры

Последняя функция, которую мы рассмотрим в этом разделе и которая предназначена для работы с раскладками клавиатуры, называется `ActivateKeyboardLayout`:

```
BOOL ActivateKeyboardLayout(
    HKL hkl, // идентификатор раскладки клавиатуры
    UINT Flags); // флаги режима работы функции
```

Эта функция делает текущей раскладку клавиатуры, идентификатор которой передается ей через параметр `hkl`. Вы можете определить этот идентификатор с помощью функции `LoadKeyboardLayout` или взять из списка загруженных идентификаторов раскладок, который определяется функцией `GetKeyboardLayoutList`.

Параметр `Flags` определяет режимы работы функции и имеет следующие значения:

Константа	Описание
KLF_REORDER	Система выполняет циклический сдвиг раскладок клавиатур в списке
KLF_UNLOADPREVIOUS	Выгрузка раскладки, которая раньше была активна

Пример использования функций `GetKeyboardLayoutList` и `ActivateKeyboardLayout` вы найдете в исходных текстах приложения `SETLOCAL`, к описанию которых мы и переходим.

### Приложение SETLOCAL

В приложении `SETLOCAL` мы демонстрируем использование функций, предназначенных для работы с наборами национальных параметров, для переключения клавиатурных раскладок и получения форматированных текстовых строк времени и даты.

Главное окно приложения `SETLOCAL` показано на рис. 4.1.

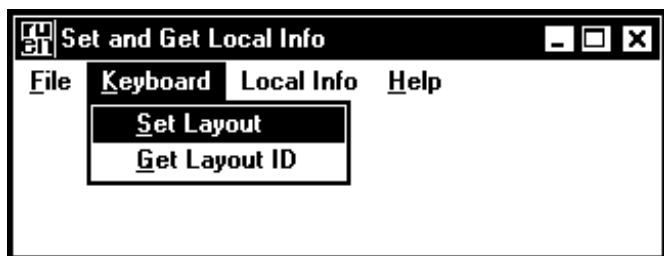


Рис. 4.1. Главное окно приложения SETLOCAL

Если из меню Keyboard выбрать строку Set Layout, на экране появится диалоговая панель Set Keyboard Layout, показанная на рис. 4.2. С помощью этой диалоговой панели вы можете изменить клавиатурную раскладку.

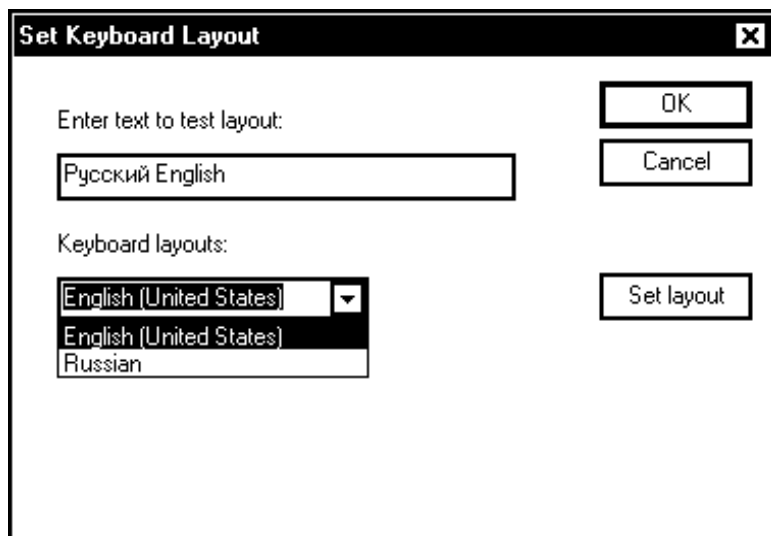


Рис. 4.2. Диалоговая панель Set Keyboard Layout

С помощью списка Keyboard layouts вы можете выбрать одну из установленных в системе клавиатурных раскладок. Если после выбора раскладки нажать кнопку Set layout или OK, выбранная раскладка станет активной.

В поле Enter text to test layout вы можете вводить символы для проверки выбранной раскладки клавиатуры. При этом удобно пользоваться кнопкой Set layout, нажатие на которую не приводит к закрытию диалоговой панели.

Кнопка Cancel позволяет отменить изменение текущей раскладки клавиатуры.

Если из меню Keyboard выбрать строку Get Layout ID, на экране появится диалоговая панель с идентификатором текущей раскладки клавиатуры. На рис. 4.3 и 4.4 это сообщение показано для американской и русской раскладки клавиатуры.



Рис. 4.3. Идентификатор американской раскладки клавиатуры

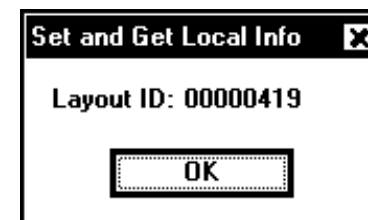


Рис. 4.4. Идентификатор русской раскладки клавиатуры

С помощью строк меню Local Info (рис. 4.5) вы можете просмотреть отдельные национальные параметры и изменить текущий набор национальных параметров.

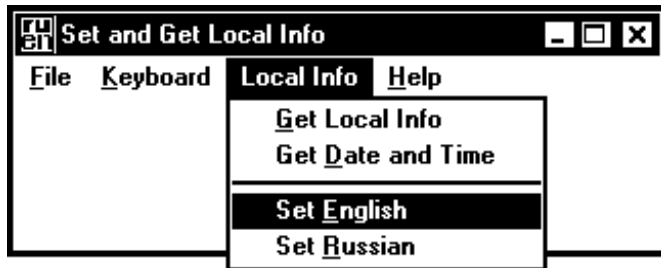


Рис. 4.5. Меню Local Info

Выбирая строки Set English и Set Russian, вы можете выбирать либо американский, либо русский набор национальных параметров. Заметим, что возможность динамического изменения набора национальных параметров отсутствует в операционной системе Microsoft Windows 95. Там вы можете изменить этот набор только при помощи приложения Control Panel с последующей перезагрузкой компьютера.

Для просмотра значений некоторых национальных параметров вы должны выбрать из меню Local Info строку Get Local Info. При этом на экране появится диалоговая панель Set and Get Local Info. На рис. 4.6 и 4.7 мы показали эту диалоговую панель для американского и русского набора параметров, соответственно.



Рис. 4.6. Значения некоторых параметров для американского набора национальных параметров



Рис. 4.7. Значения некоторых параметров для русского набора национальных параметров

Здесь мы отображаем название языка, код страны, номер кодовой страницы OEM и номер кодовой страницы ANSI.

Если из меню Local Info выбрать строку Get Date and Time, на экране появится диалоговая панель, отображающая строки даты и времени в формате, соответствующем текущему набору национальных параметров. На рис. 4.8 и 4.9 эти строки показаны для американского и русского набора параметров.

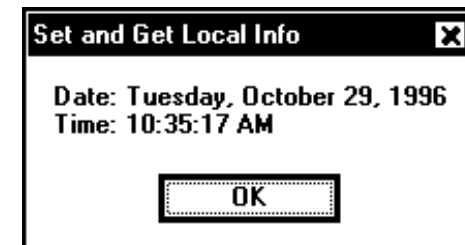


Рис. 4.8. Строка даты и времени для американского набора параметров



Рис. 4.9 Строка даты и времени для русского набора параметров

## Исходные тексты приложения SETLOCAL

Главный файл исходных текстов приложения SETLOCAL приведен в листинге 4.1.

Листинг 4.1. Файл setlocal\setlocal.c

```
// =====
// Приложение SETLOCAL
// Работа с национальными языками
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====

#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "resource.h"
#include "afxres.h"

#include "setlocal.h"

HINSTANCE hInst;
char szAppName[] = "SetLocalApp";
char szAppTitle[] = "Set and Get Local Info";

// Количество установленных раскладок клавиатуры
UINT uLayouts;

// Указатель на массив идентификаторов
// раскладок клавиатуры
HKL * lpList;

// -----
// Функция WinMain
// -----
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hWnd;
    MSG msg;

    // Сохраняем идентификатор приложения
    hInst = hInstance;

    // Проверяем, не было ли это приложение запущено ранее
    hWnd = FindWindow(szAppName, NULL);
```

```
if (hWnd)
{
    // Если было, выдвигаем окно приложения на
    // передний план
    if (IsIconic(hWnd))
        ShowWindow(hWnd, SW_RESTORE);
    SetForegroundWindow(hWnd);
    return FALSE;
}

// Определяем количество установленных
// раскладок клавиатуры
uLayouts = GetKeyboardLayoutList(0, NULL);

// Заказываем массив для хранения идентификаторов
// раскладок клавиатуры
lpList = malloc(uLayouts * sizeof(HKL));

// Заполнение массива идентификаторов
// раскладок клавиатуры
uLayouts = GetKeyboardLayoutList(uLayouts, lpList);

// Регистрируем класс окна
memset(&wc, 0, sizeof(wc));
wc.cbSize = sizeof(WNDCLASSEX);
wc.hIconSm = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICONSM),
    IMAGE_ICON, 16, 16, 0);
wc.style = 0;
wc.lpfnWndProc = (WNDPROC)WndProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInst;
wc.hIcon = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICON),
    IMAGE_ICON, 32, 32, 0);
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
wc.lpszClassName = szAppName;
if (!RegisterClassEx(&wc))
    if (!RegisterClass((LPWNDCLASS)&wc.style))
        return FALSE;

// Создаем главное окно приложения
hWnd = CreateWindow(szAppName, szAppTitle,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInst, NULL);
if (!hWnd) return(FALSE);

// Отображаем окно и запускаем цикл
```



```

// обработки сообщений
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

// Перед завершением работы приложения освобождаем
// память, полученную для хранения идентификаторов
// раскладок клавиатуры
free(lpList);

return msg.wParam;
}

// -----
// Функция WndProc
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
    switch(msg)
    {
        {
            HANDLE_MSG(hWnd, WM_COMMAND, WndProc_OnCommand);
            HANDLE_MSG(hWnd, WM_DESTROY, WndProc_OnDestroy);

            default:
                return(DefWindowProc(hWnd, msg, wParam, lParam));
        }
    }

// -----
// Функция WndProc_OnDestroy
// -----
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{
    PostQuitMessage(0);
    return 0L;
}

// -----
// Функция WndProc_OnCommand
// -----
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id,
        HWND hwndCtl, UINT codeNotify)
{
    char szBuf[1024];

```

```

char szBuf1[1024];
char szKbLayoutName[KL_NAMELENGTH];
BOOL fRc;

switch (id)
{
    case ID_FILE_EXIT:
    {
        // Завершаем работу приложения
        PostQuitMessage(0);
        return 0L;
        break;
    }

    // Установка набора национальных параметров для Англии
    case ID_LOCALINFO_SETENGLISH:
    {
        // Выполнение установки для текущей задачи
        fRc = SetThreadLocale(MAKELCID(
            MAKELANGID(LANG_ENGLISH, SUBLANG_NEUTRAL),
            SORT_DEFAULT));

        // При возникновении ошибки получаем и отображаем ее код
        if(fRc == FALSE)
        {
            wsprintf(szBuf1, "SetThreadLocale: Error %ld\n",
                GetLastError());
            MessageBox(hWnd, szBuf1, "Error", MB_OK);
        }
        break;
    }

    // Установка набора национальных параметров для России
    case ID_LOCALINFO_SETRUSSIAN:
    {
        fRc = SetThreadLocale(MAKELCID(
            MAKELANGID(LANG_RUSSIAN, SUBLANG_NEUTRAL),
            SORT_DEFAULT));

        if(fRc == FALSE)
        {
            wsprintf(szBuf1, "SetThreadLocale: Error %ld\n",
                GetLastError());
            MessageBox(hWnd, szBuf1, "Error", MB_OK);
        }
        break;
    }
}

// Получение и отображение некоторых
// национальных параметров
case ID_LOCALINFO_GETLOCALINFO:
{

```

```

// Отображение полного названия национального языка
strcpy(szBuf, "LOCALE_SLANGUAGE: ");
GetLocaleInfo(
    GetThreadLocale(), LOCALE_SLANGUAGE, szBuf1, 512);
strcat(szBuf, szBuf1);

// Отображение кода страны
strcat(szBuf, "\nLOCALE_ICOUNTRY: ");
GetLocaleInfo(
    GetThreadLocale(), LOCALE_ICOUNTRY, szBuf1, 512);
strcat(szBuf, szBuf1);

// Отображение кодовой страницы OEM
strcat(szBuf, "\nLOCALE_IDEFAULTCODEPAGE: ");
GetLocaleInfo(
    GetThreadLocale(), LOCALE_IDEFAULTCODEPAGE,
    szBuf1, 512);
strcat(szBuf, szBuf1);

// Отображение кодовой страницы ANSI
strcat(szBuf, "\nLOCALE_IDEFAULTANSICODEPAGE: ");
GetLocaleInfo(
    GetThreadLocale(), LOCALE_IDEFAULTANSICODEPAGE,
    szBuf1, 512);
strcat(szBuf, szBuf1);

MessageBox(hWnd, szBuf, szAppTitle, MB_OK);
break;
}

// Определение и отображение идентификатора
// текущей раскладки клавиатуры
case ID_KEYBOARD_GETLAYOUTID:
{
    GetKeyboardLayoutName(szKbLayoutName);
    wprintf(szBuf1, "Layout ID: %s", szKbLayoutName);

    MessageBox(hWnd, szBuf1, szAppTitle, MB_OK);
    break;
}

// Установка новой раскладки клавиатуры
case ID_KEYBOARD_SETLAYOUT:
{
    // Отображение диалоговой панели для выбора
    // раскладки клавиатуры
    DialogBox(hInst,
        MAKEINTRESOURCE(IDD_DIALOG_SETLAYOUT),
        hWnd, DlgProc);

    break;
}
}

```

```

// Просмотр текущей даты и времени в формате,
// принятом для выбранной страны
case ID_LOCALINFO_GETDATE:
{
    strcpy(szBuf, "Date: ");

    // Получаем строку даты
    GetDateFormat(
        GetThreadLocale(),
        LOCALE_NOUSEROVERRIDE | DATE_LONGDATE,
        NULL, NULL, szBuf1, 512);

    strcat(szBuf, szBuf1);
    strcat(szBuf, "\nTime: ");

    // Получаем строку времени
    GetTimeFormat(
        GetThreadLocale(),
        LOCALE_NOUSEROVERRIDE,
        NULL, NULL, szBuf1, 512);
    strcat(szBuf, szBuf1);

    // Отображаем время и дату
    MessageBox(hWnd, szBuf, szAppTitle, MB_OK);

    break;
}

case ID_HELP_ABOUT:
{
    MessageBox(hWnd,
        "Set and Get Local Information\n"
        "(C) Alexandr Frolov, 1996\n"
        "Email: frolov@glas.apc.org",
        szAppTitle, MB_OK | MB_ICONINFORMATION);
    return 0L;
    break;
}

default:
    break;
}
return FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify,
    DefWindowProc);
}

// -----
// Функция DlgProc
// -----
LRESULT WINAPI

```

```

DlgProc(HWND hdlg, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
    switch(msg)
    {
        HANDLE_MSG(hdlg, WM_INITDIALOG, DlgProc_OnInitDialog);
        HANDLE_MSG(hdlg, WM_COMMAND, DlgProc_OnCommand);

        default:
            return FALSE;
    }
}

// -----
// Функция DlgProc_OnInitDialog
// -----

BOOL DlgProc_OnInitDialog(HWND hdlg, HWND hwndFocus,
                          LPARAM lParam)
{
    UINT i;
    HKL hklCurrent;
    char szBuf[256];

    // При инициализации диалоговой панели получаем в цикле
    // идентификаторы установленных раскладок клавиатуры
    // и заполняем названиями соответствующих национальных
    // языков список типа COMBOBOX, расположенный в
    // диалоговой панели
    for(i=0; i<uLayouts; i++)
    {
        // Берем очередной идентификатор раскладки
        hklCurrent = *(lpList + i);

        // Получаем название национального языка
        GetLocaleInfo(
            MAKELCID(((UINT)hklCurrent & 0xffffffff),
                SORT_DEFAULT),
            LOCALE_SLANGUAGE, szBuf, 512);

        // Вставляем название национального языка в список
        // типа COMBOBOX
        SendMessage(GetDlgItem(hdlg, IDC_COMBO1),
            CB_ADDSTRING, 0, (LPARAM)(LPSTR)szBuf);
    }

    return TRUE;
}

// -----
// Функция DlgProc_OnCommand
// -----

```

```

#pragma warning(disable: 4098)
void DlgProc_OnCommand(HWND hdlg, int id,
                      HWND hwndCtl, UINT codeNotify)
{
    // Номер выбранной строки в списке типа COMBOBOX
    LRESULT uSelectedItem;

    switch (id)
    {
        // Когда пользователь нажимает клавишу OK,
        // устанавливаем выбранную раскладку клавиатуры
        // и завершаем работу диалоговой панели
        case IDOK:
        {
            // Определяем номер выбранной в списке строки
            uSelectedItem = SendMessage(
                GetDlgItem(hdlg, IDC_COMBO1),
                CB_GETCURSEL, 0, 0);

            // Активируем выбранную раскладку клавиатуры
            ActivateKeyboardLayout(*(lpList + uSelectedItem), 0);

            // Завершаем работу диалоговой панели
            EndDialog(hdlg, 1);
            return TRUE;
        }

        // Когда пользователь нажимает клавишу Set layout,
        // устанавливаем выбранную раскладку клавиатуры,
        // не завершая работы диалоговой панели
        case IDC_BUTTON1:
        {
            uSelectedItem = SendMessage(
                GetDlgItem(hdlg, IDC_COMBO1),
                CB_GETCURSEL, 0, 0);

            ActivateKeyboardLayout(*(lpList + uSelectedItem), 0);

            return TRUE;
        }

        // Если пользователь нажал кнопку Cancel, отменяем
        // смену раскладки клавиатуры, завершая
        // работу диалоговой панели
        case IDCANCEL:
        {
            EndDialog(hdlg, 0);
            return TRUE;
        }

        default:
            break;
    }
}

```

```
    return FALSE;
}
```

Файл setlocal.h (листинг 4.2) содержит прототипы функций, определенных в приложении SETLOCAL.

Листинг 4.2. Файл setlocal\setlocal.h

```
// -----
// Описание функций
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

void WndProc_OnCommand(HWND hWnd, int id,
    HWND hwndCtl, UINT codeNotify);

void WndProc_OnDestroy(HWND hWnd);

LRESULT WINAPI
DlgProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);

BOOL DlgProc_OnInitDialog(HWND hdlg, HWND hwndFocus,
    LPARAM lParam);
void DlgProc_OnCommand(HWND hdlg, int id,
    HWND hwndCtl, UINT codeNotify);
```

В файле resource.h (листинг 4.3), который создается автоматически системой разработки приложений Microsoft Visual C++, находятся определения констант для файла описания ресурсов.

Листинг 4.3. Файл setlocal\resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by setlocal.rc
//
#define IDR_MENU1 101
#define IDR_APPMENU 101
#define IDI_APPICON 102
#define IDI_APPICONS 103
#define IDD_DIALOG_SETLAYOUT 104
#define IDC_COMBO1 1003
#define IDC_EDIT1 1004
#define IDC_BUTTON1 1005
#define ID_FILE_EXIT 40001
#define ID_HELP_ABOUT 40002
#define ID_KEYBOARD_SETLAYOUT 40004
#define ID_KEYBOARD_GETLAYOUTID 40005
#define ID_LOCALINFO_GETLOCALINFO 40006
#define ID_LOCALINFO_SETENGLISH 40007
#define ID_LOCALINFO_SETRUSSIAN 40008
```

```
#define ID_LOCALINFO_GETDATE 40009

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 105
#define _APS_NEXT_COMMAND_VALUE 40010
#define _APS_NEXT_CONTROL_VALUE 1006
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
```

Файл описания ресурсов приложения setlocal.rc приведен в листинге 4.4. В нем определены меню, диалоговая панель, пиктограммы и текстовые строки.

Листинг 4.4. Файл setlocal\setlocal.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////
// Russian resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_RUS)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif // _WIN32

////////////////////
//
// Menu
//

IDR_APPMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit", ID_FILE_EXIT
```

```

END
POPUP "&Keyboard"
BEGIN
    MENUITEM "&Set Layout", ID_KEYBOARD_SETLAYOUT
    MENUITEM "&Get Layout ID", ID_KEYBOARD_GETLAYOUTID
END
POPUP "Local Info"
BEGIN
    MENUITEM "&Get Local Info", ID_LOCALINFO_GETLOCALINFO
    MENUITEM "Get &Date and Time", ID_LOCALINFO_GETDATE
    MENUITEM SEPARATOR
    MENUITEM "Set &English", ID_LOCALINFO_SETENGLISH
    MENUITEM "Set &Russian", ID_LOCALINFO_SETRUSSIAN
END
POPUP "&Help"
BEGIN
    MENUITEM "&About...", ID_HELP_ABOUT
END
END

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include \"afxres.h\"\\r\\n"
    "\\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\\r\\n"
    "\\0"
END

#endif // APSTUDIO_INVOKED

////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure
// application icon

```

```

// remains consistent on all systems.
IDI_APPICON          ICON          DISCARDABLE          "setlocal.ico"
IDI_APPICONSM        ICON          DISCARDABLE          "setlocsm.ico"

////////////////////////////////////
//
// Dialog

IDD_DIALOG_SETLAYOUT DIALOG DISCARDABLE 0, 0, 248, 143
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Set Keyboard Layout"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON      "OK",IDOK,191,7,50,14
    PUSHBUTTON         "Cancel",IDCANCEL,191,24,50,14
    COMBOBOX           IDC_COMBO1,14,65,102,64,CBS_DROPDOWN |
        CBS_AUTOHSCROLL | CBS_SORT | WS_VSCROLL |
        WS_TABSTOP
    EDITTEXT           IDC_EDIT1,14,28,150,14,ES_AUTOHSCROLL
    LTEXT              "Keyboard layouts:",IDC_STATIC,14,51,57,8
    LTEXT              "Enter text to test layout:",
        IDC_STATIC,14,14,75,8
    PUSHBUTTON         "Set layout",IDC_BUTTON1,191,64,50,14
END

////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG_SETLAYOUT, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 241
        TOPMARGIN, 7
        BOTTOMMARGIN, 136
    END
END
#endif // APSTUDIO_INVOKED

#endif // Russian resources

////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

```

```
////////////////////////////////////
#endif // not APSTUDIO_INVOKED
```

### Описание функций

Приведем краткое описание наиболее важных функций приложения SETLOCAL.

#### Функция WinMain

Помимо выполнения обычных действий, необходимых для создания главного окна приложения, функция WinMain получает список установленных раскладок клавиатуры, сохраняя его в глобальном массиве lpList. Память для этого массива заказывается динамически, поэтому перед завершением работы приложения мы освобождаем эту память явным образом.

Для определения размера списка и для получения самого списка раскладок клавиатуры мы используем функцию GetKeyboardLayoutList:

```
UINT uLayouts;
HKL * lpList;
uLayouts = GetKeyboardLayoutList(0, NULL);
lpList = malloc(uLayouts * sizeof(HKL));
uLayouts = GetKeyboardLayoutList(uLayouts, lpList);
```

#### Функция WndProc\_OnCommand

Функция WndProc\_OnCommand обрабатывает сообщение WM\_COMMAND, поступающее в функцию главного окна приложения от меню.

Для установки текущего набора национальных символов в этой функции используется описанная нами ранее функция SetThreadLocale, а также макрокоманды MAKELCID и MAKELANGID:

```
fRc = SetThreadLocale(MAKELCID(
    MAKELANGID(LANG_ENGLISH, SUBLANG_NEUTRAL), SORT_DEFAULT));
```

Для получения значений отдельных национальных параметров мы вызываем функцию GetLocaleInfo:

```
GetLocaleInfo(GetThreadLocale(), LOCALE_SLANGUAGE,
    szBuf1, 512);
```

В качестве идентификатора набора национальных параметров мы указываем идентификатор текущего набора параметров для основной задачи приложения, полученный от функции GetThreadLocale.

Имя текущей раскладки клавиатуры определяется при помощи функции GetKeyboardLayoutName:

```
GetKeyboardLayoutName(szKbLayoutName);
```

Для получения форматированной текстовой строки даты и времени мы вызываем функции GetDateFormat и GetTimeFormat:

```
GetDateFormat(GetThreadLocale(),
    LOCALE_NOUSEROVERRIDE | DATE_LONGDATE,
    NULL, NULL, szBuf1, 512);
GetTimeFormat(GetThreadLocale(),
    LOCALE_NOUSEROVERRIDE, NULL, NULL, szBuf1, 512);
```

#### Функция DlgProc\_OnInitDialog

Эта функция выполняет инициализацию списка типа COMBOBOX, расположенного в диалоговой панели выбора новой раскладки клавиатуры. В процессе инициализации в список записываются названия национальных языков, соответствующие установленным раскладкам клавиатуры.

#### Функция DlgProc\_OnCommand

Функция DlgProc\_OnCommand обрабатывает сообщения от органов управления, расположенных в диалоговой панели выбора новой раскладки клавиатуры.

После определения выбранной раскладки она активизируется при помощи функции ActivateKeyboardLayout, как это показано ниже:

```
uSelectedItem = SendMessage(
    GetDlgItem(hdlg, IDC_COMBO1), CB_GETCURSEL, 0, 0);
ActivateKeyboardLayout(*(lpList + uSelectedItem), 0);
```

## 5 СЕРВИСНЫЕ ПРОЦЕССЫ

Помимо обычных процессов, в операционной системе Microsoft Windows NT создаются так называемые сервисные процессы или сервисы (services). Эти процессы могут стартовать автоматически при загрузке операционной системы, по запросу приложений или других сервисов, а также в ручном режиме.

Функции, выполняемые сервисами, могут быть самыми разнообразными: от обслуживания аппаратуры и программных интерфейсов до серверов приложений, таких, например, как серверы баз данных или серверы World Wide Web (WWW).

Информация о всех серверах, установленных в системе, хранится в регистрационной базе данных. Ниже мы привели путь к этой информации:  
[HKEY\\_LOCAL\\_MACHINE\SYSTEM\CurrentControlSet\Services](#)

Для просмотра и редактирования регистрационной базы данных вы можете воспользоваться приложением regedt32.exe, которое находится в каталоге winnt\system32. Однако в программном интерфейсе WIN32 имеется набор функций, специально предназначенных для работы с записями регистрационной базы данных, имеющих отношение к сервисным процессам.

Чтобы просмотреть список установленных сервисов, вы можете запустить приложение Services из папки Control Panel. В диалоговой панели Services (рис. 5.1) отображается список установленных сервисов, их текущее состояние и режим запуска (ручной или автоматический).

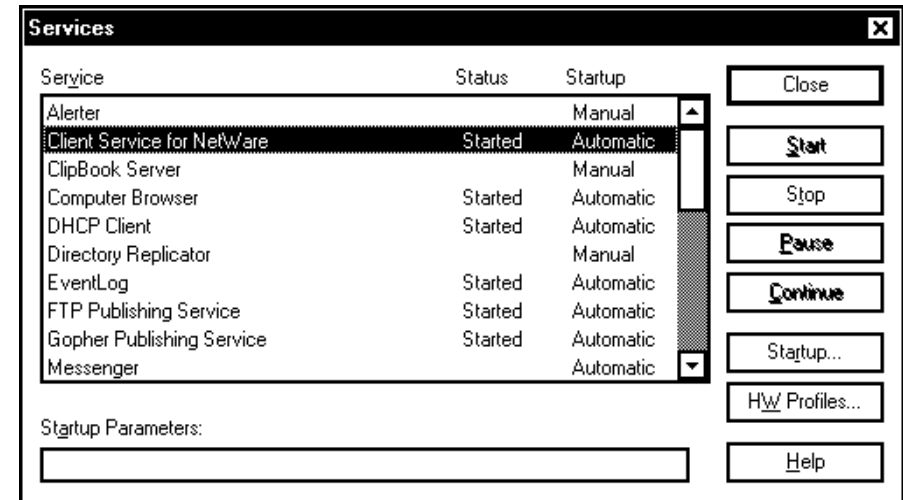


Рис. 5.1. Диалоговая панель приложения Services, предназначенная для управления сервисными процессами

Если сделать двойной щелчок левой клавишей мыши по названию сервиса, на экране появится диалоговая панель, с помощью которой можно настроить параметры сервиса (рис. 5.2).



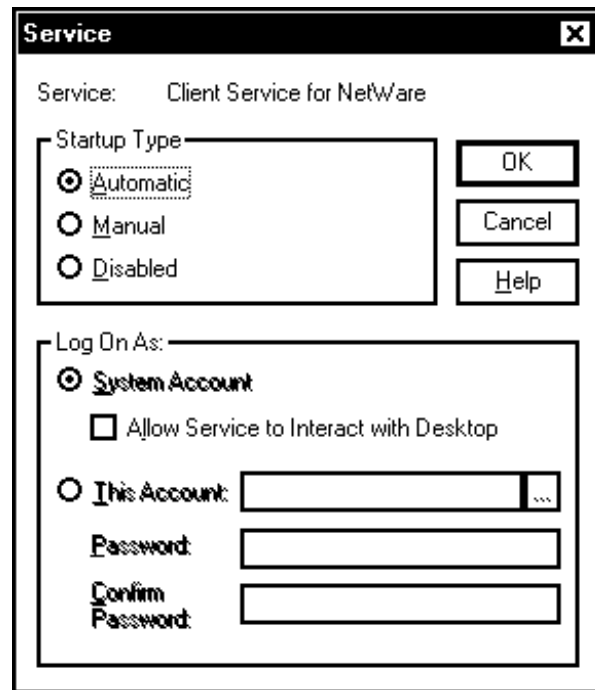


Рис. 5.2. Диалоговая панель, предназначенная для настройки параметров сервиса

С помощью группы переключателей Startup Type вы можете выбрать один из трех способов запуска сервиса.

Если включен переключатель Automatic, сервис будет запускаться автоматически при загрузке операционной системы. Этот режим удобен для тех сервисов, которые нужны постоянно, например для сервера базы данных. При этом сервер базы данных будет запускаться автоматически без участия оператора, что очень удобно.

При включении переключателя Manual сервис будет запускаться в ручном режиме. Пользователь может запустить сервис, нажав кнопку Start в диалоговой панели Services, показанной на рис. 5.1. Другое приложение или другой сервис также может запустить этот сервис при помощи специальной функции программного интерфейса WIN32. Эту функцию мы рассмотрим позже.

И, наконец, если включить переключатель Disabled, работа сервиса будет заблокирована.

Сервис может работать с привилегиями выбранных пользователей или с привилегиями системы LocalSystem. Для выбора имени пользователя в ы должны включить переключатель This Account и выбрать из списка, расположенного справа от этого переключателя, имя пользователя. Дополнительно в полях Password и Confirm Password необходимо ввести пароль пользователя.

Если включить переключатель System Account, сервис будет работать с привилегиями системы. Если сервис будет взаимодействовать с программным интерфейсом рабочего стола Desktop, следует включить переключатель Allow Service to Interact with Desktop.

Сервисы могут быть двух типов: стандартные сервисы и сервисы, соответствующие протоколам драйверов устройств Microsoft Windows NT. Последние описаны в документации DDK и не рассмотрены в нашей книге. Нажав в диалоговой панели Services, показанной на рис. 5.1, кнопку HW Profiles, вы можете выбрать один из установленных файлов конфигурации аппаратуры, которая обслуживается данным сервисом (рис. 5.3), разрешить или запретить использование выбранного файла конфигурации.

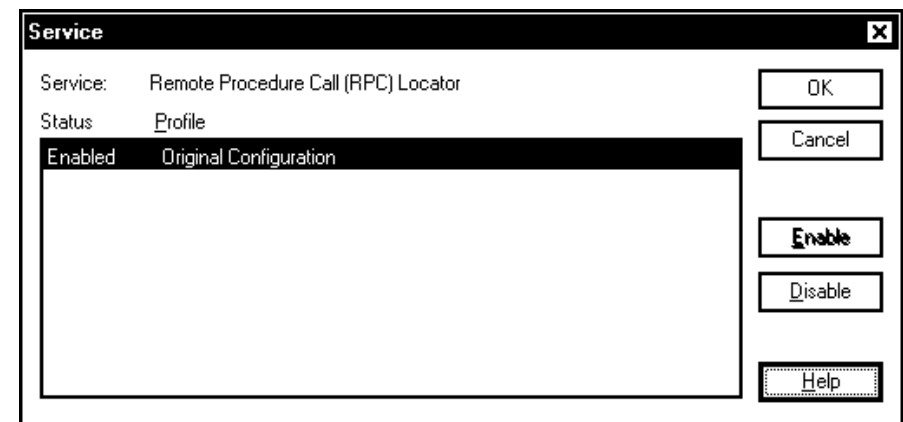


Рис. 5.3. Выбор файла конфигурации аппаратуры

При запуске операционной системы Microsoft Windows NT автоматически стартует специальный процесс, который называется процессом управления сервисами (Service Control Manager). В программном интерфейсе WIN32 имеются функции, с помощью которых приложения и сервисы могут управлять работой сервисов, обращаясь к процессу управления сервисами. Некоторые из этих функций будут рассмотрены в нашей книге.

## Создание сервисного процесса

Для того чтобы создать загрузочный модуль сервиса, вы должны подготовить исходные тексты обычного консольного приложения, имеющего функцию main (не WinMain, а именно main).

### Функция main сервисного процесса

В простейшем случае функция main вызывает функцию StartServiceCtrlDispatcher, что необходимо для подключения главной задачи сервисного процесса к процессу управления сервисами. Ниже мы привели пример функции main сервисного процесса:

```
#define MYServiceName "Sample of simple service"
void main(int argc, char *argv[])
{
    SERVICE_TABLE_ENTRY DispatcherTable[] =
    {
        {
            MYServiceName, (LPSERVICE_MAIN_FUNCTION)ServiceMain
        },
        {
            NULL, NULL
        }
    };
    if(!StartServiceCtrlDispatcher(DispatcherTable))
    {
        fprintf(stdout, "StartServiceCtrlDispatcher: Error %ld\n",
            GetLastError());
        getch();
        return;
    }
}
```

Функции StartServiceCtrlDispatcher передается указатель на массив структур типа SERVICE\_TABLE\_ENTRY. В этом массиве описываются точки входа всех сервисов, определенных в данном файле. Таким образом, в одном файле можно определить сразу несколько сервисов. Последняя строка таблицы всегда должна содержать значения NULL - это признак конца таблицы.

Тип SERVICE\_TABLE\_ENTRY и соответствующий указатель определены следующим образом:

```
typedef struct _SERVICE_TABLE_ENTRY
{
    LPCTSTR lpServiceName;
    LPSERVICE_MAIN_FUNCTION lpServiceProc;
} SERVICE_TABLE_ENTRY, *LPSERVICE_TABLE_ENTRY;
```

В поле lpServiceName записывается указатель на текстовую строку имени сервиса, а в поле lpServiceProc - указатель на точку входа сервиса.

Заметим, что функция main должна вызвать функцию StartServiceCtrlDispatcher достаточно быстро - не позднее чем через 30 секунд после запуска.

Получив управление, функция StartServiceCtrlDispatcher не возвращает его до тех пор, пока все сервисы, запущенные в рамках данного процесса, не завершат свою работу.

При успешном завершении функция StartServiceCtrlDispatcher возвращает значение TRUE. Если же произойдет ошибка, возвращается значение FALSE. Код ошибки можно определить при помощи функции GetLastError.

### Точка входа сервиса

Точка входа сервиса - это функция, адрес которой записывается в поле lpServiceProc массива структур SERVICE\_TABLE\_ENTRY. Имя функции может быть любым, а прототип должен быть таким, как показанный ниже:

```
void WINAPI ServiceMain(DWORD dwArgc, LPSTR *lpszArgv);
```

Точка входа сервиса вызывается при запуске сервиса функцией StartService (эту функцию мы рассмотрим позже). Через параметр dwArgc передается счетчик аргументов, а через параметр lpszArgv - указатель на массив строк параметров. В качестве первого параметра всегда передается имя сервиса. Остальные параметры можно задать при запуске сервиса функцией StartService.

Функция точки входа сервиса должна зарегистрировать функцию обработки команд и выполнить инициализацию сервиса.

Первая задача решается с помощью функции RegisterServiceCtrlHandler, прототип которой приведен ниже:

```
SERVICE_STATUS_HANDLE RegisterServiceCtrlHandler(
    LPCTSTR lpszServiceName, // имя сервиса
    LPHANDLER_FUNCTION lpHandlerProc); // адрес функции
// обработки команд
```

Через первый параметр этой функции необходимо передать адрес текстовой строки имени сервиса, а через второй - адрес функции обработки команд (функция обработки команд будет рассмотрена ниже).

Вот пример использования функции RegisterServiceCtrlHandler:

```
SERVICE_STATUS_HANDLE ssHandle;
ssHandle =
    RegisterServiceCtrlHandler(MYServiceName, ServiceControl);
```

Функция RegisterServiceCtrlHandler в случае успешного завершения возвращает идентификатор состояния сервиса. При ошибке возвращается нулевое значение.

Заметим, что регистрация функции обработки команд должна быть выполнена немедленно в самом начале работы функции точки входа сервиса.

Теперь перейдем к решению второй задачи - инициализации сервиса.

В процессе инициализации функция точки входа сервиса выполняет действия, которые зависят от назначения сервиса. Необходимо, однако, помнить, что без принятия дополнительных мер инициализация должна выполняться не дольше одной секунды.

А что делать, если инициализация сервиса представляет собой достаточно длительный процесс?

В этом случае перед началом инициализации функция точки входа сервиса должна сообщить процессу управления сервисами, что данный сервис находится в состоянии ожидания запуска. Это можно сделать с помощью функции `SetServiceStatus`, которая будет описана позже. Перед началом инициализации вы должны сообщить процессу управления сервисами, что сервис находится в состоянии `SERVICE_START_PENDING`.

После завершения инициализации функция точки входа сервиса должна указать процессу управления сервисами, что процесс запущен и находится в состоянии `SERVICE_RUNNING`.

### Функция обработки команд

Как следует из названия, функция обработки команд, зарегистрированная функцией `RegisterServiceCtrlHandler`, обрабатывает команды, передаваемые сервису операционной системой, другими сервисами или приложениями. Эта функция может иметь любое имя и выглядит следующим образом:

```
void WINAPI ServiceControl(DWORD dwControlCode)
{
    switch(dwControlCode)
    {
        case SERVICE_CONTROL_STOP:
        {
            ss.dwCurrentState = SERVICE_STOP_PENDING;
            ReportStatus(ss.dwCurrentState, NOERROR, 0);

            // Выполняем остановку сервиса, вызывая функцию,
            // которая выполняет все необходимые для этого действия
            // ServiceStop();

            ReportStatus(SERVICE_STOPPED, NOERROR, 0);
            break;
        }
        case SERVICE_CONTROL_INTERROGATE:
        {
            ReportStatus(ss.dwCurrentState, NOERROR, 0);
            break;
        }
        default:
        {
            ReportStatus(ss.dwCurrentState, NOERROR, 0);
            break;
        }
    }
}
```

```
}
```

В приведенном выше фрагменте кода для сообщения процессу управления сервисами текущего состояния сервиса мы вызываем созданную нами функцию `ReportStatus`. Эта функция будет описана в следующем разделе.

Через единственный параметр функция обработки команд получает код команды, который может принимать одно из перечисленных ниже значений.

Значение	Описание
<code>SERVICE_CONTROL_STOP</code>	Остановка сервиса
<code>SERVICE_CONTROL_PAUSE</code>	Временная остановка сервиса
<code>SERVICE_CONTROL_CONTINUE</code>	Продолжение работы сервиса после временной остановки
<code>SERVICE_CONTROL_INTERROGATE</code>	Когда поступает эта команда, сервис должен немедленно сообщить процессу управления сервисами свое состояние
<code>SERVICE_CONTROL_SHUTDOWN</code>	Сервис должен прекратить работу в течении 20 секунд, так как завершается работа операционной системы

### Состояние сервиса

Как мы уже говорили, сервис может сообщить процессу управления сервисами свое состояние, для чего он должен вызвать функцию `SetServiceStatus`. Прототип этой функции мы привели ниже:

```
BOOL SetServiceStatus(
    SERVICE_STATUS_HANDLE sshServiceStatus, // идентификатор
                                           // состояния сервиса
    LPSERVICE_STATUS lpssServiceStatus);    // адрес структуры,
                                           // содержащей состояние сервиса
```

Через параметр `sshServiceStatus` функции `SetServiceStatus` вы должны передать идентификатор состояния сервиса, полученный от функции `RegisterServiceCtrlHandler`.

В параметре `lpssServiceStatus` вы должны передать адрес предварительно заполненной структуры типа `SERVICE_STATUS`:

```
typedef struct _SERVICE_STATUS
{
    DWORD dwServiceType;           // тип сервиса
    DWORD dwCurrentState;          // текущее состояние сервиса
    DWORD dwControlsAccepted;       // обрабатываемые команды
    DWORD dwWin32ExitCode;         // код ошибки при запуске
    // и остановке сервиса
```

```

DWORD dwServiceSpecificExitCode; // специфический код ошибки
DWORD dwCheckPoint;             // контрольная точка при
                                // выполнении длительных операций
DWORD dwWaitHint;               // время ожидания
} SERVICE_STATUS, *LPSERVICE_STATUS;

```

В поле `dwServiceType` необходимо записать один из перечисленных ниже флагов, определяющих тип сервиса:

Флаг	Описание
<code>SERVICE_WIN32_OWN_PROCESS</code>	Сервис работает как отдельный процесс
<code>SERVICE_WIN32_SHARE_PROCESS</code>	Сервис работает вместе с другими сервисами в рамках одного и того же процесса
<code>SERVICE_KERNEL_DRIVER</code>	Сервис представляет собой драйвер операционной системы Microsoft Windows NT
<code>SERVICE_FILE_SYSTEM_DRIVER</code>	Сервис является драйвером файловой системы
<code>SERVICE_INTERACTIVE_PROCESS</code>	Сервисный процесс может взаимодействовать с программным интерфейсом рабочего стола Desktop

В поле `dwCurrentState` вы должны записать текущее состояние сервиса. Здесь можно использовать одну из перечисленных ниже констант:

Константа	Состояние сервиса
<code>SERVICE_STOPPED</code>	Сервис остановлен
<code>SERVICE_START_PENDING</code>	Сервис находится в состоянии запуска, но еще не работает
<code>SERVICE_STOP_PENDING</code>	Сервис находится в состоянии остановки, но еще не остановился
<code>SERVICE_RUNNING</code>	Сервис работает
<code>SERVICE_CONTINUE_PENDING</code>	Сервис начинает запускаться после временной остановки, но еще не работает
<code>SERVICE_PAUSE_PENDING</code>	Сервис начинает переход в состояние временной остановки, но еще не остановился

## SERVICE\_PAUSED

Сервис находится в состоянии временной остановки

Задавая различные значения в поле `dwControlsAccepted`, вы можете указать, какие команды обрабатывает сервис. Ниже приведен список возможных значений:

Значение	Команды, которые может воспринимать сервис
<code>SERVICE_ACCEPT_STOP</code>	Команда остановки сервиса <code>SERVICE_CONTROL_STOP</code>
<code>SERVICE_ACCEPT_PAUSE_CONTINUE</code>	Команды временной остановки <code>SERVICE_CONTROL_PAUSE</code> и продолжения работы после временной остановки <code>SERVICE_CONTROL_CONTINUE</code>
<code>SERVICE_ACCEPT_SHUTDOWN</code>	Команда остановки при завершении работы операционной системы <code>SERVICE_CONTROL_SHUTDOWN</code>

Значение в поле `dwWin32ExitCode` определяет код ошибки `WIN32`, который используется для сообщения о возникновении ошибочной ситуации при запуске и остановки сервиса. Если в этом поле указать значение `ERROR_SERVICE_SPECIFIC_ERROR`, то будет использован специфический для данного сервиса код ошибки, указанной в поле `dwServiceSpecificExitCode` структуры `SERVICE_STATUS`. Если ошибки нет, в поле `dwWin32ExitCode` необходимо записать значение `NO_ERROR`.

Поле `dwServiceSpecificExitCode` используется в том случае, когда в поле `dwWin32ExitCode` указано значение `ERROR_SERVICE_SPECIFIC_ERROR`.

Теперь о поле `dwCheckPoint`.

Это поле должно содержать значение, которое должно периодически увеличиваться при выполнении длительных операций запуска, остановки или продолжения работы после временной остановки. Если выполняются другие операции, в это поле необходимо записать нулевое значение.

Содержимое поля `dwWaitHint` определяет ожидаемое время выполнения (в миллисекундах) длительной операции запуска, остановки или продолжения работы после временной остановки. Если за указанное время не изменится содержимое полей `dwCheckPoint` или `dwCurrentState`, процесс управления сервисами будет считать, что произошла ошибка.

В наших примерах для сообщения текущего состояния сервиса процессу управления сервисами мы используем функцию ReportStatus, исходный текст которой приведен ниже:

```
void ReportStatus(DWORD dwCurrentState,
    DWORD dwWin32ExitCode, DWORD dwWaitHint)
{
    static DWORD dwCheckPoint = 1;
    if(dwCurrentState == SERVICE_START_PENDING)
        ss.dwControlsAccepted = 0;
    else
        ss.dwControlsAccepted = SERVICE_ACCEPT_STOP;
    ss.dwCurrentState = dwCurrentState;
    ss.dwWin32ExitCode = dwWin32ExitCode;
    ss.dwWaitHint = dwWaitHint;
    if((dwCurrentState == SERVICE_RUNNING) ||
        (dwCurrentState == SERVICE_STOPPED))
        ss.dwCheckPoint = 0;
    else
        ss.dwCheckPoint = dwCheckPoint++;
    SetServiceStatus(ssHandle, &ss);
}
```

При заполнении структуры SERVICE\_STATUS эта функция проверяет содержимое поля dwCurrentState. Если сервис находится в состоянии ожидания запуска, в поле допустимых команд dwControlsAccepted записывается нулевое значение. В противном случае функция записывает туда значение SERVICE\_ACCEPT\_STOP, в результате чего сервису может быть передана команда остановки. Далее функция заполняет поля dwCurrentState, dwWin32ExitCode и dwWaitHint значениями, полученными через параметры.

В том случае, когда сервис выполняет команды запуска или остановки, функция увеличивает значение счетчика шагов длительных операций dwCheckPoint. Текущее значение счетчика хранится в статической переменной dwCheckPoint, определенной в нашей функции.

После подготовки структуры SERVICE\_STATUS ее адрес передается функции установки состояния сервиса SetServiceStatus.

Для определения текущего состояния сервиса вы можете использовать функцию QueryServiceStatus, прототип которой приведен ниже:

```
BOOL QueryServiceStatus(
    SC_HANDLE schService, // идентификатор сервиса
    LPSERVICE_STATUS lpssServiceStatus); // адрес структуры
// SERVICE_STATUS
```

Идентификатор сервиса вы можете получить от функций OpenService или CreateService, которые будут описаны ниже.

## Управление сервисами

Вы можете создать приложение или сервис, управляющее сервисами. В этом разделе мы рассмотрим основные функции программного интерфейса WIN32, предназначенные для управления сервисами. Более подробную информацию вы найдете в документации SDK.

### Получение идентификатора системы управления сервисами

Идентификатор системы управления сервисами нужен для выполнения различных операций над сервисами, таких например, как установка сервиса. Вы можете получить этот идентификатор с помощью функции OpenSCManager:

```
SC_HANDLE OpenSCManager(
    LPCTSTR lpszMachineName, // адрес имени рабочей станции
    LPCTSTR lpszDatabaseName, // адрес имени базы данных
    DWORD fdwDesiredAccess); // нужный тип доступа
```

Задавая имя рабочей станции через параметр lpszMachineName, вы можете получить идентификатор системы управления сервисами на любом компьютере сети. Для локального компьютера необходимо указать значение NULL.

Для наших примеров параметр lpszDatabaseName, определяющий имя базы данных системы управления сервисами, нужно указать как NULL. При этом по умолчанию будет использована база данных активных сервисов ServicesActive.

Через параметр fdwDesiredAccess нужно задать требуемый тип доступа. Здесь можно использовать следующие константы:

Константа	Разрешенный тип доступа
SC_MANAGER_ALL_ACCESS	Полный доступ
SC_MANAGER_CONNECT	Подключение к системе управления сервисами
SC_MANAGER_CREATE_SERVICE	Создание новых сервисов и добавление их к регистрационной базе данных
SC_MANAGER_ENUMERATE_SERVICE	Просмотр списка всех установленных сервисов при помощи функции EnumServicesStatus (в нашей книге эта функция и следующие две функции не описаны)
SC_MANAGER_LOCK	Блокирование баз данных функцией LockServiceDatabase

SC_MANAGER_QUERY_LOCK_STATUS	Определение состояние блокировки базы данных функцией QueryServiceLockStatus
------------------------------	--

Ниже мы привели пример вызова функции OpenSCManager:

```
schSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
```

После использования вы должны закрыть идентификатор, полученный от функции OpenSCManager. Для этого необходимо вызвать функцию CloseServiceHandle:

```
CloseServiceHandle(schSCManager);
```

### Установка сервиса

Для установки сервиса в систему вы должны использовать функцию CreateService, которая вносит все необходимые дополнения в регистрационную базу данных.

Прототип функции CreateService мы привели ниже:

```
SC_HANDLE CreateService(
    SC_HANDLE hSCManager,    // идентификатор базы данных системы
                             // управления сервисами
    LPCTSTR lpServiceName,    // имя сервиса, которое будет использовано
                             // для запуска
    LPCTSTR lpDisplayName,    // имя сервиса для отображения
    DWORD dwDesiredAccess,    // тип доступа к сервису
    DWORD dwServiceType,      // тип сервиса
    DWORD dwStartType,        // способ запуска сервиса
    DWORD dwErrorControl,     // действия при ошибках в момент запуска
    LPCTSTR lpBinaryPathName, // путь к загрузочному файлу сервиса
    LPCTSTR lpLoadOrderGroup, // имя группы порядка загрузки
    LPDWORD lpdwTagId,        // адрес переменной для сохранения
                             // идентификатора тега
    LPCTSTR lpDependencies,    // адрес массива имен взаимосвязей
    LPCTSTR lpServiceStartName, // адрес имени пользователя, права
                             // которого будут применены для работы сервиса
    LPCTSTR lpPassword );      // адрес пароля пользователя
```

Через параметр hSCManager вы должны передать функции CreateService идентификатор базы данных системы управления сервисами, полученный от функции OpenSCManager, описанной выше.

Через параметры lpServiceName и lpDisplayName задаются, соответственно, имя сервиса, которое будет использовано для запуска и имя сервиса для отображения в списке установленных сервисов.

С помощью параметра dwDesiredAccess вы должны указать тип доступа, разрешенный при обращении к данному сервису. Здесь вы можете указать следующие значения:

Значение	Разрешенный тип доступа
----------	-------------------------

SERVICE_ALL_ACCESS	Полный доступ
SERVICE_CHANGE_CONFIG	Изменение конфигурации сервиса функцией ChangeServiceConfig
SERVICE_ENUMERATE_DEPENDENTS	Просмотр сервиса в списке сервисов, созданных на базе данного сервиса функцией EnumDependentServices
SERVICE_INTERROGATE	Выдача сервису команды немедленного определения текущего состояния сервиса при помощи функции ControlService (эта функция будет описана ниже)
SERVICE_PAUSE_CONTINUE	Временная остановка сервиса или продолжение работы после временной остановки
SERVICE_QUERY_CONFIG	Определение текущей конфигурации функцией QueryServiceConfig
SERVICE_QUERY_STATUS	Определение текущего состояния сервиса функцией QueryServiceStatus
SERVICE_START	Запуск сервиса функцией StartService
SERVICE_STOP	Остановка сервиса выдачей соответствующей команды функцией ControlService
SERVICE_USER_DEFINE_CONTROL	Возможность передачи сервису команды, определенной пользователем, с помощью функции ControlService

Через параметр dwServiceType необходимо передать тип сервиса. Здесь вы можете указывать те же самые флаги, что и в поле dwServiceType структуры SERVICE\_STATUS, описанной выше:

Флаг	Описание
SERVICE_WIN32_OWN_PROCESS	Сервис работает как отдельный процесс
SERVICE_WIN32_SHARE_PROCESS	Сервис работает вместе с другими сервисами в рамках одного и того же



SERVICE_KERNEL_DRIVER	процесса Сервис представляет собой драйвер операционной системы Microsoft Windows NT
SERVICE_FILE_SYSTEM_DRIVER	Сервис является драйвером файловой системы
SERVICE_INTERACTIVE_PROCESS	Сервисный процесс может взаимодействовать с программным интерфейсом рабочего стола Desktop

В параметре `dwStartType` указывается один из следующих способов запуска сервиса:

Константа	Способ запуска
SERVICE_BOOT_START	Используется только для сервисов типа SERVICE_KERNEL_DRIVER или SERVICE_FILE_SYSTEM_DRIVER (драйверы). Указывает, что драйвер должен загружаться при загрузке операционной системы
SERVICE_SYSTEM_START	Аналогично предыдущему, но драйвер запускается при помощи функции <code>IoInitSystem</code> , не описанной в нашей книге
SERVICE_AUTO_START	Драйвер или обычный сервис, который запускается при загрузке операционной системы
SERVICE_DEMAND_START	Драйвер или обычный сервис, который запускается функцией <code>StartService</code>
SERVICE_DISABLED	Отключение возможности запуска драйвера или обычного сервиса

Параметр `dwErrorControl` задает действия, выполняемые при обнаружении ошибки в момент загрузки сервиса. Здесь можно указывать одно из следующих значений:

Значение	Реакция на ошибку
SERVICE_ERROR_IGNORE	Протоколирование ошибки в системном журнале и продолжение процедуры запуска сервиса
SERVICE_ERROR_NORMAL	Протоколирование ошибки в

SERVICE_ERROR_SEVERE	системном журнале без продолжения процедуры запуска сервиса Протоколирование ошибки в системном журнале. Если это возможно, используется конфигурация, с которой сервис успешно был запущен в прошлый раз. В противном случае система перезапускается с использованием работоспособной конфигурации
SERVICE_ERROR_CRITICAL	Критичная ошибка. Сообщение при возможности записывается в системный журнал. Операция запуска отменяется, система перезапускается с использованием работоспособной конфигурации

В параметре `lpBinaryPathName` вы должны указать полный путь к загрузочному файлу сервиса.

Через параметр `lpLoadOrderGroup` передается указатель на имя группы порядка загрузки сервиса. Сделав сервис членом одной из групп порядка загрузки, вы можете определить последовательность загрузки вашего сервиса относительно других сервисов. Список групп порядка загрузки находится в регистрационной базе данных:

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ServiceGroupOrder`

Если относительный порядок загрузки не имеет значения, укажите для параметра `lpLoadOrderGroup` значение `NULL`.

Параметр `lpdwTagId` используется только в том случае, если значение параметра `lpLoadOrderGroup` не равно `NULL`.

Параметр `lpDependencies` должен содержать указатель на массив строк имен сервисов или групп порядка загрузки, которые должны быть запущены перед запуском данного сервиса. Последняя строка такого массива должна быть закрыта двумя двоичными нулями. Если зависимостей от других сервисов нет, для параметра `lpDependencies` можно указать значение `NULL`.

Последние два параметра функции `lpServiceStartName` и `lpPassword` указывают, соответственно, имя и пароль пользователя, с правами которого данный сервис будет работать в системе (имя указывается в форме "ИмяДомена\имяПользователя"). Если параметр `lpServiceStartName` указан как `NULL`, сервис подключится к системе как пользователь `LocalSystem`. При этом параметр `lpPassword` должен быть указан как `NULL`.

Ниже мы привели фрагмент исходного текста приложения, в котором выполняется установка сервиса из каталога `c:\ntbk2\src\service\small\debug`:

```
schSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
```



```
schService = CreateService(
    schSCManager, MYServiceName, MYServiceName,
    SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS,
    SERVICE_DEMAND_START, SERVICE_ERROR_NORMAL,
    "c:\\ntbk2\\src\\service\\small\\debug\\small.exe",
    NULL, NULL, "", NULL, NULL);
CloseServiceHandle(schSCManager);
```

### Получение идентификатора сервиса

Для выполнения операций с сервисом вы должны получить его идентификатор. Это нетрудно сделать с помощью функции `OpenService`, прототип которой мы привели ниже:

```
SC_HANDLE OpenService(
    SC_HANDLE schSCManager, // идентификатор базы данных системы
                                // управления сервисами
    LPCTSTR lpszServiceName, // имя сервиса
    DWORD fdwDesiredAccess); // тип доступа к сервису
```

Через параметр `schSCManager` вы должны передать функции `OpenService` идентификатор базы данных системы управления сервисами, полученный от функции `OpenSCManager`.

Параметр `lpszServiceName` определяет имя сервиса, а параметр `fdwDesiredAccess` - желаемый тип доступа к сервису.

### Выдача команд сервису

Приложение или сервис может выдать команду сервису, вызвав функцию `ControlService`:

```
BOOL ControlService(
    SC_HANDLE hService, // идентификатор сервиса
    DWORD dwControl, // код команды
    LPSERVICE_STATUS lpServiceStatus); // адрес структуры состояния
                                // сервиса SERVICE_STATUS
```

В качестве кода команды вы можете указать один из следующих стандартных кодов:

Код	Команда
<code>SERVICE_CONTROL_STOP</code>	Остановка сервиса
<code>SERVICE_CONTROL_PAUSE</code>	Временная остановка сервиса
<code>SERVICE_CONTROL_CONTINUE</code>	Продолжение работы сервиса после временной установки
<code>SERVICE_CONTROL_INTERROGATE</code>	Запрос текущего состояния сервиса

Дополнительно вы можете указывать коды команд, определенные вами. Они должны находиться в интервале значений от 128 до 255.

### Удаление сервиса из системы

Для удаления сервиса из системы используется функция `DeleteService`. В качестве единственного параметра этой функции необходимо передать идентификатор сервиса, полученный от функции `OpenService`.

Ниже мы привели фрагмент приложения, удаляющий сервис с именем `MYServiceName` из системы:

```
schSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
schService = OpenService(
    schSCManager, MYServiceName, SERVICE_ALL_ACCESS);
ControlService(schService, SERVICE_CONTROL_STOP, &ss);
DeleteService(schService);
CloseServiceHandle(schSCManager);
```

Заметим, что перед удалением мы останавливаем сервис.

### Запуск сервиса

Для запуска сервиса вы должны использовать функцию `StartService`:

```
BOOL StartService(
    SC_HANDLE schService, // идентификатор сервиса
    DWORD dwNumServiceArgs, // количество аргументов
    LPCTSTR *lpszServiceArgs); // адрес массива аргументов
```

Через параметр `schService` вы должны передать функции `StartService` идентификатор сервиса, полученный от функции `OpenService`.

Параметры `dwNumServiceArgs` и `lpszServiceArgs` определяют, соответственно, количество аргументов и адрес массива аргументов, которые получит функция точки входа сервиса. Эти параметры могут использоваться в процессе инициализации.

Ниже мы привели фрагмент исходного текста приложения, выполняющий запуск сервиса:

```
schSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
schService = OpenService(
    schSCManager, MYServiceName, SERVICE_ALL_ACCESS);
StartService(schService, 0, NULL);
CloseServiceHandle(schSCManager);
```

### Остановка сервиса

Остановка сервиса выполняется посылкой сервису команды `SERVICE_CONTROL_STOP`, для чего эта команда передается функции `ControlService`:

```
ControlService(schService, SERVICE_CONTROL_STOP, &ss);
```

### Определение конфигурации сервиса

Приложение или сервис может определить конфигурацию заданного сервиса, вызвав для этого функцию `QueryServiceConfig`:

```

BOOL QueryServiceConfig(
    SC_HANDLE schService, // идентификатор сервиса
    LPQUERY_SERVICE_CONFIG lpqscServConfig, // адрес структуры
    // QUERY_SERVICE_CONFIG, в которую будет
    // записана конфигурация сервиса
    DWORD cbBufSize, // размер буфера для записи конфигурации
    LPDWORD lpcbBytesNeeded); // адрес переменной, в которую будет
    // записан размер буфера, необходимый для
    // сохранения всей информации о конфигурации

```

Формат структуры QUERY\_SERVICE\_CONFIG приведен ниже:

```

typedef struct _QUERY_SERVICE_CONFIG
{
    DWORD dwServiceType;
    DWORD dwStartType;
    DWORD dwErrorControl;
    LPTSTR lpBinaryPathName;
    LPTSTR lpLoadOrderGroup;
    DWORD dwTagId;
    LPTSTR lpDependencies;
    LPTSTR lpServiceStartName;
    LPTSTR lpDisplayName;
} QUERY_SERVICE_CONFIG, LPQUERY_SERVICE_CONFIG;

```

Содержимое полей этой структуры соответствует содержанию параметров функции CreateService, описанной ранее.

Ниже расположен фрагмент кода, в котором определяется текущая конфигурация сервиса:

```

schSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
schService = OpenService(
    schSCManager, MYServiceName, SERVICE_ALL_ACCESS);
lpBuf = (LPQUERY_SERVICE_CONFIG)malloc(4096);
if(lpBuf != NULL)
{
    QueryServiceConfig(schService, lpBuf, 4096, &dwBytesNeeded);
    ...
    free(lpBuf);
}

```

## Приложение SRVCTRL

В этом разделе мы приведем исходные тексты простейшего сервиса Simple и приложения SRVCTRL, с помощью которого можно установить данный сервис, запустить, остановить или удалить его, а также определить текущую конфигурацию.

Главное меню приложения SRVCTRL и временное меню Service показано на рис. 5.4.

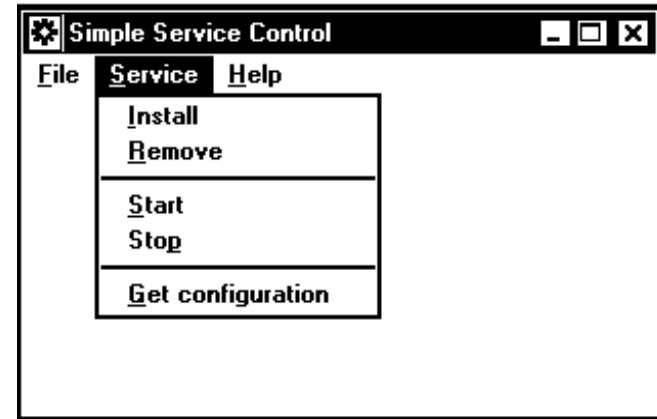


Рис. 5.4. Главное меню приложения SRVCTRL

С помощью строки Install вы можете установить сервис. Не забудьте перед этим записать загрузочный файл сервиса в каталог c:\ntbk2\src\service\small\debug, так как приложение SRVCTRL может установить сервис только из этого каталога.

После установки имя сервиса появится в списке сервисов, который можно просмотреть при помощи приложения Services из папки Control Panel (рис. 5.5).

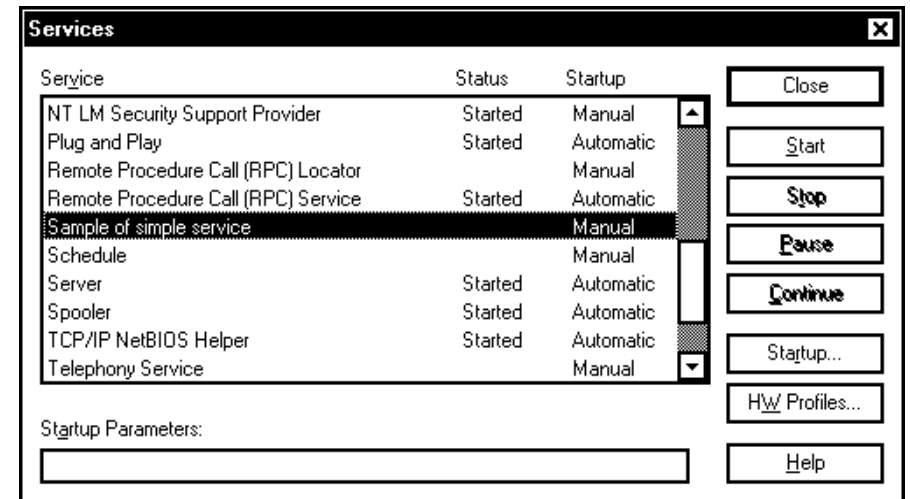


Рис. 5.5. В списке сервисов появился новый сервис *Sample of simple service*

Если теперь из меню Service нашего приложения выбрать строку Get configuration, на экране появится диалоговая панель, в которой будут отображены некоторые поля структуры конфигурации сервиса (рис. 5.6).

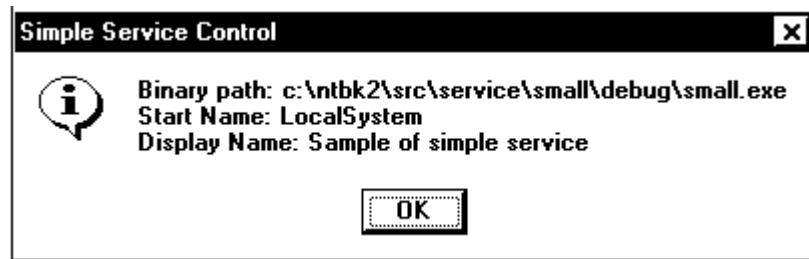


Рис. 5.6. Просмотр конфигурации сервиса

### Исходный текст сервиса

Исходный текст сервиса представлен в листинге 5.1. Так как ранее мы уже подробно описывали структуру этого сервиса, то мы оставим вам этот листинг и листинг приложения SRVCTRL на самостоятельное изучение.

Листинг 5.1. Файл service/small/small.c

```
// =====
// Сервис "Sample of simple service"
// Шаблон простейшего сервиса Windows NT
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====

#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

#include "small.h"

// -----
// Глобальные переменные
// -----

// Код ошибки
DWORD dwErrCode;
```

```
// Текущее состояние сервиса
SERVICE_STATUS ss;

// Идентификатор сервиса
SERVICE_STATUS_HANDLE ssHandle;

// -----
// Функция main
// Точка входа процесса
// -----
void main(int argc, char *argv[])
{
    // Таблица точек входа
    SERVICE_TABLE_ENTRY DispatcherTable[] =
    {
        {
            // Имя сервиса
            MYServiceName,

            // Функция main сервиса
            (LPSERVICE_MAIN_FUNCTION)ServiceMain
        },
        {
            NULL,
            NULL
        }
    };

    printf("Sample of simple service\n"
        "(C) A. Frolov, 1996, Email: frolov@glas.apc.org\n");

    // Запуск диспетчера
    if(!StartServiceCtrlDispatcher(DispatcherTable))
    {
        fprintf(stdout,
            "StartServiceCtrlDispatcher: Error %ld\n",
            GetLastError());
        getch();
        return;
    }

    // -----
    // Функция ServiceMain
    // Точка входа сервиса
    // -----
    void WINAPI ServiceMain(DWORD argc, LPSTR *argv)
    {
        // Регистрируем управляющую функцию сервиса
        ssHandle =
```

```

    RegisterServiceCtrlHandler(MYServiceName, ServiceControl);

    if(!ssHandle)
        return;

    // Устанавливаем состояние сервиса

    // Сервис работает как отдельный процесс
    ss.dwServiceType = SERVICE_WIN32_OWN_PROCESS;

    // Код ошибки при инициализации и завершения сервиса
    // не используется
    ss.dwServiceSpecificExitCode = 0;

    // Начинаем запуск сервиса.
    // Прежде всего устанавливаем состояние ожидания
    // запуска сервиса
    ReportStatus(SERVICE_START_PENDING, NO_ERROR, 4000);

    // Вызываем функцию, которая выполняет все
    // необходимые инициализирующие действия
    // ServiceStart(argc, argv);

    // После завершения инициализации устанавливаем
    // состояние работающего сервиса
    ReportStatus(SERVICE_RUNNING, NOERROR, 0);

    return;
}

// -----
// Функция ServiceControl
// Точка входа функции обработки команд
// -----
void WINAPI ServiceControl(DWORD dwControlCode)
{
    // Анализируем код команды и выполняем эту команду
    switch(dwControlCode)
    {
        // Команда остановки сервиса
        case SERVICE_CONTROL_STOP:
        {
            // Устанавливаем состояние ожидания остановки
            ss.dwCurrentState = SERVICE_STOP_PENDING;
            ReportStatus(ss.dwCurrentState, NOERROR, 0);

            // Выполняем остановку сервиса, вызывая функцию,
            // которая выполняет все необходимые для этого действия
            // ServiceStop();

            // Отмечаем состояние как остановленный сервис
            ReportStatus(SERVICE_STOPPED, NOERROR, 0);

            break;
        }

        // Определение текущего состояния сервиса
        case SERVICE_CONTROL_INTERROGATE:
        {
            // Возвращаем текущее состояние сервиса
            ReportStatus(ss.dwCurrentState, NOERROR, 0);
            break;
        }

        // В ответ на другие команды просто возвращаем
        // текущее состояние сервиса
        default:
        {
            ReportStatus(ss.dwCurrentState, NOERROR, 0);
            break;
        }
    }
}

// -----
// Функция ReportStatus
// Посылка состояния сервиса системе управления сервисами
// -----
void ReportStatus(DWORD dwCurrentState,
                 DWORD dwWin32ExitCode, DWORD dwWaitHint)
{
    // Счетчик шагов длительных операций
    static DWORD dwCheckPoint = 1;

    // Если сервис не находится в процессе запуска,
    // его можно остановить
    if(dwCurrentState == SERVICE_START_PENDING)
        ss.dwControlsAccepted = 0;
    else
        ss.dwControlsAccepted = SERVICE_ACCEPT_STOP;

    // Сохраняем состояние, переданное через
    // параметры функции
    ss.dwCurrentState = dwCurrentState;
    ss.dwWin32ExitCode = dwWin32ExitCode;
    ss.dwWaitHint = dwWaitHint;

    // Если сервис не работает и не остановлен,
    // увеличиваем значение счетчика шагов
    // длительных операций
    if((dwCurrentState == SERVICE_RUNNING) ||
        (dwCurrentState == SERVICE_STOPPED))
        ss.dwCheckPoint = 0;
    else
        ss.dwCheckPoint = dwCheckPoint++;
}

```

```
// Вызываем функцию установки состояния
SetServiceStatus(ssHandle, &ss);
}
```

В файле small.h (листинг 5.2) определено имя сервиса MYServiceName и прототипы функций.

Листинг 5.2. Файл service/small/small.h

```
#define MYServiceName "Sample of simple service"

void WINAPI ServiceMain(DWORD dwArgc, LPSTR *lpszArv);
void WINAPI ServiceControl(DWORD dwControlCode);
void ReportStatus(DWORD dwCurrentState,
    DWORD dwWin32ExitCode, DWORD dwWaitHint);
```

### Исходные тексты приложения SRVCTRL

Главный файл исходных текстов приложения SRVCTRL, предназначенного для управления сервисом Sample of simple service приведен в листинге 5.3.

Листинг 5.3. Файл service/srvctrl.c

```
// =====
// Приложение SRVCTRL
// Работа с сервисом "Sample of simple service"
//
// (C) Фролов А.В., 1996
// Email: frolov@glas.apc.org
// =====

#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "resource.h"
#include "afxres.h"

#include "srvctrl.h"

HINSTANCE hInst;
char szAppName[] = "ServiceCtlApp";
char szAppTitle[] = "Simple Service Control";

// Состояние сервиса
SERVICE_STATUS ss;

// -----
// Функция WinMain
// -----
int APIENTRY
```

```
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hWnd;
    MSG msg;

    // Сохраняем идентификатор приложения
    hInst = hInstance;

    // Проверяем, не было ли это приложение запущено ранее
    hWnd = FindWindow(szAppName, NULL);
    if(hWnd)
    {
        // Если было, выдвигаем окно приложения на
        // передний план
        if(IsIconic(hWnd))
            ShowWindow(hWnd, SW_RESTORE);
        SetForegroundWindow(hWnd);
        return FALSE;
    }

    // Регистрируем класс окна
    memset(&wc, 0, sizeof(wc));
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.hIconSm = LoadImage(hInst,
        MAKEINTRESOURCE(IDI_APPICONSM),
        IMAGE_ICON, 16, 16, 0);
    wc.style = 0;
    wc.lpfnWndProc = (WNDPROC)WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInst;
    wc.hIcon = LoadImage(hInst,
        MAKEINTRESOURCE(IDI_APPICON),
        IMAGE_ICON, 32, 32, 0);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
    wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
    wc.lpszClassName = szAppName;
    if(!RegisterClassEx(&wc))
        if(!RegisterClass((LPWNDCLASS)&wc.style))
            return FALSE;

    // Создаем главное окно приложения
    hWnd = CreateWindow(szAppName, szAppTitle,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
        NULL, NULL, hInst, NULL);
    if(!hWnd) return(FALSE);

    // Отображаем окно и запускаем цикл
```

```

// обработки сообщений
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

return msg.wParam;
}

// -----
// Функция WndProc
// -----
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
    switch(msg)
    {
        HANDLE_MSG(hWnd, WM_COMMAND, WndProc_OnCommand);
        HANDLE_MSG(hWnd, WM_DESTROY, WndProc_OnDestroy);

        default:
            return(DefWindowProc(hWnd, msg, wParam, lParam));
    }
}

// -----
// Функция WndProc_OnDestroy
// -----
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{
    PostQuitMessage(0);
    return 0L;
}

// -----
// Функция WndProc_OnCommand
// -----
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id,
        HWND hwndCtl, UINT codeNotify)
{
    // Идентификатор сервиса
    SC_HANDLE schService;

    // Идентификатор системы управления сервисами
    SC_HANDLE schSCManager;

```

```

LPQUERY_SERVICE_CONFIG lpBuf;
DWORD dwBytesNeeded;
char szBuf[1024];

switch (id)
{
    case ID_FILE_EXIT:
    {
        // Завершаем работу приложения
        PostQuitMessage(0);
        return 0L;
        break;
    }

    // Установка сервиса в систему
    case ID_SERVICE_INSTALL:
    {
        // Открываем систему управления сервисами
        schSCManager = OpenSCManager(NULL, NULL,
            SC_MANAGER_ALL_ACCESS);

        if(!schSCManager)
            break;

        // Создаем сервис с именем MYServiceName
        schService = CreateService(
            schSCManager,
            MYServiceName,
            MYServiceName,
            SERVICE_ALL_ACCESS,
            SERVICE_WIN32_OWN_PROCESS,
            SERVICE_DEMAND_START,
            SERVICE_ERROR_NORMAL,
            "c:\\ntbk2\\src\\service\\small\\debug\\small.exe",
            NULL,
            NULL,
            "",
            NULL,
            NULL);

        // Закрываем идентификатор системы управления
        // сервисами
        CloseServiceHandle(schSCManager);
        break;
    }

    // Удаление сервиса из системы
    case ID_SERVICE_REMOVE:
    {
        // Открываем систему управления сервисами
        schSCManager = OpenSCManager(NULL, NULL,
            SC_MANAGER_ALL_ACCESS);
    }
}

```

```

    if(!schSCManager)
        break;

    // Открываем сервис с именем MYServiceName
    schService = OpenService(
        schSCManager, MYServiceName,
        SERVICE_ALL_ACCESS);

    if(!schService)
        break;

    // Останавливаем сервис
    ControlService(schService,
        SERVICE_CONTROL_STOP, &ss);

    // Вызываем функцию удаления сервиса из системы
    DeleteService(schService);

    // Закрываем идентификатор системы управления
    // сервисами
    CloseServiceHandle(schSCManager);
    break;
}

case ID_SERVICE_START:
{
    // Открываем систему управления сервисами
    schSCManager = OpenSCManager(NULL, NULL,
        SC_MANAGER_ALL_ACCESS);

    if(!schSCManager)
        break;

    // Открываем сервис с именем MYServiceName
    schService = OpenService(
        schSCManager, MYServiceName,
        SERVICE_ALL_ACCESS);

    if(!schService)
        break;

    // Запускаем сервис
    StartService(schService, 0, NULL);

    // Закрываем идентификатор системы управления
    // сервисами
    CloseServiceHandle(schSCManager);
    break;
}

case ID_SERVICE_STOP:

```

```

{
    // Открываем систему управления сервисами
    schSCManager = OpenSCManager(NULL, NULL,
        SC_MANAGER_ALL_ACCESS);

    if(!schSCManager)
        break;

    // Открываем сервис с именем MYServiceName
    schService = OpenService(
        schSCManager, MYServiceName,
        SERVICE_ALL_ACCESS);

    // Останавливаем сервис
    ControlService(schService,
        SERVICE_CONTROL_STOP, &ss);

    // Закрываем идентификатор системы управления
    // сервисами
    CloseServiceHandle(schSCManager);
    break;
}

case ID_SERVICE_GETCONFIGURATION:
{
    // Открываем систему управления сервисами
    schSCManager = OpenSCManager(NULL, NULL,
        SC_MANAGER_ALL_ACCESS);

    if(!schSCManager)
        break;

    // Открываем сервис с именем MYServiceName
    schService = OpenService(
        schSCManager, MYServiceName,
        SERVICE_ALL_ACCESS);

    if(!schService)
        break;

    // Получаем буфер для сохранения конфигурации
    lpBuf = (LPQUERY_SERVICE_CONFIG)malloc(4096);

    if(lpBuf != NULL)
    {
        // Сохраняем конфигурацию в буфере
        QueryServiceConfig(schService,
            lpBuf, 4096, &dwBytesNeeded);

        // Отображаем некоторые поля конфигурации
        wsprintf(szBuf, "Binary path: %s\n"
            "Start Name: %s\n"

```



```

        "Display Name: %s\n",
        lpBuf->lpBinaryPathName,
        lpBuf->lpServiceStartName,
        lpBuf->lpDisplayName);

    MessageBox(hWnd, szBuf, szAppTitle,
        MB_OK | MB_ICONINFORMATION);

    // Освобождаем буфер
    free(lpBuf);
}

// Закрываем идентификатор системы управления
// сервисами
CloseServiceHandle(schSCManager);
break;
}

case ID_HELP_ABOUT:
{
    MessageBox(hWnd,
        "Simple Service Control\n"
        "(C) Alexandr Frolov, 1996\n"
        "Email: frolov@glas.apc.org",
        szAppTitle, MB_OK | MB_ICONINFORMATION);
    return 0L;
    break;
}

default:
    break;
}

return FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify,
    DefWindowProc);
}

```

В файле `srvctrl.h` (листинг 5.4) определено имя сервиса и прототипы функций.

#### Листинг 5.4. Файл `service/srvctrl.h`

```

// Имя сервиса
#define MYServiceName "Sample of simple service"

// -----
// Описание функций
// -----

LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

void WndProc_OnCommand(HWND hWnd, int id,
    HWND hwndCtl, UINT codeNotify);

```

```

void WndProc_OnDestroy(HWND hWnd);

LRESULT WINAPI
DlgProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);

BOOL DlgProc_OnInitDialog(HWND hdlg, HWND hwndFocus,
    LPARAM lParam);

void DlgProc_OnCommand(HWND hdlg, int id,
    HWND hwndCtl, UINT codeNotify);

```

Файл `resource.h` (листинг 5.5) содержит описания констант, которые используются в файле определения ресурсов приложения.

#### Листинг 5.5. Файл `service/resource.h`

```

//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by srvctrl.rc
//
#define IDR_MENU1 101
#define IDR_APPMENU 101
#define IDI_APPICON 102
#define IDI_APPICONSM 103
#define ID_FILE_EXIT 40001
#define ID_HELP_ABOUT 40002
#define ID_SERVICE_INSTALL 40010
#define ID_SERVICE_REMOVE 40011
#define ID_SERVICE_START 40012
#define ID_SERVICE_STOP 40013
#define ID_SERVICE_GETCONFIGURATION 40014

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 105
#define _APS_NEXT_COMMAND_VALUE 40015
#define _APS_NEXT_CONTROL_VALUE 1006
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif

```

Файл определения ресурсов приложения приведен в листинге 5.6.

#### Листинг 5.6. Файл `service/srvctrl.rc`

```

//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS

```

[illegible]

## ЗАКЛЮЧЕНИЕ

Итак, на этом мы завершаем 27 том “Библиотеки системного программиста”, посвященный программированию для операционной системы Microsoft Windows NT.

access control list, 58  
 ACCESS\_SYSTEM\_SECURITY, 58  
 ACL, 58  
 ActivateKeyboardLayout, 106; 107; 112; 113; 116  
 Anonymous Pipes, 56  
 AnsiToOem, 24  
 AnsiToOemBuff, 24  
 CallNamedPipe, 62; 63  
 CD-ROM, 26; 27; 29; 35; 40  
 ChangeServiceConfig, 123  
 CharToOem, 24  
 CharToOemBuff, 24  
 Clipboard, 41  
 CloseHandle, 10; 16; 19; 23; 24; 26; 42; 44; 46; 62; 63; 68; 70; 75; 76  
 COFF, 85  
 ConnectNamedPipe, 59; 60; 62; 67; 68; 69  
 CONSTANT, 82; 83; 85  
 ControlService, 123; 125; 126; 131  
 COPYDATASTRUCT, 46; 47; 49; 50; 52; 54; 55  
 CREATE\_ALWAYS, 61  
 CREATE\_NEW, 61  
 CreateFile, 7; 8; 9; 10; 15; 16; 19; 23; 26; 41; 42; 43; 44; 59; 60; 61; 62; 63; 65; 70; 72; 75; 76  
 CreateFileMapping, 8; 9; 10; 19; 26; 41; 42; 43; 44  
 CreateNamedPipe, 57; 59; 60; 62; 63; 64; 67; 68  
 CreatePipe, 57  
 CreateService, 122; 123; 125; 126; 130  
 DATE\_LONGDATE, 105  
 DATE\_SHORTDATE, 105  
 DATE\_USE\_ALT\_CALENDAR, 105  
 DDE, 41; 61  
 DeleteService, 125; 131  
 DisconnectNamedPipe, 62  
 DISKINFO, 29; 30; 33; 34; 37; 38; 39; 40  
 DLL, 77  
 DLL\_PROCESS\_ATTACH, 81; 82; 84; 88  
 DLL\_PROCESS\_DETACH, 81; 82; 84; 88  
 DLL\_THREAD\_ATTACH, 81; 82; 88  
 DLL\_THREAD\_DETACH, 81; 82; 84; 88  
 DLLEntryPoint, 81; 82; 84; 88; 89  
 Dynamic-Link Libraries, 77  
 EnumDependentServices, 123  
 EnumServicesStatus, 122  
 EnumWindows, 89  
 EnumWindowsProc, 89  
 ERROR\_ALREADY\_EXISTS, 9; 43; 62

Конечно, в объеме двух книг невозможно охватить все вопросы программирования для этой весьма сложной современной операционной системы. При необходимости вы должны будете обращаться к документации, которая поставляется в составе SDK или к справочной системе, встроенной в систему разработки приложений Microsoft Visual C++. Рекомендуем вам также прочитать 24 том “Библиотеки системного программиста”, которые называются “Microsoft Visual C++ и MFC. Программирование для Windows 95 и Windows NT”.

ERROR\_HANDLE\_EOF, 17; 18; 25  
 ERROR\_IO\_PENDING, 17; 18; 25  
 ERROR\_NO\_DATA, 60  
 ERROR\_PIPE\_CONNECTED, 60  
 ERROR\_PIPE\_LISTENING, 60  
 ERROR\_SERVICE\_SPECIFIC\_ERROR, 121  
 EXPORTS, 82  
 FILE\_ATTRIBUTE\_ARCHIVE, 61  
 FILE\_ATTRIBUTE\_COMPRESSED, 61  
 FILE\_ATTRIBUTE\_HIDDEN, 61  
 FILE\_ATTRIBUTE\_NORMAL, 61  
 FILE\_ATTRIBUTE\_READONLY, 61  
 FILE\_ATTRIBUTE\_SYSTEM, 61  
 FILE\_FLAG\_BACKUP\_SEMANTICS, 61  
 FILE\_FLAG\_DELETE\_ON\_CLOSE, 61  
 FILE\_FLAG\_NO\_BUFFERING, 61  
 FILE\_FLAG\_OVERLAPPED, 61  
 FILE\_FLAG\_POSIX\_SEMANTICS, 61  
 FILE\_FLAG\_RANDOM\_ACCESS, 61  
 FILE\_FLAG\_SEQUENTIAL\_SCAN, 61  
 FILE\_FLAG\_WRITE\_THROUGH, 58; 61  
 FILE\_MAP\_ALL\_ACCESS, 10  
 FILE\_MAP\_COPY, 10  
 FILE\_MAP\_READ, 10; 42; 43; 45; 46  
 FILE\_MAP\_WRITE, 10; 19; 26; 41; 42; 43; 45; 46  
 FILE\_SHARE\_READ, 8; 16; 23; 60; 72; 75; 76  
 FILE\_SHARE\_WRITE, 8; 60  
 FindWindow, 13; 22; 30; 48; 49; 52; 53; 54; 92; 109  
 FlushViewOfFile, 11  
 FreeLibrary, 81; 84; 94; 98  
 gdi.exe, 78  
 gdi32.dll, 78  
 GENERIC\_READ, 7; 8; 15; 16; 23; 60; 62; 70  
 GENERIC\_WRITE, 7; 8; 16; 23; 60; 62; 70; 72; 75; 76  
 GetDateFormat, 104; 105; 111; 115  
 GetDiskFreeSpace, 33; 35; 39; 40  
 GetDlgItemText, 98  
 GetFileSize, 19; 26  
 GetKeyboardLayout, 105; 106; 107; 109; 111; 115  
 GetKeyboardLayoutList, 105; 106; 107; 109; 115  
 GetKeyboardLayoutName, 106; 111; 115  
 GetLastError, 9; 17; 18; 25; 43; 44; 45; 57; 59; 60; 62; 63; 64; 65; 66; 67; 68; 69; 70; 71; 73; 74; 75; 84; 99; 111  
 GetLocaleInfo, 102; 103; 106; 111; 112; 115  
 GetLogicalDriveStrings, 34; 39; 40  
 GetMailslot, 72; 73; 74; 75

GetNamedPipeHandleState, 62; 65; 66  
 GetNamedPipeInfo, 62; 66  
 GetOverlappedResult, 18; 25  
 GetProcAddress, 84; 85; 94; 97  
 GetSystemDefaultLangID, 102  
 GetSystemDefaultLCID, 102  
 GetSystemInfo, 9  
 GetThreadLocals, 101  
 GetTimeFormat, 103; 104; 105; 111; 115  
 GetUserDefaultLangID, 102  
 GetVolumeInformation, 33; 35; 39; 40  
 GetWindowText, 90  
 GMEM\_SHARE, 79  
 HKL, 105  
 import library, 83  
 INVALID\_HANDLE\_VALUE, 8; 16; 59; 62; 67; 70; 71; 74; 75  
 kernel32.dll, 78  
 KL\_NAMELENGTH, 106; 110  
 KLF\_ACTIVATE, 106  
 KLF\_REORDER, 106; 107  
 KLF\_SUBSTITUTE\_OK, 106  
 KLF\_UNLOADPREVIOUS, 106; 107  
 krm1286.exe, 78  
 krm1386.exe, 78  
 LANG\_NEUTRAL, 101  
 LANGIDFROMLCID, 101  
 LCID, 99; 101; 102; 103; 104; 106; 110; 111; 112; 115  
 LibEntry, 80  
 LibMain, 81  
 LIBRARY, 85  
 LoadKeyboardLayout, 106  
 LoadLibrary, 81; 83; 84; 85; 94; 97  
 LOCALE\_NOUSEROVERRIDE, 104; 105  
 LOCALE\_SYSTEM\_DEFAULT, 99  
 LOCALE\_USER\_DEFAULT, 99  
 localtime, 54  
 LockServiceDatabase, 123  
 Mailslot, 41; 60; 71; 72; 73; 74; 75; 76  
 MAILSLOT\_NO\_MESSAGE, 73  
 MAILSLOT\_WAIT\_FOREVER, 71; 72  
 MAKEINTRESOURCE, 13; 14; 22; 30; 31; 48; 53; 84; 93; 110; 111  
 MAKELANGID, 99  
 MAKELCID, 99; 101; 106; 110; 111; 112; 115  
 MapViewOfFile, 9; 10; 19; 26; 41; 42; 43; 44; 45; 46  
 MapViewOfFileEx, 9; 10  
 Microsoft Development Library, 57  
 MoveWindow, 34; 48; 50  
 MSDN, 57  
 Mutex, 41  
 Named Pipes, 56  
 NM\_DBLCLK, 40  
 NMPWAIT\_NOWAIT, 63  
 NMPWAIT\_USE\_DEFAULT\_WAIT, 63; 64  
 NMPWAIT\_WAIT\_FOREVER, 63; 64  
 NONAME, 82; 83; 85  
 OemToAnsi, 24

OemToAnsiBuff, 24  
 OemToChar, 24  
 OemToCharBuff, 24  
 OLE, 41  
 OPEN\_ALWAYS, 61  
 OPEN\_EXISTING, 61  
 OpenFileMapping, 10; 42; 45; 46  
 OpenSCManager, 122; 123; 125; 126; 130; 131  
 OpenService, 122; 125; 126; 131; 132  
 OVERLAPPED, 13; 16; 17; 23; 24; 30; 53; 58; 59; 61; 63; 64; 93; 110  
 PAGE\_READONLY, 8  
 PAGE\_READWRITE, 8; 10; 19; 26; 41; 42; 43  
 PAGE\_WRITECOPY, 8; 10  
 PeekNamedPipe, 62; 64  
 PIPE\_ACCESS\_DUPLEX, 58  
 PIPE\_ACCESS\_INBOUND, 58  
 PIPE\_ACCESS\_OUTBOUND, 58  
 PIPE\_CLIENT\_END, 66  
 PIPE\_NOWAIT, 58; 59; 65  
 PIPE\_READMODE\_BYTE, 58; 59; 65  
 PIPE\_READMODE\_MESSAGE, 58; 59; 60; 65; 67; 69  
 PIPE\_SERVER\_END, 66  
 PIPE\_TYPE\_BYTE, 59  
 PIPE\_TYPE\_MESSAGE, 58; 59; 60; 66; 67; 69  
 PIPE\_UNLIMITED\_INSTANCES, 59; 66  
 PIPE\_WAIT, 58; 65  
 PostMessage, 46  
 PostQuitMessage, 50  
 PRIMARYLANGID, 101  
 putch, 44  
 QUERY\_SERVICE\_CONFIG, 126  
 QueryServiceConfig, 123; 126  
 QueryServiceLockStatus, 123  
 QueryServiceStatus, 123  
 ReadFile, 17; 24; 25; 41; 56; 57; 59; 62; 63; 64; 68; 69; 70; 71; 72; 73; 74; 75  
 regedt32.exe, 117  
 RegisterServiceCtrlHandler, 119; 120; 128  
 SC\_HANDLE, 122; 123; 125; 126; 130  
 SC\_MANAGER\_ALL\_ACCESS, 122  
 SC\_MANAGER\_CONNECT, 122  
 SC\_MANAGER\_CREATE\_SERVICE, 122  
 SC\_MANAGER\_ENUMERATE\_SERVICE, 122  
 SC\_MANAGER\_LOCK, 123  
 SC\_MANAGER\_QUERY\_LOCK\_STATUS, 123  
 SEC\_COMMIT, 8  
 SEC\_IMAGE, 8  
 SEC\_NOCACHE, 8  
 SEC\_RESERVE, 8  
 SECURITY\_ATTRIBUTES, 7; 60  
 SEM\_FAILCRITICALERRORS, 85  
 SEM\_NOGPFAULTERRORBOX, 85  
 SEM\_NOOPENFILEERRORBOX, 85  
 SEM\_NOOPENFILEERRORBOX, 85  
 SendMessage, 46; 54; 55; 112; 113; 116  
 SERVICE\_ACCEPT\_PAUSE\_CONTINUE, 121  
 SERVICE\_ACCEPT\_SHUTDOWN, 121

SERVICE\_ACCEPT\_STOP, 121; 122; 129  
SERVICE\_ALL\_ACCESS, 123  
SERVICE\_AUTO\_START, 124  
SERVICE\_BOOT\_START, 124  
SERVICE\_CHANGE\_CONFIG, 123  
SERVICE\_CONTINUE\_PENDING, 121  
SERVICE\_CONTROL\_CONTINUE, 120; 125  
SERVICE\_CONTROL\_INTERROGATE, 120; 125  
SERVICE\_CONTROL\_PAUSE, 120; 125  
SERVICE\_CONTROL\_SHUTDOWN, 120  
SERVICE\_CONTROL\_STOP, 120; 125  
SERVICE\_DEMAND\_START, 124  
SERVICE\_DISABLED, 124  
SERVICE\_ENUMERATE\_DEPENDENTS, 123  
SERVICE\_ERROR\_CRITICAL, 124  
SERVICE\_ERROR\_IGNORE, 124  
SERVICE\_ERROR\_NORMAL, 124  
SERVICE\_ERROR\_SEVERE, 124  
SERVICE\_FILE\_SYSTEM\_DRIVER, 121; 124  
SERVICE\_INTERACTIVE\_PROCESS, 121; 124  
SERVICE\_INTERROGATE, 123  
SERVICE\_KERNEL\_DRIVER, 121; 124  
SERVICE\_PAUSE\_CONTINUE, 123  
SERVICE\_PAUSE\_PENDING, 121  
SERVICE\_PAUSED, 121  
SERVICE\_QUERY\_CONFIG, 123  
SERVICE\_QUERY\_STATUS, 123  
SERVICE\_RUNNING, 120; 121; 122; 128; 129  
SERVICE\_START, 123  
SERVICE\_START\_PENDING, 120; 121; 122; 128; 129  
SERVICE\_STATUS, 119; 120; 121; 122; 123; 125; 127; 129  
SERVICE\_STOP, 123  
SERVICE\_STOP\_PENDING, 121  
SERVICE\_STOPPED, 121  
SERVICE\_SYSTEM\_START, 124  
SERVICE\_TABLE\_ENTRY, 119; 127  
SERVICE\_USER\_DEFINE\_CONTROL, 123  
SERVICE\_WIN32\_OWN\_PROCESS, 121; 124  
SERVICE\_WIN32\_SHARE\_PROCESS, 121; 124  
SetErrorMode, 85  
SetMailslot, 73  
SetNamedPipeHandleState, 62; 64; 65  
SetServiceStatus, 120; 122; 129  
SetThreadLocale, 99; 101; 110; 111; 115  
Sleep, 75  
SORT\_DEFAULT, 99

**АННОТАЦИЯ..... 2**

**ВВЕДЕНИЕ..... 3**

**БЛАГОДАРНОСТИ ..... 4**

StartService, 119; 123; 124; 125; 126; 128; 131  
StartServiceCtrlDispatcher, 119; 128  
strchr, 34; 35; 40  
SUBLANG\_DEFAULT, 101  
SUBLANG\_NEUTRAL, 101  
SUBLANG\_SYS\_DEFAULT, 101  
SUBLANGID, 101; 102  
SYSTEM\_INFO, 9  
SYSTEMTIME, 104  
time, 54  
TIME\_FORCE24HOURFORMAT, 104  
TIME\_NOMINUTESORSECONDS, 104  
TIME\_NOSECONDS, 104  
TIME\_NOTIMEMARKER, 104  
TransactNamedPipe, 62; 63; 64  
TRUNCATE\_EXISTING, 61  
UnloadKeyboardLayout, 106  
UnmapViewOfFile, 10; 19; 26; 42; 44; 46  
user32.dll, 78  
VirtualAlloc, 8; 10  
WaitForMultipleObjects, 44  
WaitForSingleObject, 18; 25  
WaitNamedPipe, 62; 64  
WEP, 81  
WM\_COPYDATA, 41; 46; 47; 48; 49; 50; 52; 54; 55  
WM\_TIMER, 54  
World Wide Web, 117  
WRITE\_DAC, 58  
WRITE\_OWNER, 58  
WriteFile, 17; 18; 24; 25; 41; 56; 57; 59; 62; 63; 68; 69; 70; 71; 72; 76  
WriteFileEx, 57  
WS\_POPUPWINDOW, 50  
WS\_THICKFRAME, 50  
WWW, 117  
библиотека импорта, 83  
виртуальная память, 6  
гранулярность памяти, 9  
динамический импорт функций, 83  
импортирование функций, 83  
пиктограмма Regional Settings, 99  
пользователь LocalSystem, 125  
структура DLL-библиотеки, 80  
структура DLL-библиотеки, 78  
утилита tdump.exe, 85  
файл определения модуля для DLL-библиотеки, 85  
экспортируемые функции, 82

**КАК СВЯЗАТЬСЯ С АВТОРАМИ..... 5**

**1 СНОВА О ФАЙЛАХ..... 6**

Файлы, отображаемые на память..... 6

Создание отображения файла ..... 7

Выполнение отображения файла в память ..... 9

Открытие отображения .....	10
Отмена отображения файла .....	10
Принудительная запись измененных данных .....	10
Приложение Oem2Char .....	10
Исходные тексты приложения .....	12
Определения и глобальные переменные .....	20
Описание функций .....	21
Приложение DiskInfo .....	25
Исходные тексты приложения .....	28
Определения и глобальные переменные .....	37
Описание функций .....	37
<b>2 ПЕРЕДАЧА ДАННЫХ МЕЖДУ ПРОЦЕССАМИ .....</b>	<b>41</b>
Обмен через файлы, отображаемые на память .....	41
Приложение Fmap/Server .....	42
Приложение Fmap/Client .....	44
Передача сообщений между приложениями .....	46
Приложение RCLOCK .....	47
Приложение STIME .....	52
Каналы передачи данных Pipe .....	56
Именованные и анонимные каналы .....	56
Имена каналов .....	56
Реализации каналов .....	56
Функции для работы с каналами .....	57
Примеры приложений .....	66
Каналы передачи данных Mailslot .....	70
Создание канала Mailslot .....	70
Открытие канала Mailslot .....	71
Запись сообщений в канал Mailslot .....	71
Чтение сообщений из канала Mailslot .....	72
Определение состояния канала Mailslot .....	72
Изменение состояния канала Mailslot .....	72
Примеры приложений .....	73
<b>3 БИБЛИОТЕКИ ДИНАМИЧЕСКОЙ КОМПОНОВКИ .....</b>	<b>76</b>
Статическая и динамическая компоновка .....	76
DLL-библиотеки в операционной системе Windows NT .....	77
Отображение страниц DLL-библиотеки .....	78
Обмен данными между приложениями через DLL-библиотеку .....	79
Как работает DLL-библиотека .....	79
Инициализация DLL-библиотеки в среде Microsoft Windows NT .....	80
Экспортирование функций и глобальных переменных .....	81
Импортирование функций .....	82
Файл определения модуля для DLL-библиотеки .....	84

Анализ DLL-библиотек при помощи программы dumpbin.exe .....	84
Исходные тексты DLL-библиотеки DLLDEMO .....	86
Приложение DLLCALL .....	89
Глобальные переменные и определения .....	95
Функция WinMain .....	95
Функция WndProc .....	95
Функция WndProc_OnDestroy .....	95
Функция WndProc_OnCommand .....	96
ФункцияDlgProc .....	96
ФункцияDlgProc_OnInitDialog .....	96
ФункцияDlgProc_OnCommand .....	96
<b>4 НАЦИОНАЛЬНЫЕ ПАРАМЕТРЫ .....</b>	<b>98</b>
Наборы национальных параметров .....	98
Установка набора национальных параметров .....	98
Определение национального набора параметров .....	100
Определение отдельных национальных параметров .....	101
Форматное преобразование даты и времени .....	102
Преобразование времени .....	102
Преобразование даты .....	103
Изменение раскладки клавиатуры .....	104
Получение списка установленных раскладок .....	104
Определение названия текущей раскладки клавиатуры .....	104
Определение идентификатора раскладки клавиатуры для задачи .....	105
Загрузка раскладки клавиатуры .....	105
Выгрузка раскладки клавиатуры .....	105
Переключение раскладки клавиатуры .....	105
Приложение SETLOCAL .....	105
Исходные тексты приложения SETLOCAL .....	108
Описание функций .....	114
<b>5 СЕРВИСНЫЕ ПРОЦЕССЫ .....</b>	<b>115</b>
Создание сервисного процесса .....	117
Функция main сервисного процесса .....	117
Точка входа сервиса .....	117
Функция обработки команд .....	118
Состояние сервиса .....	118
Управление сервисами .....	120
Получение идентификатора системы управления сервисами .....	120
Установка сервиса .....	121
Получение идентификатора сервиса .....	123
Выдача команд сервису .....	123
Удаление сервиса из системы .....	123
Запуск сервиса .....	123

---

Остановка сервиса .....	123
Определение конфигурации сервиса .....	123
Приложение SRVCTRL .....	124
Исходный текст сервиса .....	125
Исходные тексты приложения SRVCTRL .....	127
<b>ЗАКЛЮЧЕНИЕ .....</b>	<b>132</b>